An Integration of Z and Timed CSP for Specifying Real-Time Embedded Systems

vorgelegt von
Diplom-Informatiker
Carsten Sühl
aus Berlin

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr.-Ing. Martin Buss
Berichter: Prof. Dr.-Ing. Stefan Jähnichen
Berichterin: Prof. Dr. rer. nat. Maritta Heisel

Tag der wissenschaftlichen Aussprache: 17. Dezember 2002

Berlin 2002
D 83

Zusammenfassung

Fortschritte in der Entwicklung der Hardwaretechnologie haben zu einem Vordringen von Software in neue Anwendungsbereiche und zu ihrem Einsatz zur Lösung immer komplexerer Probleme geführt. Dies trifft insbesondere auf den Bereich eingebetteter Systeme zu.

Viele eingebettete Systeme, vor allem im Bereich der Prozesskontrolle, sind sicherheitskritisch, d.h. sie stellen eine potentielle Gefahr für das Eigentum oder sogar das Leben von Menschen dar. Weil von solchen Systemen ein hoher Grad an Zuverlässigkeit verlangt wird, besitzt die Verwendung formaler Techniken in diesem Anwendungsbereich im Vergleich zu nicht sicherheitskritischen Systemen, bei denen ihr adäquater Einsatz lediglich zu einer allgemeinen Qualitätsverbesserung führt, ein noch größeres Potential. Trotz dieser Vorzüge besitzen die etablierten formalen Spezifikationssprachen verschiedene Restriktionen, welche ihre Nützlichkeit für die Entwicklung eingebetteter Systeme einschränken. Unter anderem diese Erkenntnis hat dazu geführt, dass sich das Forschungsgebiet der „Integration formaler Methoden“ in jüngster Zeit wachsenden Interesses erfreut.

Die vorliegende Arbeit hat eine solche Integration von zwei formalen Techniken, der modellbasierten Notation Z und der Echtzeitprozessalgebra Timed-CSP, zum Gegenstand. Z ist eine verbreitete Notation zur Spezifikation datenbezogener Eigenschaften transformationeller Systeme, jedoch nicht zur Modellierung von Verhaltensaspekten geeignet. Timed-CSP hingegen, eine Erweiterung der Prozessalgebra CSP um Echtzeit, ermöglicht die Definition von Verhalten inklusive Echtzeitaspekten, unterstützt allerdings nicht die Modellierung von Datentypen. Ziel dieser Arbeit ist die Ausnutzung der Stärken der beiden komplementären Formalismen im Rahmen eines Integrationsformalismus', der als RT-Z bezeichnet wird. RT-Z unterstützt den Entwicklungsprozess in verschiedenen Phasen — von der Anforderungsspezifikation über den Architektur- bis zum Detailentwurf — und erlaubt eine kohärente, formale Spezifikation aller relevanten Aspekte eines eingebetteten Echtzeitsystems.

Im Folgenden seien die vier wesentlichen Charakteristika von RT-Z skizziert.

Zur Ausnutzung der oben genannten Vorzüge formaler Techniken ist RT-Z vollständig formal fundiert, was die Syntax, die Semantik und den Verfeinerungsbegriff, der die Grundlage des Prozesses der schrittweisen Entwicklung von Spezifikationen bildet, einschließt.

Die meisten eingebetteten Systeme bestehen aus einer Vielzahl stark interdependenter Komponenten. Um diese inhärente Komplexität bewältigen zu können, stellt RT-Z eigene, formal fundierte Strukturierungskonzepte bereit. Dies ist notwendig, weil beide Basisformalismen nicht über adäquate Strukturierungsmechanismen verfügen.

Weiterhin unterstützt RT-Z sowohl die Anforderungs- als auch die Entwurfsphasen. RT-Z verfügt zu diesem Zweck über Sprachkonstrukte auf verschiedenen Abstraktionsebenen: abstrakte Konstrukte zur Spezifikation von Anforderungen, ohne dabei gleichzeitig Implementierungsdetails festzulegen, und konkrete Konstrukte zur Festlegung solcher Implementierungsdetails im Entwurf.

Schließlich ist eine herausragende Zielsetzung bei der Integration etablierter Techniken die Wiederverwendbarkeit ihrer Infrastruktur, z.B. Werkzeuge. RT-Z ist als „konservierende Integration“ konzipiert und stellt einen optimalen Kompromiss zwischen oben genannter Wiederverwendbarkeit und der Kohärenz der Teile einer Spezifikation dar.

Die Eignung von RT-Z — im Vergleich zu anderen Formalismen — für den anvisierten Anwendungsbereich folgt aus dem Zusammenwirken der Gesamtheit dieser Eigenschaften.

Acknowledgements

First of all, I would like to thank my advisers. Stefan Jähnichen, who employed me as a PhD student at GMD FIRST, gave me the freedom to develop my own research interests and to choose my PhD topic. Also, the opportunity to work in interesting projects allowed me to gain insight into current research issues. Maritta Heisel, who introduced me to research, supported my work from the beginning and provided me with important advice and feedback for developing this thesis.

I am also grateful to my colleagues from the software engineering groups at Fraunhofer FIRST (formerly GMD FIRST) and the Technical University of Berlin for their interest and support. Matthias Anlauff, Jochen Burghardt, Steffen Helke, Stephan Herrmann, Florian Kammüller, Jeff Sanders, Thomas Santen, Dirk Seifert, Graeme Smith and Kirsten Winter helped me obtain an understanding of formal methods and software engineering in general. They allowed me to benefit from their expertise and experience.

Thomas Santen deserves special thanks for relieving me in the QUASAR project during the final stage of my dissertation as far as possible, allowing me to finish the work presented in this thesis.

I am particularly grateful to Graeme Smith for reading the long chapter dealing with the semantics of RT-Z and for giving me important advice how to make the definition of the semantics conforming to my informal description of RT-Z.

Many thanks to Jochen Burghardt, who assisted me in improving the overview of timed CSP.

Discussions with Jan Brederecke helped me improve my understanding of and clear some misconceptions concerning CSP-OZ.

Last, but not least, Wolfgang Grieskamp's tool ZETA was extremely useful for typesetting and checking the formal definitions.

Contents

I Introduction **1**

1 Envisaged Application Area **7**

2 Underlying System Model and Development Process **9**

2.1 Development Process 9
2.2 System Model 11
2.2.1 System Requirements and Design Specifications 12
2.2.2 Software Requirements Specifications 13
2.2.3 Software Design Specifications 14

3 Objectives **15**

3.1 Restrictions 16

4 State of the Art: Related Work **17**

4.1 CSP-OZ 17
4.2 TCOZ 20
4.3 Object-Z / CSP 21
4.4 LOTOS 23
4.5 RAISE Specification Language 24
4.6 Temporal Logic of Actions 25
4.7 Others 26

5 Classification **31**

II RT-Z: An Integration of Z and Timed CSP **35**

6 Integration Principles **39**

6.1 Specifying Properties (Abstract Model of Integration) 39

6.2 Specifying Models (Concrete Model of Integration) 45

7 Specification Units **53**

7.1 Concrete Specification Units 54

7.2 Abstract Specification Units 61

8 Structuring Mechanisms **65**

8.1 Aggregation 66

8.1.1 Example 66

8.1.2 Global Invariants 68

8.1.3 Syntax 71

8.1.4 Simple Aggregation 72

8.1.5 Indexed Aggregation 78

8.2 Extension 83

8.3 Renaming 94

8.4 Hiding 94

8.5 Parametrisation 94

8.6 Example: Alternating Bit Protocol 95

III Formal Foundation **101**

9 Denotational Semantics **103**

9.1 Basics 103

9.2 Concrete Specification Units (Open System View) 105

9.2.1 Concurrency on Common Data State 105

9.2.2 Semantic Integration: Overview 109

9.2.3 History Model 111

9.2.4 Extended Timed Failures Model (ETFM) 122

9.2.5 Timed Failures/States Model (TFSM) 141

9.2.6 Semantic Integration: Definition 144

9.3 Abstract Specification Units (Open System View) 146

9.4 Closed System View 147

9.4.1 Concrete Specification Units 147

9.4.2 Abstract Specification Units 147

9.5 Definedness of Recursive Process Equations 148

10 Refinement 151
10.1 Refining Single Specification Units . . . 152
10.1.1 Retrieve Specification Units . . . 152
10.1.2 Relating Timed Observations of a Refined and a Refining Unit . . . 154
10.1.3 Closed System View . . . 156
10.1.4 Open System View . . . 157
10.1.5 Techniques for Establishing Refinement . . . 158
10.2 Refining Compound Specification Units . . . 160
10.2.1 Aggregation . . . 161
10.2.2 Extension . . . 163
10.3 Relationship between Simulation and Refinement . . . 163

IV Discussion 169

11 Comparison with Related Formalisms 171
11.1 CSP-OZ . . . 171
11.2 TCOZ . . . 176
11.3 Object-Z / CSP . . . 180
11.4 LOTOS . . . 180

12 Case Studies 183
12.1 Multi-lift System . . . 183
12.1.1 Problem Description and System Architecture . . . 183
12.1.2 Requirements Specification . . . 185
12.1.3 Design Specification . . . 189
12.1.4 Discussion . . . 205
12.2 Gas Burner . . . 207
12.2.1 System Requirements . . . 207
12.2.2 System Design . . . 209
12.2.3 Software Requirements . . . 212
12.2.4 Software Design . . . 214
12.2.5 Discussion . . . 220

13 Conclusions 223
13.1 Contributions . . . 223
13.2 Satisfaction of Objectives . . . 226
13.3 Future Work . . . 226

V Appendices 231

A Base Formalisms 233
A.1 Z Notation . . . 233
A.1.1 Standardisation . . . 233
A.1.2 Discussion . . . 235
A.2 Timed CSP . . . 235
A.2.1 Computational Model . . . 237
A.2.2 Process Term Language and Operational Semantics . . . 238
A.2.3 Timed Transition Systems . . . 244
A.2.4 Denotational Semantics . . . 247
A.2.5 Predicate Language . . . 258
A.2.6 Verification . . . 260
A.2.7 Discussion . . . 262
A.2.8 Other Approaches to Modelling and Reasoning about Real-Time . . . 263

B Satisfaction of TFM Properties 265
B.1 Lemmata . . . 265
B.2 Properties of the Timed Failures Model . . . 267

C Relationship between Simulation and Refinement 291

D RT-Z Syntax 303

E Glossary of Timed CSP Notation 311

Bibliography 313

Index of Terms 321

Index of Semantic Definitions 325

Part I

Introduction

Recent advances in hardware technology have led to the introduction of software into new application domains and, within particular domains, to its use to solve increasingly complex problems; this holds particularly for embedded systems. The relevance of the domain of embedded systems is increasing, because it occupies a growing portion of the market. It is estimated [Gupta, 2002] that the market for embedded computer systems will surpass the market for personal computers until the end of this decade.

Many embedded systems, e.g., in the area of process control, are of a safety-critical nature, i.e., they have the potential to threat the safety of property or even of human life. Because of the high degree of reliability required of such systems, the use of formal methods is beneficial in this application domain. Sometimes it is even mandatory. The UK Ministry of Defence [1997], e.g., issued a standard concerning the procurement of safety-critical software in defence equipment. This standard mandates the extensive use of formal methods.

But there are also economical reasons for employing formal methods: Potter et al. [1996] have pointed out that "the bulk of the costs of a software project are related to the fixing of errors at the implementation and test stages and that a very large percentage of these errors are traceable to imprecision in the early stages of a project." Consequently, techniques able to eliminate imprecision and to introduce clarity and rigour at those early stages are likely to lead to a reduced error rate and hence to substantial cost savings. The use of formal methods can assist in achieving these goals. Their mathematical foundation makes it possible to state requirements and designs in a clear and unambiguous way and to reason about their properties rigorously. Following Clarke and Wing [1996], we use the term *formal methods* throughout this thesis in order to refer to mathematically-based languages, techniques and tools for specifying and verifying systems.

Despite the merits generally attributed to formal methods, the established formal specification languages exhibit limitations that restrict their usefulness for the development of software in embedded systems. This observation, among others, has caused the integration of formal methods to become a research field of growing interest. This is witnessed, e.g., by the establishment of a series of international conferences dedicated to this research topic [Grieskamp et al., 2000, Butler et al., 2002, Ehrig and Große-Rhode, 2002], but also by the funding of research projects such as ESPRESS [1998], UNIFORM [1998] and SoftSpez [2000].

The present thesis deals with such an integration of two established formal specification languages: the state-based specification language Z and the real-time process algebra timed CSP. The integrated formalism, called RT-Z, is aimed at formally specifying all relevant aspects of real-time embedded systems in a coherent manner. RT-Z is designed to support the development process of embedded systems in different phases—from the requirements engineering via the architectural to the detailed design.

The Z notation [Woodcock and Davies, 1996] is a formal language for specifying data-related characteristics of transformational systems, which is in widespread use in academia and—to a restricted extent—also in industry. However, it is not designed to model behaviour. Timed CSP [Schneider, 1999b], a real-time extension of the process algebra CSP, on the other hand, is a powerful language for specifying behaviour, including real-time aspects. However, it does not provide adequate constructs to model data. The formalisms can hence be considered as complementary, covering disjoint aspects of real-time embedded systems. Further, their underlying concepts are well suited to each other. RT-Z exploits the strengths of both formal-

isms in order to provide a smoothly integrated, single formalism to model all relevant facets of real-time embedded systems.

Let us stress four essential characteristics of RT-Z.

First, to exploit the abovementioned benefits of formal methods, the integrated formalism RT-Z is defined completely formally, including its syntax and semantics. Further, its formally defined notion of refinement enables the stepwise development of RT-Z specifications towards implementations.

Second, most embedded systems are fairly complex systems, containing a large number of highly interdependent system components. An appropriate specification language must provide adequate structuring (and encapsulation) mechanisms to cope with this inherent complexity; they are introduced and backed up formally by RT-Z on top of the integration of the notations of Z and timed CSP, because both base formalisms are lacking corresponding mechanisms.

Third, as already indicated, the aim of RT-Z is to support the requirements and the design phases of the development process. To this end, it provides two kinds of language constructs: *abstract* constructs to specify requirements without fixing aspects of the final implementation and *concrete* constructs to fix such implementation aspects in the architectural and detailed design.

Last, but not least, a crucial rationale of integrating existing, well-established formalisms is the reuse of their infrastructure, e.g., tools or the knowledge and experience of existing users. RT-Z is designed as a so-called 'conserving integration' and reconciles both the desire to reuse existing infrastructure and the coherence between different parts of the specification of an embedded system.

The appropriateness of RT-Z—compared with other integrated formalisms—for specifying real-time embedded systems results from the interaction of all these characteristics.

Overview

The thesis is organised as follows.

In Part I we introduce the context of the integrated formalism RT-Z: its envisaged application area (Chapter 1), its underlying system model and the development process into which it is incorporated (Chapter 2). The objectives that RT-Z must meet to be useful for the envisaged application area within the considered phases of the development process are stated in Chapter 3. Based on these objectives, we discuss the related work in Chapter 4. Then, to get a better overview and to have a basis for categorising our approach with respect to the related work, we propose a classification of approaches to integrating complementary formalisms in Chapter 5.

Part II deals with the basics of the integrated formalism RT-Z in an informal way. We first discuss the general principles pursued in integrating the formalisms Z and timed CSP in Chapter 6. In particular, we distinguish between an abstract and a concrete model of integration, covering the different abstraction levels to be supported during the development process. Then, in Chapter 7, we treat the notation of the basic units of an RT-Z specification,

which we call specification units. We finally introduce the structuring operators of RT-Z in Chapter 8 allowing us to hierarchically decompose the specification of a large and complex system into basic specification units.

In Part III we back up formally the notation introduced in the previous part. This includes the definition of a denotational semantics for RT-Z specification units in Chapter 9 and of the notion of refinement in Chapter 10, the latter chapter including techniques for establishing refinement between specification units.

A discussion of the integrated formalism RT-Z is the subject of Part IV. This includes a comparison of RT-Z with directly related integrated formalisms in Chapter 11 and two case studies in Chapter 12, carried out in order to illustrate and validate the concepts underlying RT-Z. In Chapter 13 we draw conclusions and discuss open items that remain as subjects for future work.

Finally, the appendices contain a detailed account of the two base formalisms of RT-Z (especially of timed CSP) and two substantial mathematical proofs concerning the consistency of our semantic definitions and the soundness of our refinement rules for RT-Z.

Chapter 1

Envisaged Application Area

The integrated formalism we are aiming at is directed towards the development of software in embedded systems. Embedded systems cover a wide area of applications, including process control, signal processing, communication and networking, etc. Following Koopman [1999, 1996] and Gupta [2002] embedded systems are characterised by:

- being heterogeneous, i.e., including software, digital, electric and mechanical components, etc;
- being time-dependent, i.e., associated with (hard) real-time constraints;
- being reactive, i.e., performing computations in response to periodic and spontaneous events controlled by the environment;
- involving data-processing tasks (e.g., digital signal processing);
- being safety-critical (e.g., process control);
- being inherently concurrent and distributed;
- being application specific, i.e., the application is known a priori before the system design begins.

Note that not necessarily all of the above characteristics need to apply to each individual embedded system.

Thus, software components in embedded systems usually

- have complex interrelations to their environment (constituted by various types of components);
- are reactive (as opposed to transformational), having several concurrent threads of control;
- involve data-processing;
- must perform their computations timely, i.e., are associated with real-time constraints;

- are distributed and must hence communicate via networks;
- are required to have a high degree of reliability;
- are not general-purpose.

The required abilities of the integrated formalism we are aiming at derive from the above characteristics of embedded systems and the software components they contain.

The high degree of reliability that is often required of embedded systems, particularly in safety-critical applications, renders the use of formal methods a worthwhile option, if not even prescribed by certification authorities or customers.

A characteristic of embedded systems that makes the use of formal methods realistic is that they are developed for specific, a priori known purposes. This is a contrast to desktop computing applications, which usually have a more general purpose. It is very hard to elicit clear requirements for such applications, which makes the use of formal methods problematic. Moreover, desktop computing applications tend to become very complex and put the main emphasis on aspects that are hardly amenable to a formalisation, e.g., the graphical user interface. Embedded systems do not have these characteristics and are hence more amenable to formality.

Of course, the diversity of the relevant aspects—reactiveness, real-time, concurrency, distribution, data processing, and so on—is challenging for a formalism that is to be employed for developing software in embedded systems.

Chapter 2

Underlying System Model and Development Process

In this chapter, we discuss the system model underlying the integrated formalism we are aiming at and the overall development process for embedded systems of which the integrated formalism is one constituent.

2.1 Development Process

We start with outlining the model of the system development process in which the application of the integrated formalism we are aiming at is embedded. This process model follows the waterfall model enhanced with the concept of iterations as presented, e.g., by Sommerville [1995, p. 9]. In this model, the development process is considered to be partitioned into a sequence of phases, each of which results in a well-defined set of documents. The result of a phase is the basis for performing the subsequent phase. In contrast to the basic waterfall model, the extended one takes into account the need of verification and validation (V&V) activities at the end of each phase, a negative result of a V&V activity giving rise to a repetition of phases already passed through.

The considered process model is depicted in Figure 2.1. Phases are depicted by rectangles, and the resulting documents are depicted by ellipses. The backward arrows represent the need to go back to a previous phase because of a failed V&V activity. The grey rectangle encompasses those parts of the overall development process that are to be covered by the envisaged integrated formalism.

The first two and the last two phases of this model apply to entire systems. In contrast, the phases in the middle are applicable only to a certain type of system components. The phases supporting the development of particular types of system components are arranged in different levels represented by the outmost rectangles. The front rectangle contains the phases for developing software components.

The aim of the system requirements analysis phase, resulting in the system requirements specification, is to analyse an existing system and its environment and to specify the require-

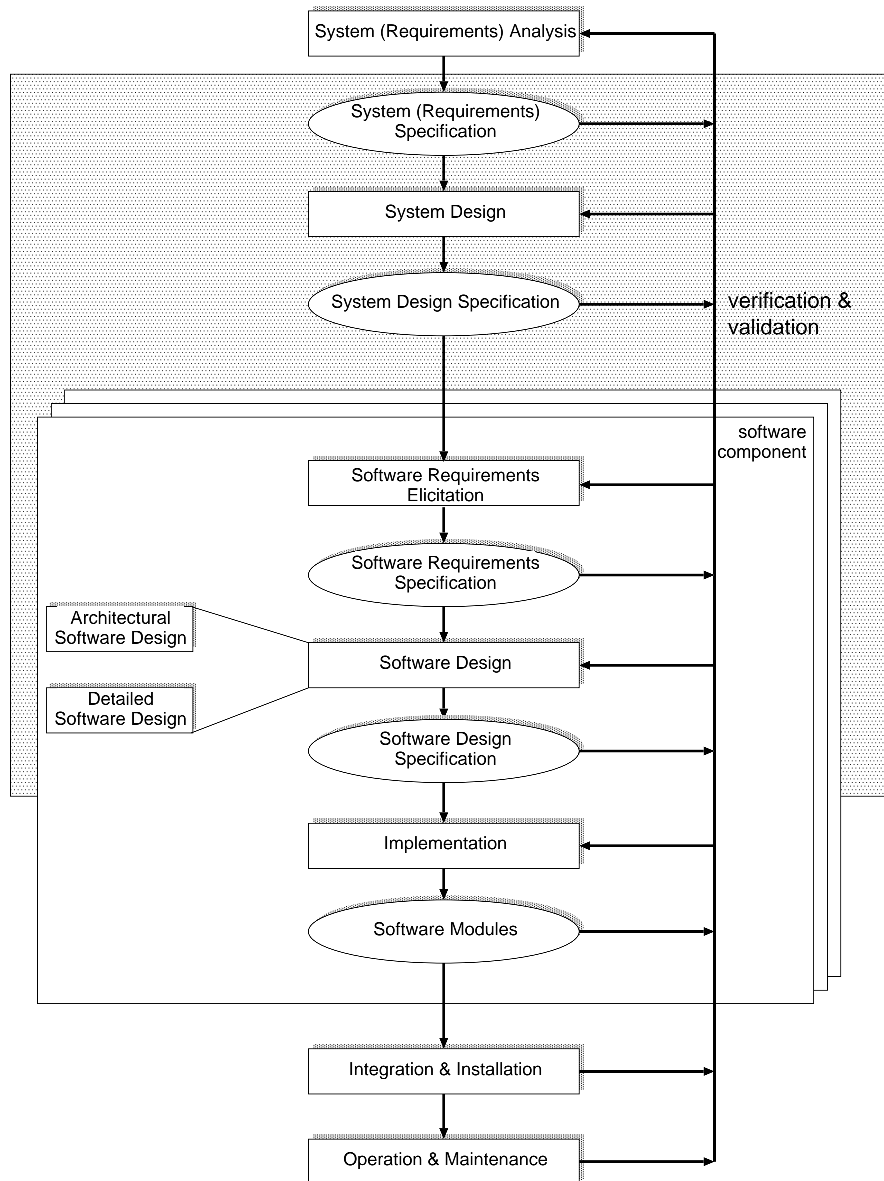


Figure 2.1: System development process.

ments (system function) and constraints the redeveloped system must meet with respect to its environment.

The system design phase, completed by the system design specification, deals with devising an implementation concept for the considered system that satisfies the requirements and constraints defined in the system requirements specification. This involves the system architecture, e.g., the decomposition of the system into certain kinds of system components. The system design specification contains specifications of different types of system components. In the following, we consider the development of software components.

The subject of the software requirements elicitation phase, resulting in the software require-

ments specification, is to define the requirements and constraints each software component, identified in the system design, must satisfy at its external interface. At this stage, no aspects of its internal implementation should be fixed.

In the software design phase, an implementation concept is worked out for each specified software component. This includes stipulating internal aspects, e.g., the internal structure, the state space, etc. The result of this phase is a software design specification for each software component.

The subsequent implementation phase transforms the design specification of each software component into an executable piece of code (module) in the chosen implementation language.

All developed system components—software, hardware, mechanical, electrical components, and so on—must be integrated eventually to form the entire system, and the integrated system must be installed in its operational environment. The operation of the developed system may entail the need to correct, to change or to extend the system, which leads to a repetition of previous phases.

2.2 System Model

We proceed with discussing the system model underlying the envisaged formalism. It determines the abstraction to be carried out when modelling an embedded system and incorporates the information needed in order to interpret specifications of the integrated formalism.[1]

We start with clarifying the notion of a system; this leads us to a rough, generic system model. The integrated formalism aimed at is to be applied in different phases of the system development process just described. Each of the phases is associated with a particular instance of this generic system model. These instances are described in the following sections.

Following Leveson [1995, pp. 136–137] we interpret a *system* as a collection of *components* acting together and interacting with each other in order to achieve some common *goal* with respect to its *environment* at the same time obeying some *constraints*.

The system *state* at any point in time is the set of relevant properties characterising the system at that time. The system environment is a collection of components that are not part of the system but whose behaviour can affect the system. The system interacts with the environment through *events* crossing the *boundary* between the system and its environment. Any set of components that is considered as a system is part of a hierarchy of systems: a system may contain subsystems and may also be part of larger systems.

We must distinguish between two types of entities in the following description of the instances of the system model.

[1] In our opinion, clarifying the system model underlying a specification language is crucial. For instance, the success of the formal specification language SCR [Heitmeyer et al., 1996] rests, among other things, on the underlying 'Four Variable Model' of Parnas and Madey [1995].

- Entities that specify characteristics of the artifact at hand that are visible to the environment.
- Entities that specify characteristics of the artifact at hand that are not visible to the environment. Such entities are used in order to enable a more convenient modelling of those characteristics that are externally visible.

As elaborated in the following sections, the state is an entity that is considered to be externally visible in the system-related phases but not to be externally visible in the software-related phases.

The above distinction leads to two different *views* on artifacts of the system development process: the *closed system view* in which an artifact is considered to be completely characterised in terms of its behaviour at the external interface; and the *open system view* in which "internal" aspects of its description are taken as an additional part of its characterisation.

2.2.1 System Requirements and Design Specifications

The aim of the system-related phases is to define the embedding of the software to be developed within its context, which may be constituted by digital, electrical, mechanical components, human operators, etc. When developing safety-critical systems, the definition of *safety constraints* is one of the most important tasks in these phases.

The entities treated in the system-related phases are (almost) of an arbitrary kind: mechanical components, e.g., valves, electrical components, e.g., fuses, digital components, e.g., electronic circuits, human operators, and—last but not least—software components. Due to this diversity there are different kinds of interaction: information flows, electrical current, mechanical pressure, etc. The system model underlying this phase must be general enough as to take this diversity of components and interactions into account.

As indicated, a system is modelled as a hierarchy of interacting components. Interaction between components (and between a system and its environment) is modelled to take place by means of *events* occurring on *channels*. This means that components interact via *synchronous* communication: for an interaction to take place, the "sender" and the "receiver" have to simultaneously participate in that interaction. The occurrence of an event may be associated with the communication of information units (between software components), the transportation of material (between mechanical components), the flow of current (between electrical components), etc. That is, we associate a particular type with each channel which restricts the set of objects that can be transported with the occurrence of events on that channel.

Each component is considered to have an *interface*, which consists of all events that serve the component to interact with its environment. Furthermore, since a component may be structured into sub-components, there may be additional events that are not part of the interface but are related to the local interaction between these sub-components. Therefore, we distinguish between external and internal events with respect to a component. An internal event is solely controlled by the component, whereas an external event is controlled by both the component and its environment.

Each component is embedded within a context. It need not achieve its goal under all possible circumstances, but it may require its environment to satisfy some conditions called *environ-*

mental assumptions: if these assumptions are violated by the environment, the component is neither required to achieve its goal properly nor to obey its constraints.

The state of a component at a particular instant of time is constituted by the set of relevant, physically available properties characterising the component at that time. In the system-related phases, as already indicated, the state is considered as an externally visible entity. That is, the definition of a state with a certain structure and certain properties in a system specification forces any conforming implementation to have a counterpart with that structure and that properties. There are two reasons why the state cannot be considered as an internal aspect of a system specification.

Firstly, safety is a property that is of relevance in the area of embedded systems. Safety, according to Leveson [1995], is a property of a *system* whose definition is directly related to the *state* of that system (and its components). Therefore, the meaning associated with a system specification must incorporate the state, because the classification of an implementation as safe or unsafe can be carried out only on this basis.

Secondly, as discussed in detail in Section 9.1 by means of an example, there are relevant types of requirements in the context of system specifications that cannot be handled properly if not including the state evolution in the meaning associated with a system specification.

The state of a component is transformed in reaction to interactions occurring at the external interface. State transitions need not be defined deterministically, i.e., the transition from a pre to a post state in reaction to an external event need not be unique. State transitions may depend on inputs and they may produce outputs. State transitions are instantaneous; the continuous change of the state during a time interval is not expressible within the chosen system model: we do not aim at describing hybrid behaviour.

The decomposition of a system specification into components renders physical structures that are already fixed by the existing development context. Therefore, the structure of a system specification fixes the structure of conforming implementations.

2.2.2 Software Requirements Specifications

The artifacts referred to in software requirements specifications are software components. These software components interact by communicating information units. The facts that we deal with software and that we do not consider whole systems but only *components* entail some consequences on the underlying system model for this phase.

In the software-related phases, as already indicated, the state defined in a specification is considered as an internal aspect of the specification that does not force conforming implementations to have a counterpart. The meaning associated with a software specification, therefore, does not incorporate the evolution of the state. The reason for this different interpretation of the state in the system-related and the software-related phases is twofold.

Firstly, safety is a property that does not refer to software components: a software component cannot be safe or unsafe; but it can contribute to the safety of the system into which it is embedded depending on its external behaviour at the interface. So, our above reasoning about safety on the system level does not apply to the software level.

Secondly, since the invention of the information hiding principle, software components are considered as encapsulated entities that interact only via defined interfaces; the environment cannot observe the internal structure and the internal state of a software component.

Following Broy et al. [1992], the specification of a state and its operations does not necessarily constrain implementations to have an implementation of the state and all its operations. Rather, the state machine model is a tool for specifying the data-related characteristics at the external interface more succinctly and conveniently. The current state of a software component can be interpreted as representing the relevant properties of the past interaction that determine the software component's ability to interact with the environment in the present and future.

In contrast to system requirements and design specifications, the structure of a software requirements specification has only conceptual reasons and does not restrict conforming implementations.

2.2.3 Software Design Specifications

The artifacts and their interrelations that are referred to in software design specifications are essentially identical to those in a software requirements specification. Thus, the system model underlying software design specifications is almost identical to the one underlying software requirements specifications. However, the following difference exist.

In the software design phase, it is the general task to devise an "implementation concept" for the requirements specified in the software requirements specification. Like in the system design, such an implementation concept encompasses (among other things) the decomposition of software components into sub-components and the definition of their interfaces. That is, the structure of a software design specification has immediate consequences for conforming implementations.

This completes our discussion of the instances of the system model underlying RT-Z. These instances roughly stipulate how the notation of RT-Z, treated in Part II, is to be interpreted, as formally fixed in Part III.

Chapter 3

Objectives

In Chapter 1, we have discussed the envisaged application area of the integrated formalism we are aiming at: real-time embedded systems. The development process incorporating the application of the integrated formalism and the phases of this process to be supported have been discussed in Chapter 2. We now derive from these context conditions objectives to be met by the formalism.

Formality, Abstractness and Rigour: Developing safety-critical systems requires a high degree of reliability. Only a notation based on mathematical concepts and associated with a formal semantics enables the clear and precise definition of requirements, constraints and models. A mathematical basis is also the prerequisite for the ability to specify at different abstraction levels. Finally, formality is the prerequisite for reasoning about specifications by means of rigorous proof techniques.

Adequate Expressiveness: The provided language constructs need to be adequate for the considered application area. On the one hand, the formalism must be expressive enough; on the other hand, the trade-off between a high expressiveness and the ability to reason about specifications must be taken into account.

Since, as claimed in Chapter 1, embedded systems are usually reactive, are usually associated with real-time constraints and usually involve data-processing tasks, the formalism should provide language constructs to cover all these three dimensions of the behaviour of an embedded system.

Coverage of the System Development Process: As stated in Chapter 2, the formalism must be appropriate for different phases of the system development process. It is to be used to set up system requirements specifications, system design specifications, software requirements specifications and software design specifications. It must hence be able to cover all involved abstraction levels.

The formalism must, on the one hand, provide abstract language constructs, which allow us to specify requirements on the externally observable behaviour in system and software requirements specifications without fixing implementation aspects. On the other hand, it must provide concrete language constructs to allow us to specify architectural and implementation aspects in system and software design specifications.

Scalability (Modularity): The formalism must cope with large and complex systems. To this end, it must provide appropriate structuring and decomposition concepts.

The above objectives are used to assess the integrated formalism that we present in this thesis as well as the related work treated in the next chapter.

3.1 Restrictions

For pragmatic reasons we do not aim at achieving all those desirable objectives whose satisfaction is vital only for a sub-class of the considered application area. The following restrictions apply to the formalism we are aiming at.

Hybrid Behaviour: The specification of hybrid systems with discrete and continuous elements is not to be taken into account. The class of systems that can be specified is thereby restricted. However, taking into account hybrid behaviour would make the formalism disproportionately complex.

Stochastic Behaviour: Similarly, the specification of stochastic systems—systems consisting of parts whose concrete behaviours occur with a certain probability—is not to be taken into account. Also this decision causes a restriction of the considered system classes, however, in favour of an essentially simpler model.

Simulation/Executability: The ability to animate and to simulate a formal specification, respectively, is not our main concern. Of course, the simulation of a formal specification is an important means for validation. However, to focus on the ability to simulate a specification appears to be in conflict with a high level of abstraction (intelligibility): simulating a specification necessitates the employment of concrete (executable) language constructs, which contradicts the requirement for abstractness at least in the early phases.

Chapter 4

State of the Art: Related Work

Before introducing our integrated formalism RT-Z, we discuss some related approaches, which partly share our goals. The discussion in this chapter is the basis for our classification of approaches to combining/integrating formalisms proposed in the next chapter. Those approaches discussed in this chapter that are related most closely to RT-Z and that have strongly influenced its design are compared with RT-Z in Chapter 11.

In the remainder of this chapter, the related approaches are introduced in the order of closeness to RT-Z.

4.1 CSP-OZ

CSP-OZ [Fischer, 2000, 1997, Fischer and Wehrheim, 1999] is an integration of Object-Z and (untimed) CSP. According to the classification scheme discussed in the following chapter, it is a *monolithic integration*.

In CSP-OZ, each system specification is composed of class specifications. Fischer distinguishes between Object-Z classes, encapsulating only Object-Z notation, and CSP-OZ classes, encapsulating the notations of Object-Z and CSP. Objects, being instances of class specifications, interact with each other along typed CSP channels: each class has an interface part declaring a set of CSP channels each of which associated with a Z type. In CSP-OZ, each interaction at the interface of an object corresponds to the application of an Object-Z operation that is defined in the corresponding class. With each operation application, the input and output parameters of the corresponding operation are communicated along the corresponding channel. The Z type associated with a channel is thus the schema type containing the parameters of the corresponding operation schema.

Roughly speaking, each CSP-OZ class consists of three parts.

1. The interface part mentioned above.
2. The Object-Z part. It defines, in the style of Object-Z, data types, constants, a data state schema, an initialisation schema and operation schemas.[1]

[1] Since 'schemas' is established in the Z community as the plural of 'schema,' we use it instead of the grammatically correct 'schemata.'

3. The CSP part. It defines the dedicated process term *main*; this process basically defines the order in which operations are applied.

CSP-OZ partly adopts the structuring mechanisms of Object-Z: inheritance and object instantiation. Compared with Object-Z, however, object instantiation is allowed only in a restricted form: only Object-Z classes, which do not contain CSP definitions, are allowed to instantiate other Object-Z classes in their data state schema and thus to exploit the full generality of object instantiation in Object-Z (including the ability to express invariants relating the data states of several instantiated objects). Objects of CSP-OZ classes, on the other hand, can instantiate objects of CSP-OZ classes only as fully encapsulated entities; they can refer only to the interfaces of the instantiated objects. Instantiation of objects in CSP-OZ classes takes place in the CSP part: the instantiating object can dynamically create objects of CSP-OZ classes depending on its control state.

In CSP-OZ, the integration of Object-Z and CSP is achieved in two layers (phases). In the first layer, an intermediate language, CSP_Z, is defined that integrates Z and CSP. More precisely, CSP_Z is defined as an extension of Standard Z with CSP process expressions. CSP_Z expressions constitute the CSP part of CSP-OZ classes. In the second layer, the structure of CSP-OZ classes is defined in terms of the Object-Z notation and the intermediate CSP_Z language. Most important, the meaning of CSP-OZ classes is defined by means of translation rules into the CSP_Z syntax.

CSP_Z is an extension of the Z notation with the process term syntax of CSP. It is a very close integration of the base notations and the basis for our classification of CSP-OZ as a monolithic integration. Roughly speaking, the Z grammar is extended with additional production rules for CSP process expressions: each CSP expression is a Z expression by definition. This leads to a very convenient and concise style of specification. For example, the interleaving of a collection of processes indexed by members of a set expression can be expressed as

$$|||\, i : 1 \,..\, N \bullet c!i \rightarrow Stop$$

where $i : 1 \,..\, N$ is a Z expression and $c!i \rightarrow Stop$ is a CSP process expression.

Central to the proposed integration of the Z and CSP notation is the introduction of parametrised process definitions, i.e., the ability to parametrise CSP process definitions by arbitrary Z expressions, e.g.,

$$P(i : \mathbb{N}) \stackrel{c}{=} c!i \rightarrow Stop.$$

To define the meaning of such parametrised process definitions, Fischer provided syntactic transformation rules mapping parametrised process definitions to Z syntax. The above definition, e.g., would be mapped to the axiomatic definition

$$\begin{array}{|l} P : \mathbb{N} \rightarrow PROCESS \\ \hline \forall i : \mathbb{N} \bullet P(i) = c!i \rightarrow Stop \end{array}$$

where *PROCESS* is a given set assumed to be present in each CSP_Z specification and to represent all possible process expressions.

The meaning of CSP_Z process terms is defined by a so-called hybrid semantics, which in fact is a denotational semantics, but associates with each process term a denotation that represents a labelled transition system; this is the operational nature of the semantics justifying the term 'hybrid.' A main principle of defining the hybrid semantics, written $[\![_]\!]^{\mathcal{O}}$, is to interpret each process term in the context of a Z model.[2] Accordingly, the semantic function of CSP_Z is a curried function with the signature

$$[\![_]\!]^{\mathcal{O}} : PROCESS \rightarrow Model \rightarrow LTS$$

where *LTS* is intended to represent the set of all labelled transition systems. The function $[\![_]\!]^{\mathcal{O}}$ is induced by a set of inference rules (several for each CSP operator).

In the second layer, as already indicated, the meaning of CSP-OZ classes is defined in terms of transformation rules, mapping hierarchies of CSP-OZ classes to single classes and single CSP-OZ classes to equivalent CSP_Z terms. This transformational approach to defining the semantics of CSP-OZ classes follows the approach of Object-Z, which maps Object-Z syntax to (plain) Z syntax.

The main principle of defining the meaning of a CSP-OZ class, constituted by an Object-Z and a CSP part, follows an idea of Smith [1997]: the Object-Z part of a CSP-OZ class is interpreted as a CSP process and is subsequently composed in parallel with the CSP part, where both parts must synchronise on the operations of the Object-Z part. Accordingly, the transformation rules yield two CSP_Z process terms *procC* and *procZ* for each CSP-OZ class, representing the CSP and Object-Z part, respectively, and the process term *proc*, representing the meaning of the whole CSP-OZ class.

$$proc(C) = (procC(C) \ldots \parallel procZ(C) \ldots) \setminus \ldots$$

A detailed discussion of the strengths and drawbacks of CSP-OZ can be found in Chapter 11. Only there, after having introduced RT-Z, it is meaningful to further elaborate the description of CSP-OZ, including the discussion of which of the objectives stated in the previous chapter are met.

Extensions

Hoenicke and Olderog [2002] have proposed an extension of CSP-OZ, called CSP-OZ-DC, which uses the Duration Calculus [Chaochen et al., 1991] for specifying real-time constraints on the behaviour of objects. In the Duration Calculus, real-time constraints are specified in terms of integrals over continuous functions. Thus, the time models underlying CSP-OZ-DC and RT-Z, which is based on timed CSP, are diametrical.

Sherif et al. [2001] have adapted the predecessor of CSP-OZ, CSP-Z, by substituting timed CSP for CSP. This allows them to express real-time constraints in addition to the capabilities of CSP-Z. The authors applied the resulting language, called Timed-CSP-Z, to the specification and validation of the on-board computer of a Brazilian research satellite.

Because no semantics has been defined for Timed-CSP-Z, it is not backed up formally. The authors defined a translation from Timed-CSP-Z specifications to high-level Petri Nets,

[2] The term 'Z model,' which is Z Standard terminology, is explained in Appendix A.1.

which can be checked by existing analysis tools for particular types of properties. The translation process, however, must be carried out manually. Further, since the meaning of Timed-CSP-Z is not defined formally, there is no base against which the correctness of this translation process could be checked.

4.2 TCOZ

Timed Communicating Object-Z (TCOZ) [Mahony and Dong, 2000, 1998a, 1999b] is an integration of Object-Z and timed CSP. According to the classification scheme discussed in the following chapter, it is a *monolithic integration*.

Each system specification in TCOZ is composed of class specifications. TCOZ classes incorporate the notations of Object-Z and timed CSP. Objects, being instances of class specifications, interact with each other along (untyped) CSP channels, which constitute the interface of classes. The most important structuring operators are inheritance and object instantiation.

Each object is associated with a data state specified by an Object-Z state schema in the corresponding class, and it transforms the data state by performing operations specified by Object-Z operation schemas. The dynamic behaviour of an object, i.e., its interaction with the environment and the internal control of its operations, is specified by the dedicated timed CSP process term *MAIN* in the corresponding class.[3]

The main principles of integrating the notations of Object-Z and timed CSP to obtain the syntax of TCOZ classes is outlined in the following.

- Object-Z operation schemas are interpreted by TCOZ as terminating timed CSP processes. For each operation schema *Op* of a TCOZ class, the timed CSP process *MAIN* of this class can refer to the process *Op* in order to apply this operation to the current data state.

- In each operation schema, input and output parameters may be associated with channels being part of the interface of the class. Applying an operation means to read the specified input parameters from the corresponding channels, to transform the object state according to the relationships specified in the operation schema and to subsequently write the computed output parameters to the corresponding channels. The exact order of reading input parameters and writing output parameters, the exact timing aspects of these parameter interactions and the time instant at which the state transition actually takes place is not fixed by the schema definition.

- The state schema of a class specified in Object-Z is extended by an interface definition, i.e., a list of (untyped) CSP channels connecting the class objects to other objects. The operations of the class read and write their parameters from and to these channels, respectively.

[3] TCOZ distinguishes active and passive objects [Dong and Mahony, 1998]. For active objects, the dynamic behaviour is explicitly defined by the process *MAIN*. In contrast, the behaviour of passive objects is controlled by instantiating objects. Objects instantiate other objects by declaring a state variable of the respective class type in their state schema. The data state of the instantiated object is part of the data state of the instantiating object. The instantiating object may apply operations of the instantiated object and thereby transform its data state.

- The specification of the dynamic behaviour in a TCOZ class mainly follows the syntax of timed CSP, however, essential extensions have been defined to enable the reference to the data state.
  - So-called *state guards* serve to make the dynamic behaviour, more specifically the flow of control, dependent on the current data state of an object. A state guard is an arbitrary schema expression and may refer to the data state that is present when the process guarded by the state guard starts.
  - Value expressions within timed CSP processes may refer to arbitrary components of the data state in order to denote values to be communicated via external channels. These state variables are evaluated with respect to the data state present at the time of communication.

In summary, the notations of Object-Z and timed CSP are integrated very closely.

Considering the semantic model of TCOZ, the monolithic nature of the integration becomes even more evident. Both timed CSP and Object-Z are equipped with a formal semantics, see [Schneider, 1999a] and [Smith, 2000]. By merging the notations of Object-Z and timed CSP in TCOZ, however, there are no identifiable parts of a TCOZ class solely formulated by one of both notations. Therefore, it is not feasible to reuse the semantic definitions of the base formalisms, because there are no parts within a TCOZ class to which the semantic functions of Object-Z and timed CSP could be applied to somehow obtain the semantic function of TCOZ. Consequently, the semantics of TCOZ is defined—more or less—from scratch.

The semantic model underlying TCOZ is the so-called timed, infinite states model [Mahony and Dong, 1999a, 1997]. Each object of a TCOZ class is associated with a set of observations each of which consisting of a state history, an infinite timed trace and a timed refusal. The definition of the semantics of TCOZ is rather abstract, using Z as the meta language. The meaning of the most important structuring facilities of Object-Z—inheritance, object instantiation, containment—is not taken into account in the current version of the TCOZ semantics. This is problematic, as the Object-Z extensions are the only operators to structure a TCOZ specification, and they are extensively used (cf. [Mahony and Dong, 1998b]). Inheritance and object instantiation are undoubtedly utmost powerful notational means, but their use is problematic within a formal approach if their meaning is not defined precisely.

The strengths and drawbacks of TCOZ, including the discussion of which objectives stated in the previous chapter are met, are discussed in detail in Chapter 11 after having introduced RT-Z.

4.3 Object-Z / CSP

Smith and Derrick [Smith and Derrick, 2001, Derrick and Smith, 2000] have presented an integration of Object-Z and CSP. According to the classification scheme discussed in the following chapter, it is a *conserving integration*. The approach they proposed on the basis of their integrated formalism supports—in addition to specifying concurrent and distributed systems—aspects of refinement and verification.

The main idea underlying the integration is the identification of corresponding concepts of the specification of object-oriented and concurrent systems—classes and operations on the one and processes and events on the other hand. This identification is underpinned semantically by giving Object-Z classes a semantics in the failures–divergences model of CSP; it allows Object-Z classes to be referred to as CSP processes and hence to be composed by CSP operators.

A system specification in the considered approach consists of two parts. The Object-Z part comprises a number of Object-Z classes, each defining the behaviour of a single system component. The CSP part fixes the configuration of the defined system components and their interaction with the help of CSP operators, e.g., parallel composition. That is, the approach chosen by the authors to integrate Object-Z and CSP is absolutely different from the approach taken in the design of CSP-OZ and TCOZ: in the current approach, Object-Z and CSP cover different layers of a system specification. The first layer deals with single components by solely using Object-Z, and the second layer deals with composing the components defined in the first layer by solely using CSP. Thus, the notations of the base formalisms are *strictly* separated from each other. This separation between different layers is a contrast to the approach taken by the other integrations of Object-Z/Z and timed/untimed CSP, in which the respective notations are integrated within the component level.

Since Object-Z classes are associated with a CSP semantics, refinement between integrated specifications is simply defined in terms of CSP refinement, i.e., failures–divergences refinement. The authors have adapted the work of Josephs [1988], which has provided refinement relations for state-based systems that are sound and complete with respect to CSP refinement, to the Object-Z setting. This enables them to check CSP refinement relations between Object-Z classes directly using state-based techniques without the need to compute the failures–divergences semantics of these classes.

The authors have also outlined an approach to verifying integrated specifications. Because of the strict separation of the base formalisms, the existing verification techniques of Object-Z and CSP can be combined to check properties of an overall system specification: first CSP's inference rules are used in order to reduce properties that the overall system must satisfy to properties that its components must meet, and then Object-Z's verification techniques are applied in order to show that the components really meet these properties.

The major benefit of this approach is the absolutely strict separation of the base formalisms, exploiting the maximal potential to reuse their infrastructure. The choice of CSP to define the interactions between system components allows the convenient definition of architectural aspects.

A disadvantage of this approach, on the other hand, is that CSP is not used in order to define the dynamic behaviour of single system components (e.g., the temporal order of operation applications). It is still Object-Z that is used for this purpose. From our point of view, CSP is more appropriate than Object-Z to specify behavioural aspects.

Furthermore, Smith [2002] has presented an integration of Real-Time Object-Z [Smith and Hayes, 1999] and CSP. The principles underlying this integration are identical to those explained above. However, using Real-Time Object-Z for defining the behaviour of system

components, the specifier is also able to express real-time constraints. Real-Time Object-Z is based on the timed trace notation of Fidge et al. [1998], which is an interval-based set-theoretic notation expressing real-time constraints in terms of continuous functions. Thus, the time models underlying Real-Time Object-Z/CSP and RT-Z are diametrical.

4.4 LOTOS

LOTOS[4] [Bolognesi et al., 1995, Bolognesi and Brinksma, 1987] is a formal description technique developed by the International Standardization Institute (ISO) in the context of the specification of the Open Systems Interconnection (OSI) architecture. It integrates two complementary languages:

- a process algebra, which is mainly based on CCS [Milner, 1989], for specifying behavioural aspects and
- the algebraic specification language ACT ONE [Ehrig and Mahr, 1985] for specifying abstract data types, i.e., data structures and value expressions.

According to the classification scheme discussed in the following chapter, it is a *conserving integration*.

The system model underlying LOTOS regards a distributed, concurrent system as a *process*, composed of a collection of interacting sub-processes. A process is an entity able to control internal (unobservable) actions and to interact with its environment (constituted by a set of other processes). Complex interactions between processes are built up of elementary, atomic units of synchronisations called *actions*. The occurrence of an action implies process synchronisation, i.e., all processes that are connected by an action have to participate in its occurrence simultaneously. The occurrence of an action may be associated with the transmission of data values. Actions are thought of as occurring at a synchronisation point called *gate*. The interface of a process is constituted by the set of gates via which it interacts with its environment.

One distinguishes between full LOTOS, providing the full range of syntactic constructs, and basic LOTOS, providing only the process algebra language.

A process definition specifies the externally observable behaviour of a system component in terms of sequences of observable actions at its gates. A process definition is parametrised by a list of formal gates; these constitute the interface of the process via which the process interacts with the environment (through synchronisation and value transmissions). The observable behaviour of a process is defined by a so-called *behaviour expression*.

In the data type part of a process definition, a set of abstract data types are introduced by using the algebraic specification language ACT ONE. Basic LOTOS is extended to full LOTOS by language constructs referring to the elements of the abstract data types, i.e., constants, functions and carrier sets. In full LOTOS, the occurrence of an action may be associated with the communication of an arbitrarily complex value.

[4] LOTOS is a shorthand for Language of Temporal Ordering Specification.

The most important additional language constructs of full LOTOS are selection predicates and guarded behaviour expressions.

Let us first consider a selection predicate in the example action denotation $g\,?x : T1\,!y\,[x < y]$. Each occurrence of an action at gate g involves the communication of two values: the first value must be of type $T1$ and is bound to the variable x during synchronisation, and the second value is uniquely determined by the expression y. The *selection predicate* $x < y$ defines a further constraint on the value bound to the variable x during synchronisation.

As a second extension, each behaviour expression, say BE, may be guarded by a predicate, say G, written $G \rightarrow BE$. If the guard G evaluates to *true*, the behaviour of the guarded expression is BE; otherwise the whole expression is equivalent to deadlock. The guard may refer to any process parameter and to any variable bound by a prior communication of the process; this allows the dynamic behaviour to depend on the current (data) state of the process.

The meaning of behaviour expressions is defined by an operational semantics. To relate different descriptions of a system, e.g., its requirements specification and its design specification, several *behavioural equivalence relations* are defined between behaviour expressions based on the operational semantics. The core of the process algebra LOTOS is constituted by a set of algebraic laws stating equivalences between behaviour expressions.

To conclude, LOTOS is an expressive language, closely integrating notations for defining behaviour and data types. The module concept of LOTOS is elaborate allowing it to tackle large and complex systems. Moreover, LOTOS has a completely formal foundation. Its proof rules support a rigorous development of a specification towards an implementation.

However, LOTOS has also some drawbacks. First, it does not aim to specify real-time constraints. Second, it is based on an algebraic specification language. While an algebraic specification language is perfectly suited to specifying requirements, it is difficult to use it in order to fix design decisions, in particular architectural aspects. Finally, its process algebra is based on an operational semantics. This implies that requirements must be expressed in terms of behaviour expressions (rather than predicates).[5] In our opinion, specifying requirements in terms of behaviour expressions is not as adequate as specifying requirements in terms of predicates.

4.5 RAISE Specification Language

RSL is part of the formal method RAISE [Haxthausen and George, 1993], which is an acronym for *Rigorous Approach to Industrial Software Engineering*. RSL is a formal specification language and has resulted from an integration of concepts from VDM [Jones, 1986], CSP [Hoare, 1985], algebraic specification languages and ML [Paulson, 1991]. It is a wide-spectrum language in that it enables the user to specify a variety of aspects of the system behaviour (e.g., dynamic behaviour, functional I/O behaviour) and in that it supports different abstraction levels. For the abstract levels, axiomatic (algebraic) constructs are provided; for the more concrete levels, model-oriented constructs are provided.

[5] Later in this thesis, we will see that a process algebra that is based on a denotational semantics (such as CSP) provides a predicate language in addition to the language of process terms.

The notation is equipped with methodological support, where the overall development process mainly follows the stepwise-refinement paradigm. The meaning of RSL specifications is formally defined by a denotational semantics. Based on this semantics, proof rules have been developed allowing one to derive properties from RSL specifications and to establish refinement between two successive specifications in the development process. The entire RAISE method is also supported by tools, e.g., editors, type-checkers and provers.

In RSL, a system is modelled as a network of interacting system components. Each system component is specified by a module. Interaction between different modules takes place via typed communication channels. The language provides an elaborate module concept, comprising aggregation, indexed aggregation, parametrisation of modules, stepwise module extension and hiding and renaming of module components. Inside a module, however, there are no elaborate structuring facilities comparable, e.g., with the Z schema calculus.

The language has been designed to encompass the main language constructs of VDM, CSP and algebraic specification languages; this has been achieved by a very close integration of these base notations. Within an RSL module, these different language constructs (e.g., CSP processes, VDM operations) are not separated syntactically. Thus they needed to be lifted on a uniform semantic interpretation. CSP processes, e.g., are interpreted as VDM functions with side effects on external channels and on the global state.

As already indicated, RSL is an utmost expressive language covering various aspects of the behaviour of a system and several phases of the development process. From our point of view, however, RSL is far too expressive, integrating features of too many different formalisms. This leads to a low degree of intelligibility.

Further, the monolithic nature of the integration makes it almost impossible to reuse the infrastructure of the base formalisms, e.g., existing tools or the knowledge and experience of their existing users.

4.6 Temporal Logic of Actions

Lamport [1994] has developed the formal specification language TLA (Temporal Logic of Actions). It is a variant of temporal logic, as invented by Manna and Pnueli [1992], instantiated by a particular system model. TLA^+ [Lamport, 2000] improves TLA by introducing a module concept, i.e., it is designed to cope with complexity. TLA^+ is suited for specifying and reasoning about concurrent systems.

A system specification in TLA^+ is composed of a collection of modules. Each module specifies a system component in terms of the behaviours it may perform.[6] Usually, a TLA^+ module has the following structure. It first introduces a collection of variables, which constitute the state of the component under consideration. TLA^+ is an untyped language, so in principle variables can assume arbitrary values. State transitions, i.e., pairs of adjacent states in a behaviour, are then defined by actions, which are predicates on pairs of states. Finally, safety and liveness properties of the admissible behaviours are defined by using temporal logic. Safety properties deal with the initial states in which the component may start and

[6] A behaviour is an infinite sequence of states, and a state is an assignment of values to variables.

with the state transitions it is allowed to perform. Safety properties are usually defined by a temporal formula of the form

$$Init \wedge \Box[Next]$$

where *Init* characterises the set of initial states and *Next*, the so-called next-state action, defines the relation between adjacent states in a correct behaviour. Thus, safety properties define what the component must not do. Liveness properties, in the form of weak and strong fairness conditions, on the other hand, define which actions are assumed to be (eventually) performed under which conditions.

TLA$^+$ adds the module concept to TLA. Modules can extend other modules by importing their definitions. This allows general-purpose definitions, e.g., the natural numbers or sequences, to be stored in module libraries. The other module operator is instantiation, which allows a module to incorporate arbitrary collections of instances of other modules. The instantiation operator supports the hierarchical decomposition of system specifications.

Modules in TLA$^+$ are basically containers for collections of definitions. They are used to structure the definitions of large specifications; but they cannot be used to fix the structure of a system, e.g., the configuration of its components and their interactions. This is the case because the interactions between the instantiated modules are defined only implicitly: the instantiating module connects the instantiated modules by defining actions on the overall state—including the states of the instantiated modules—thereby composing sub-behaviours of the instantiated modules to complete behaviours.

A major strength of TLA$^+$ is its module concept, which allows the specifier to structure large collections of definitions in an intelligible way. Another benefit of using TLA$^+$ is its tool support. Among others, there is a syntax checker, a model checker and a simulator for a subclass of "executable" TLA$^+$ specifications. For TLA, there is also an implementation in the generic theorem prover Isabelle [Kalvala, 1995].

However, TLA$^+$ has also some shortcomings.

Most important, it is difficult (if not impossible) to fix architectural aspects of a design, because the configuration of components and their interactions are not made explicit in a system specification and could hence be changed arbitrarily in refining specifications and hence in the ultimate implementation.

Further, TLA$^+$ is not a typed language, which, from our point of view, is a major drawback. Typed languages, e.g., the Z notation, allow specifications to be type-checked, which helps detect important types of errors, including conceptual ones.

Last, but not least, the notation provided by TLA$^+$ for defining the state does not provide any structuring concepts comparable with the Z notation (schema calculus). The state is constituted by a plain (unstructured) collection of variables. The same applies to the specification of actions, for which only the operators of predicate logic can be used.

4.7 Others

In the remainder of this chapter, we discuss some approaches that are related to RT-Z only in a wider context but that are nevertheless useful in discussing the design principles under-

lying RT-Z.

The following viewpoint-oriented approaches are based on the general ViewPoints framework developed by Finkelstein et al. [1994, 1992], whose discussion is beyond the scope of this thesis.

Viewpoints in a Formal Setting

Boiten et al. [Boiten et al., 2000, 1996, Bowman et al., 1999a, 1996] have proposed a system development framework in the context of the Open Distributed Processing (ODP) [ISO, 1998] standardisation, which is based on the use of multiple viewpoints.

According to the authors, the complete specification of any non-trivial distributed system involves a large amount of information. To capture all aspects of such a system in a single model is generally infeasible. As a consequence, their framework aims to establish a configuration of models each aimed at capturing a particular facet of the system, satisfying the requirements which are the concern of a particular stakeholder of the development process. More specifically, the framework achieves this separation of concerns by the identification of a set of so-called *viewpoints*. The general idea is that multiple partial specifications (viewpoints) of a system are developed. Each partial specification represents a different perspective on the system under development at a particular level of abstraction. In particular, different viewpoints may be specified using different formal description techniques (FDT).

An immediate consequence of adopting a multiple viewpoint approach is that descriptions of the same or related entities can appear in different viewpoints. Thus, different viewpoints can impose contradictory requirements on the system under development and the consistency of viewpoint specifications needs to be checked. Thus, providing techniques to check consistency is one of the major research topics in the context of viewpoint modelling. A necessary prerequisite for checking consistency is to define the relationships between viewpoints that overlap. This could be achieved by using the same name for common aspects in different viewpoints. In general, however, more complicated mechanisms for relating common aspects of viewpoints are needed. They are called *correspondences* in this framework.

The traditional approach to consistency checking is to translate all viewpoints to a common, underlying semantic model (e.g., first-order predicate logic): if all the translations have a common model, i.e., their conjunction is satisfiable, then the viewpoints are consistent. However, this approach has two shortcomings. First, a detected inconsistency is in terms of the common semantic model, and it is in general impossible to trace it back to the involved viewpoints. Second, the translation is accompanied by a loss of syntactic structure, impeding an incremental approach. For these reasons, the authors pursued an alternative approach to consistency checking: consistency of viewpoint specifications is defined in terms of the existence of a common *development*. The term development comprises several mechanisms for evolving specifications towards implementations, e.g., the stepwise refinement within a single formal description technique and the translation between different formal description techniques. Since all viewpoint specifications must eventually be realised by a single implementation, there must be a way to combine specifications from different viewpoints during development; this combination is called a *unification*. To unify specifications of different formal description techniques, a *translation* mechanism is needed to transform a specification between the respective languages.

The definition of consistency in the considered framework is as follows.

> A set of viewpoint specifications are consistent if there exists a specification that is a development of each of the viewpoint specifications with respect to the identified development relations and the correspondences between viewpoints. This common development is called a unification. [Boiten et al., 2000]

Besides the definition of consistency, the framework incorporates a method for constructively establishing the consistency of a set of viewpoint specifications [Bowman et al., 1999b]. This involves algorithms that build unifications from pairs of viewpoint specifications. An important notion in this context is that of a *least developed unification*. This is a unification such that all other unifications are developments of it. Thus, it is the least developed of the set of possible unifications according to the respective development relations. Using least developed unifications as intermediate stages, global consistency of a set of viewpoints can be established by a series of binary checks.

The framework also distinguishes between *intra-language consistency* and *inter-language consistency*. Intra-language consistency is a relation between specifications expressed in the same language, whereas inter-language consistency crosses a language boundary. As mentioned, to establish inter-language consistency, translation mechanisms are needed to transform a specification in one language into another language.

A particular instance of this framework is described in [Boiten et al., 2000] by means of a case study. This instance uses the formal description techniques LOTOS and Object-Z for the different viewpoints. Appropriate translation techniques from LOTOS to Object-Z are investigated in [Derrick et al., 1999].

To conclude, the current approach is not a particular integrated formalism but rather a general framework containing techniques and guidelines to use the particular formalisms best suited to a particular development task in a combined way.

Weber [1997] has presented a viewpoint-oriented approach, which aims at the combination of formal and semi-formal specification techniques for the development of embedded control systems. He proposed to decompose the specification of an embedded control system into three viewpoints: an architectural model, a behavioural model and a functional model. Although his approach is general in that it does not explicitly fix the particular specification languages to be used for the three models, Weber described his approach with respect to a particular instance, which uses Z for the functional model, Statecharts for the behavioural model and class and instance diagrams for the architectural model.

Weber has put emphasis on checking the consistency between the three models. To this end, he defined a mapping that relates elements in different models that are intended to be descriptions of a single entity and necessary conditions that must be met for the different descriptions to be consistent. Of course, this mapping and the consistency conditions are specific to the particular specification languages chosen for the three models.

Viewpoints in Z

Ainsworth et al. [1994] have investigated techniques to organise large and complex Z specifications. Using their approach, any specification is divided into a set of *viewpoints*; each

viewpoint is a partial specification that describes only certain aspects of the system under development and is constituted by a "state-oriented" Z specification. The process of integrating all viewpoints of a system to a single specification and of checking their mutual consistency is called *amalgamation* in this approach.

Since all viewpoints of a system specification use the same language (Z notation), the approach is homogeneous and avoids many problems that arise when multiple notations must be reconciled. On the other hand, this approach does not aim to improve the expressive power of Z; it restricts itself to manage the complexity of large specifications and thus to enhance the structuring facilities of plain Z.

The process of amalgamating viewpoints, as illustrated in [Ainsworth et al., 1994] by means of an example, is mostly driven by the intuition of the human specifier. The result of the amalgamation of a pair of viewpoints is a combined viewpoint (Z specification) that must, roughly speaking, be a refinement of both original viewpoints. Corresponding specification elements of both original viewpoints and the resulting viewpoint are formally related with the help of retrieve relations, which are the basis to check for refinement.

Conjoining Specifications

Zave and Jackson [1993] have described a general framework for multi-paradigm specifications. According to their terminology, a *multi-paradigm specification* is a finite set of *partial specifications*, each covered by the formalism best suited to the particular purpose.

Central to their approach is the existence of a common, underlying semantic model: single-sorted first-order predicate logic. Each formalism to be used in a partial specification must be equipped with a semantic mapping from its expressions to equivalent assertions in first-order predicate logic. These mappings might be partial, i.e., in some cases it suffices to transform the specific parts of a formalism that are used. Composition of partial specifications, then, corresponds to the conjunction of the assertions that are the transformations of the partial specifications.

An important notion in this framework is that of the *vocabulary* of a partial specification, which is the set of predicates contained in its equivalent assertion. Different partial specifications interact with each other via the intersection of their vocabularies: the predicates that both partial specifications define and use, respectively. A set of partial specifications are *consistent* if there exists a common model of their associated assertions, i.e., if the conjuction of their associated assertions is satisfiable.

The approach described in [Zave and Jackson, 1993] is a framework rather than a concrete specification language, because no particular *decomposition style* is fixed, encompassing

- the decomposition of a multi-paradigm specification into a fixed set of partial specifications,
- the set of formalisms to be used within these partial specifications and
- how partial specifications interact.

Moreover, the framework does not fix a particular system model: it is not fixed how a specified system interacts with its environment and which assumptions about the environment

are made. It describes how to organise/structure specifications using different formalisms, how to relate aspects covered in different partial specifications, how to obtain the meaning of a multi-paradigm specification and how to check for consistency.

Major shortcomings of the framework are that most formalisms can be translated at most partially into predicate logic[7] and that the interpretation of a multi-paradigm specification as the conjunction of the assertions associated with its partial specifications is not intuitive and hence very difficult to understand.

An instance of this framework is presented in [Zave and Jackson, 1996]. The instantiation is achieved by fixing a particular decomposition style and system model. The formalisms used by this instance are primarily Z and finite state machines.

Composing Specifications

Abadi and Lamport [1993] have analysed sufficient conditions that specifications of concurrent components must satisfy in order to be successfully composed to a system specification with given properties.

They have investigated these conditions in a general semantic model, namely the transition-axiom method (TAM), which is an extension to temporal logic. That is, their composition principles are independent of particular specification languages. As long as the semantics of a specification formalism can be mapped to the TAM model, it can be used to specify components of the system under development. Thus, the TAM model can be regarded as a common, underlying semantic model in which the meaning of system components, possibly specified by several formalisms, can be expressed. The meaning of the composition of a set of component specifications is defined to be the conjunction of the meaning associated with the component specifications.

Note that in this approach the decomposition of a system into the parts that are covered by distinct formalisms is absolutely different from the decomposition in the other approaches discussed in this chapter. In the current approach, different "partial specifications" describe different system components, so there is no overlap between different partial specifications. In the other approaches, in contrast, different partial specifications describe the same system from distinct viewpoints, where the covered aspects usually overlap. Thus, the approaches are based on orthogonal principles for decomposing system specifications.

[7] Große-Rhode [2001] suggests transformation systems as a semantic domain into which all kinds of specification languages can be mapped semantically. His approach is more general because transformation systems, which extend labelled transition systems, are more powerful than first-order predicate logic.

Chapter 5

Classification

Considering the plethora of approaches discussed in the previous chapter and the international conferences dedicated to this subject, e.g., [Grieskamp et al., 2000, Ehrig and Große-Rhode, 2002], we conclude that integrating existing formalisms has become a research field of growing interest. To get a better overview of the field and to have a basis for categorising our approach into the existing work, we propose a classification of approaches to combining/integrating existing formalisms. On the top level of this classification, we distinguish between two classes.

In *component-oriented approaches*, the structure of composing the deployed base formalisms corresponds to the component structure of the specified system, i.e., each component of the system under development is specified by a particular base formalism. The only instance of this class described in the previous chapter is the approach of Abadi and Lamport [1993]. In *viewpoint-oriented approaches*, by contrast, different base formalisms are used in order to cover different aspects of the system under development. That is, the composition of the used base formalisms is orthogonal to the component structure of the specified system: each system component is specified using several base formalisms. All approaches discussed in the previous chapter with the exception of that of Abadi and Lamport belong to this class.

Let us further refine the class of viewpoint-oriented approaches by proposing three sub-classes. All three sub-classes can be thought of as being located within a spectrum of the degree of integrating the respective base formalisms, see Figure 5.1.

Viewpoint-oriented Combinations. One end of the spectrum, with a low degree of integration, is occupied by the class of viewpoint-oriented combinations. They are frameworks for combining multiple languages rather than particular combined languages. They provide an infrastructure containing general techniques and guidelines

- to decompose complex specifications into viewpoints (partial specifications),
- to relate common aspects of different viewpoints,
- to map all viewpoints to a common, underlying semantic model and
- to check a multi-viewpoint specification for consistency.

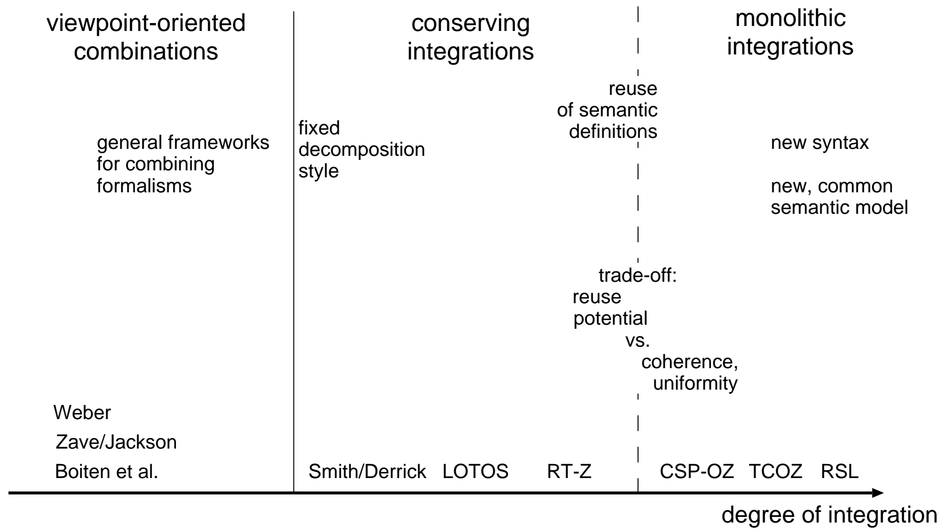


Figure 5.1: Viewpoint-oriented approaches to integrating formalisms.

In other words, viewpoint-oriented combinations do not fix a particular *decomposition style*. We use the term decomposition style of a multi-language specification to refer to the information

- of which concrete constituents it is composed,
- which base formalisms are associated with its constituents and
- the concrete correspondences via which its constituents are related.

Particular instances of such frameworks, however, define specific decomposition styles on the basis of the general guidelines prescribed by the framework.

Instances of this class are the approaches proposed by Zave and Jackson [1993], Weber [1997], Boiten et al. [1996] and Große-Rhode [2001].

The main advantage related to viewpoint-oriented combinations is the possibility to use the base formalisms within viewpoints without the need of adaptation. This allows specifiers to make use of the infrastructure available for the respective base formalisms, e.g., specification and verification tools, and of their gained knowledge and experience. Another advantage is the flexibility to use the particular base formalisms best suited for a particular specification task. A further aspect of flexibility is the high independence of a viewpoint-oriented combination from potential changes of the used base formalisms.

The most serious disadvantage associated with viewpoint-oriented combinations, on the other hand, is that a multi-paradigm specification is not really a single document but rather an independent coexistence of loosely coupled partial specifications, i.e., there is a low coherence between the combined base formalisms. That is, it is difficult to understand the interaction between different partial specifications and to obtain the overall picture drawn by all partial specifications together. Another major problem related to viewpoint-oriented

combinations is due to the fact that a single artifact of the software development can be subject to several viewpoints. Thus, it is a major concern to ensure the consistency between different viewpoints.

The other two classes—conserving and monolithic integrations—differ from viewpoint-oriented combinations in that they fix a particular decomposition style, i.e., they define a fixed set of base formalisms to be deployed and fixed correspondences that relate the selected base formalisms to each other.

Monolithic Integrations. The other end of the spectrum, with a high degree of integration, is occupied by the class of monolithic integrations. In this class, the chosen base formalisms are integrated very closely leading—more or less—to a completely new formalism.

In the first step of the integration, selected elements of the notations of the base formalisms are merged, yielding the notation of the integrated formalism. That is, monolithic integrations do not pursue the separation of their base notations. The next step lifts the selected elements of the base notations to a uniform semantic interpretation, which necessarily differs from their original interpretation. This lifting, however, impedes the reuse of the semantic definitions of the base formalisms when defining the unified semantics of the integrated formalism. This can be illustrated with the help of the monolithic integration TCOZ. In TCOZ, the notations of Object-Z and timed CSP are merged very deeply: Object-Z operation schemas can be used in timed CSP expressions to represent terminating processes, and the TCOZ state guard operator uses an Object-Z schema expression to prefix a timed CSP process expression in order to make the flow of control dependent on the current data state. Therefore, Object-Z schema expressions and timed CSP expressions need to be lifted on a uniform semantic interpretation; the semantic function of timed CSP cannot be applied to these TCOZ process expressions because of the contained Object-Z expressions.

Instances of this class are the RAISE specification language (RSL), CSP-OZ and TCOZ.

The most notable merit of monolithic integrations is the strong coherence between the base formalisms and the notational conciseness of being able to directly merge the syntaxes of the base formalisms. Their essential drawback is due to the fact that the base formalisms are not identifiable parts of the integrated formalism, impeding the reuse of their infrastructure.

Conserving Integrations. Last but not least, conserving integrations are located in the centre of the spectrum. The major design criterion for conserving integrations is the possibility to reuse the infrastructure of the selected base formalisms. This is the distinguishing feature with respect to the class of monolithic integrations. Of particular interest for this distinction is the reuse of the semantic functions of the base formalisms within the unified semantic model of the integrated formalism.

Note that the transition between conserving and monolithic integrations is fluid, depicted in Figure 5.1 by the dashed arrow: also monolithic integrations can make use of the semantic definitions of their base formalisms to a restricted extent. Ultimately, the crucial decision concerns the direction in which the trade-off between the potential to reuse the infrastructure of the base formalisms on the one hand and the coherence and uniformity of the integrated

formalism on the other hand is resolved. The reuse of the semantic definitions is only an indication for the direction of this resolution.

To achieve this reuse potential, conserving integrations interrelate their base formalisms more loosely: each base formalism constitutes an identifiable part of the integrated formalism. Accordingly, each specification of an integrated formalism is composed of several separated *parts*, each obtained by encapsulating the notation of a single base formalism. The integration of these notationally separated parts to constitute a single, coherent unit is achieved by defining correspondences (links) between specific notational elements of the different parts. These correspondences are general in that they apply to all specifications of an integrated formalism. The above points can be illustrated with the help of the approach of Smith and Derrick [2001], in which the notations of Object-Z and CSP are strictly separated: the Object-Z notation is used to define the behaviour of the system components and the CSP notation is used to define the configuration of these components and their interaction. The link between the parts consists in identifying processes in the CSP part and classes in the Object-Z part that have identical identifiers.

Instances of this class are LOTOS, RT-Z and the approach proposed by Smith and Derrick [2001]. A further instance is the approach of Treharne [2000], who has developed an integration of B and CSP, which uses CSP process descriptions (called control executives) to control the order of executing the operations that are specified in a B Abstract System. Finally, Choppy et al. [2000] have proposed a conserving integration of an algebraic specification language and state transition diagrams; the integrated formalism is called Korrigan.

The most important feature of a conserving integration is the architecture of composing its base formalisms. In fact, the used base formalisms should be exchangeable. For LOTOS, e.g., as an instance of this class, it is planned to replace its abstract data type language.

A major benefit of conserving integrations, as just indicated, is their ability to reuse the existing infrastructure of their base formalisms, e.g., tool support, proof support and the knowledge and experience of existing users. Compared with viewpoint-oriented combinations, however, this reuse potential is more restricted because of the interdependencies between the integrated base formalisms. A disadvantage, on the other hand, which is a consequence of the separation of the base notations, is the lower notational conciseness and convenience than in monolithic integrations.

To conclude, conserving integrations are a compromise between viewpoint-oriented combinations and monolithic integrations; they hence avoid some disadvantages of the other classes, but cannot fully exploit their advantages.

Part II

RT-Z: An Integration of Z and Timed CSP

This part presents a conserving integration of the formal specification languages Z and timed CSP. This integrated formalism is designed to support the development process of real-time embedded systems in several phases—from the requirements engineering via the architectural design to the detailed design.

Embedded systems usually exhibit complex interrelations with their external environment in addition to complex internal structures. A specification language intended for the chosen application domain must cope with several dimensions of this complexity, mainly

- the static structure,
- the dynamic behaviour and
- the data-related characteristics

of embedded systems.

Let us first clarify the above terminology. The term *static structure* refers to invariant characteristics of an embedded system, e.g., the structure of its interface and its architecture. We use the term *dynamic behaviour* to combine the terms *external stimulus–response behaviour* and *internal control behaviour*. The external stimulus–response behaviour of an embedded system incorporates aspects like the temporal ordering of external interactions with the environment and also real-time constraints on their occurrence. It defines the reaction of the embedded system to external stimuli in terms of emitting responses into the environment. The internal control behaviour, on the other hand, involves the distribution and control of internal actions that implement the external interaction with the environment. Finally, the term *data-related characteristics* encompasses aspects like the structure and the invariant properties of the data state[1] of an embedded system as well as the relationships that must hold between the data values communicated with particular external interactions.

The above dimensions roughly correspond to the three views on embedded control systems identified by Weber [1997], which he termed architectural model, behavioural model and functional model. We do not adopt this terminology, because these terms suggest a viewpoint-oriented approach (cf. Chapter 5), which we do not follow as we will shortly argue.

The Z notation [Woodcock and Davies, 1996] is a powerful vehicle for specifying data-related characteristics and fragments of the static structure. However, it is not designed to model aspects of the dynamic behaviour. The real-time process algebra timed CSP [Schneider, 1999b], on the other hand, is a powerful tool for specifying the dynamic behaviour, including real-time aspects, and fragments of the static structure. However, it does not provide adequate constructs to model data-related characteristics. The two formalisms can hence be considered to be complementary, covering disjoint aspects of real-time embedded systems. Further, their underlying concepts are well suited to each other. This part deals with an integration of Z and timed CSP, called RT-Z. It exploits the strengths of both formalisms in

[1] An embedded system maintains an internal data state in order to control its interaction with the environment. This data state reflects relevant properties of its past interaction that determine its ability to interact with the environment in the present and future. Transitions on that data state (operations) are defined in order to react to the occurrence of external interactions.

order to provide a smoothly integrated, single formalism to model all relevant dimensions of real-time embedded systems. The base formalisms of RT-Z are outlined in Appendix A.

For a description of the general features of RT-Z that is more compact than our treatment in this thesis, consult [Sühl, 2002] and [Sühl, 1999]. A predecessor of RT-Z is discussed in [Heisel and Sühl, 1996].

This part is organised as follows. In Chapter 6, we describe the general principles underlying our integration of Z and timed CSP. In particular, we distinguish between two models of integration—an abstract and a concrete one—needed to cover the different abstraction levels involved during the development process. Then, in Chapter 7, we introduce specification units, the building blocks of RT-Z specifications, which constitute the frames within which the notations of Z and timed CSP are integrated. Finally, the structuring mechanisms of RT-Z, which allow us to decompose the specification of complex, highly concurrent systems, are introduced and defined in Chapter 8.

Chapter 6

Integration Principles

According to our classification of approaches to combining formalisms proposed in Chapter 5, RT-Z is a *conserving integration* of Z and timed CSP. The notations of the base formalisms are hence separated from each other. Accordingly, each RT-Z specification consists of two *parts*:

- The data-related characteristics and fragments of the static structure are specified by using the Z notation. We refer to this part as the 'Z part.'
- The dynamic behaviour and the remaining fragments of the static structure are specified by using the notation of timed CSP. We call this part 'CSP part.'

As a conserving integration, RT-Z defines several links between specific elements of its parts; these links achieve the integration of the notationally separated parts to constitute a single, uniform specification.

The fragments of the notations of Z and timed CSP used in the parts and the links with which the parts are related to each other depend on the particular phase within the development process that is to be supported. There is an abstract and a concrete model of integration; they are the subject of the following two sections.

6.1 Specifying Properties (Abstract Model of Integration)

The aim of the early phases of the development process, i.e., the system and software requirements specification phases, is to specify properties of the artifact[1] in question without fixing any implementation aspects. The following types of properties must be covered by a formalism designed to specify real-time embedded systems.

1. The *temporal ordering* of interactions at the external interface of the considered artifact, possibly including *real-time constraints*.

[1] An artifact can be a real (physical) system or a piece of software depending on the current phase of the system development process.

2. *Relationships that must hold between data values* communicated between the considered artifact and its environment via particular external interactions (I/O relations).

3. *Properties of data states* present at specific time instants during the evolution of the artifact.

It depends on the current phase of the development process which of these types are really relevant. In this section, a predicate language is introduced that is suited to the abstract specification of all these types of properties, in separation as well as in connection with each other. Evidently, properties of the first two types can hardly be specified in isolation, because data values, about which requirements are to be specified, are communicated with interactions at the external interface. The predicate language, informally introduced in this chapter, is backed up formally in Chapter 9. The purpose of each predicate is to induce the set of valid timed observations of the artifact under consideration. The structure of these timed observations is defined in detail in Chapter 9; for the time being, let us outline their structure as follows: each timed observation associated with an RT-Z specification consists of a timed trace s, a timed refusal $\aleph$ and a timed state tst. The structure of the pair $(s, \aleph)$ is identical with the denotations of timed CSP processes and denotes the external interaction of the specified artifact within the observation interval. The timed state tst, on the other hand, is a function that records the evolution of the artifact's data state in the observation interval. In other words, the purpose of each predicate of RT-Z's predicate language is to induce a relation between the timed trace, timed refusal and timed state components of timed observations (denotations). Throughout this thesis we use the convention that the timed trace, timed refusal and timed state components of a timed observation are denoted by the identifiers s, $\aleph$ and tst, respectively. That is, these identifiers are free variables of any predicate. We do not make explicit this dependence of RT-Z predicates on these identifiers.

Concerning the specification of temporal ordering and real-time properties, i.e., properties with respect to the acceptance and refusal of interactions at the external interface, we adopt the macro language introduced by Schneider [1999b], which we discuss in Section A.2. These macros refer to the timed failure component $(s, \aleph)$ of a timed observation.

Regarding the second type of properties—the specification of I/O relations—we include a new language construct into the predicate language: references to Z schemas. For each (separately specified) Z schema

$IORel == [\, var1 : T1;\ \ldots;\ varN : TN \mid Pred\,]$,

the expressions of the predicate language can contain the reference

$IORel(var1 \mathrel{\widehat{=}} expr1, \ldots, varN \mathrel{\widehat{=}} exprN)$,

where $expr1, \ldots, exprN$ are timed CSP expressions evaluating to an (untimed) trace or refusal.[2] These trace and refusal expressions contain the information which particular data values have been communicated with the occurrence of particular events along the corresponding channels (and have been refused to be communicated, respectively). The above schema reference associates the schema components $var1, \ldots, varN$ with the expressions

[2] Untimed refusals and traces are sets and finite sequences corresponding to the Z type constructors $\mathbb{P}$ and seq, respectively. $T1, \ldots, TN$ have thus the form $\mathbb{P}\, X$ or $\mathrm{seq}\, X$.

$expr1, \ldots, exprN$. The informal meaning of this reference is that the evaluated expressions $expr1, \ldots, exprN$ must satisfy the conditions specified by the schema *IORel* with respect to the variables $var1, \ldots, varN$.

Therefore, the ability to refer to Z schemas and to associate their components with trace and refusal expressions allows us to express relationships between data values that are communicated with particular events at the external interface. This is achieved by means of the Z notation, which is tailored to this task.

We illustrate the predicate language of RT-Z with the help of the requirements specification of a reliable communication medium, which is situated between an input channel *InChn* and an output channel *OutChn* and must transmit arbitrary messages in the correct order without any corruption. In the following, we discuss the specification of three requirements depicted in Figure 6.1, which the medium must meet.

The first requirement *Req*1 concerns the temporal ordering of external interactions. At any time during the protocol execution, at most one message delivered to the protocol machine at the input channel (*InChn*) may be pending, i.e., not yet delivered to the environment at the output channel (*OutChn*). This means that a new message should be accepted at the input channel only when a successful output on the output channel has been acknowledged.

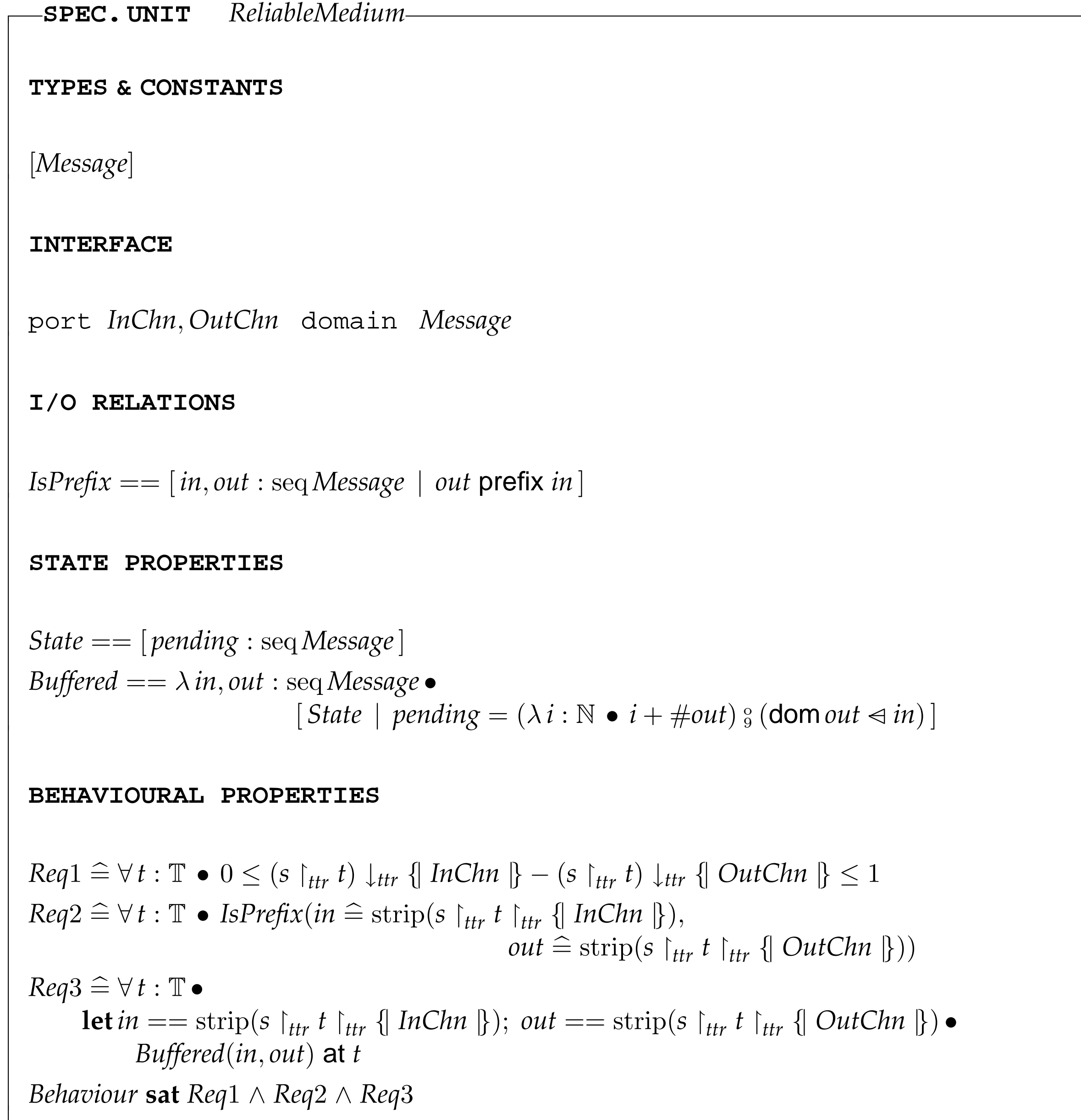
$$Req1 \mathrel{\widehat{=}} \forall t : \mathbb{T} \bullet 0 \leq (s \upharpoonright_{ttr} t) \downarrow_{ttr} \{\!|\; InChn \;|\!\} - (s \upharpoonright_{ttr} t) \downarrow_{ttr} \{\!|\; OutChn \;|\!\} \leq 1$$

The above predicate restricts the set of timed traces s that are valid observations of the protocol behaviour (the components $\aleph$ and tst are left unrestricted). It uses functions operating on timed trace expressions, which are explained in Appendix E. The expression $s \upharpoonright_{ttr} t$ yields the timed trace up to and including time t, which records all timed events that occurred before time t. Then, the expression $s \upharpoonright_{ttr} t \downarrow_{ttr} \{\!|\; InChn \;|\!\}$ yields the number of events that occurred on channel *InChn* before time t. Thus, in the above predicate, we compare the number of events that are communicated along the channels *InChn* and *OutChn*, respectively, for each time interval $[0, t]$.

The second requirement *Req*2 involves the data values communicated with external interactions: if a message is delivered to the environment at the output channel *OutChn*, then this message must not be corrupted, i.e., the contents of transmitted messages must not be changed.

$$IsPrefix == [\, in, out : \operatorname{seq} Message \mid out \ \mathsf{prefix} \ in \,]$$
$$Req2 \mathrel{\widehat{=}} \forall t : \mathbb{T} \bullet IsPrefix(in \mathrel{\widehat{=}} \mathrm{strip}((s \upharpoonright_{ttr} t) \upharpoonright_{ttr} \{\!|\; InChn \;|\!\}), \; out \mathrel{\widehat{=}} \mathrm{strip}((s \upharpoonright_{ttr} t) \upharpoonright_{ttr} \{\!|\; OutChn \;|\!\}))$$

The I/O relation schema *IsPrefix* simply specifies a relationship between the sequences *in* and *out*: *out* must be an initial prefix of *in*. This Z schema, interpreted in isolation, does not restrict the behaviour of the protocol at all; only the association of the schema components *in* and *out* with particular trace expressions within the predicate *Req*2 achieves this. The schema reference associates *in* with the expression $strip(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\; InChn \;|\!\})$ (the events that occurred on channel *InChn* before time t) and the component *out* with the expression $\mathrm{strip}(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\; OutChn \;|\!\})$ (the events that occurred on channel *OutChn* before time t). *Req*2 therefore requires, for each time instant t, that the sequence of messages delivered at the output channel must be a prefix of the sequence of messages received at the input channel.



SPEC. UNIT *ReliableMedium*

TYPES & CONSTANTS

$[Message]$

INTERFACE

`port` $InChn, OutChn$ `domain` $Message$

I/O RELATIONS

$IsPrefix == [\, in, out : \operatorname{seq} Message \mid out \text{ prefix } in \,]$

STATE PROPERTIES

$State == [\, pending : \operatorname{seq} Message \,]$

$Buffered == \lambda\, in, out : \operatorname{seq} Message \bullet$
$\qquad [\, State \mid pending = (\lambda\, i : \mathbb{N} \bullet i + \#out) \fatsemi (\operatorname{dom} out \lhd in) \,]$

BEHAVIOURAL PROPERTIES

$Req1 \mathrel{\widehat=} \forall\, t : \mathbb{T} \bullet 0 \leq (s \upharpoonright_{ttr} t) \downarrow_{ttr} \{\!|\, InChn \,|\!\} - (s \upharpoonright_{ttr} t) \downarrow_{ttr} \{\!|\, OutChn \,|\!\} \leq 1$

$Req2 \mathrel{\widehat=} \forall\, t : \mathbb{T} \bullet IsPrefix(in \mathrel{\widehat=} \operatorname{strip}(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\, InChn \,|\!\}),$
$\qquad out \mathrel{\widehat=} \operatorname{strip}(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\, OutChn \,|\!\}))$

$Req3 \mathrel{\widehat=} \forall\, t : \mathbb{T} \bullet$
$\quad \mathbf{let}\, in == \operatorname{strip}(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\, InChn \,|\!\});\; out == \operatorname{strip}(s \upharpoonright_{ttr} t \upharpoonright_{ttr} \{\!|\, OutChn \,|\!\}) \bullet$
$\qquad Buffered(in, out) \text{ at } t$

$Behaviour\ \mathbf{sat}\ Req1 \wedge Req2 \wedge Req3$

Figure 6.1: Reliable communication medium: requirements.

Finally, properties of the third type are specified in isolation as well as in connection with properties of the first two types. On the one hand, one might want to specify invariants of the data state of the artifact in question. On the other hand, one might be faced with properties of the data state that must be satisfied depending on particular interactions at the interface. To enable us to formulate requirements with respect to the evolution of the data state, i.e.,

to refer to the timed state component (*tst*) of a timed observation, we provide additional specification macros. They state properties that particular data states within the observation interval must exhibit.

We call such properties 'time-variant state properties.' Each of the additional specification macros expressing time-variant state properties relates a Z schema, specifying a particular property of the data state, and a time interval (or time instant). The timed state component *tst* of a timed observation meets such a specification macro if, and only if, all data states that are recorded in *tst* within the respective time interval (at the respective time instant) have the property expressed by the respective schema.

In this section, we give only an informal description of these specification macros. A complete definition is possible only after the discussion of the semantic model in Chapter 9. Suppose in the following that *Prop* is a Z schema.

at: The specification macro (*Prop* at *t*) is satisfied by a timed state *tst* if, and only if, the last data state that is recorded in the timed state *tst* before time instant *t* (inclusive) has property *Prop*.

before: The specification macro (*Prop* before *t*) is satisfied by a timed state *tst* if, and only if, the last data state that is recorded in the timed state *tst* before time instant *t* (exclusive) has property *Prop*.

during: The specification macro (*Prop* during *INT*) is satisfied by a timed state *tst* if, and only if, all data states of *tst* during interval *INT* have property *Prop*.

invariant: (*Prop* invariant) is satisfied by a timed state *tst* if, and only if, each data state of *tst* has property *Prop*, i.e., if *Prop* is an invariant property.

Returning to our example, there is a time-variant state property we should require: at any time during the protocol execution, the protocol machine must record in its data state the sequence of messages (*pending*) received on the input channel but not yet delivered to the output channel.[3]

$$State == [\, pending : \mathrm{seq}\, Message \,]$$
$$Buffered == \lambda\, in, out : \mathrm{seq}\, Message \bullet [\, State \mid pending = (\lambda\, i : \mathbb{N} \bullet i + \#out) \fatsemi (\mathsf{dom}\, out \lhd in) \,]$$
$$Req3 \mathrel{\widehat=} \forall\, t : \mathbb{T} \bullet$$
$$\quad \mathbf{let}\, in == \mathrm{strip}((s \restriction_{ttr} t) \restriction_{ttr} \{\!| InChn |\!\}); \; out == \mathrm{strip}((s \restriction_{ttr} t) \restriction_{ttr} \{\!| OutChn |\!\}) \bullet$$
$$\quad Buffered(in, out) \;\mathsf{at}\; t$$

The *State* schema introduces the data state of the protocol with a single component *pending*. Further, *Buffered* is a function whose range are the subsets of schema bindings associated with the schema *State*. Applied to the message sequences *in* and *out*, it restricts the component *pending* to contain the difference between the sequences *in* and *out*, i.e., the messages contained in *in* but not contained in *out*. The function is applied to the sequence of messages received at the channel *InChn* and the sequence of messages delivered at the channel *OutChn*

[3] In our example, *pending* can contain at least one message.

in the context of the predicate *Req*3. Altogether, *Req*3 requires that the protocol must always record the messages already received but not yet delivered in an appropriate order.

Note that the specification of time-variant state properties makes sense only if a model of the data state has been defined before. Specifying properties of the data state is reasonable only in the *system* requirements specification phase,[4] in which one is concerned with systems with a 'physical' state and where one must guarantee properties of this 'physical' state.

We add to the syntax of predicates as just described the ability to parametrise predicate definitions by a list of identifiers.

$ID(fp1, \ldots, fpN) \mathrel{\widehat{=}} Pred$

Pred in the above definition can contain free occurrences of $fp1, \ldots, fpN$ at arbitrary places. Each reference to such a parametrised predicate definition must be supplied with a list of actual parameters of the same length. Formal and actual parameters are identified according to their position in both lists. The meaning of referring to such a parametrised predicate definition is given by a simple textual substitution: the reference is substituted by the right hand side of the predicate definition where all free occurrences of formal parameters are simultaneously substituted by the corresponding actual parameters, supplied in the reference.

Let us elaborate the above in the light of our classification of RT-Z as a conserving integration. Each RT-Z specification in the abstract model consists of two parts: a Z part encapsulating the Z notation, used to express properties of data values communicated at the external interface and of the data state evolution; and a CSP part encapsulating the timed CSP notation, applied to formulate properties of the dynamic behaviour. Basically, there are three types of links between the parts, which are outlined in Figure 6.2.

First, interactions between an artifact and its environment are modelled in timed CSP by synchronous events, which may represent mere synchronisations or the communication of arbitrarily complex data values. Accordingly, timed CSP allows one to associate type information with channels. However, this is supported only in a rather rudimentary way, because timed CSP lacks an appropriate notation to specify data types.[5] To overcome this problem in RT-Z, we associate a *value domain* with each channel, i.e., an expression that, evaluated in the context of the Z part, denotes a set of data values. That is, the occurrence of an event on a particular channel coincides with the transmission of a data value that must be a member of the associated set.

Second, the predicates in the CSP part, defining the dynamic behaviour, can refer to schemas defined in the Z part in order to express relationships that must hold between data values communicated with external interactions (I/O relations). Within such a schema reference, data values communicated with particular events are bound to the components of the referenced schema.

Finally, the predicates in the CSP part can refer to schemas specified in the Z part in order to define time-variant state properties, i.e., properties that the evolution of the data state must

[4] as opposed to the software requirements specification phase

[5] Regarding plain CSP, the current input language of the model checker FDR2 [Formal Systems (Europe) Ltd, 2000], CSP_M, comprises a syntax for expressions, types and channel declarations. Moreover, Scattergood [1998] has defined a formal semantics for CSP_M.

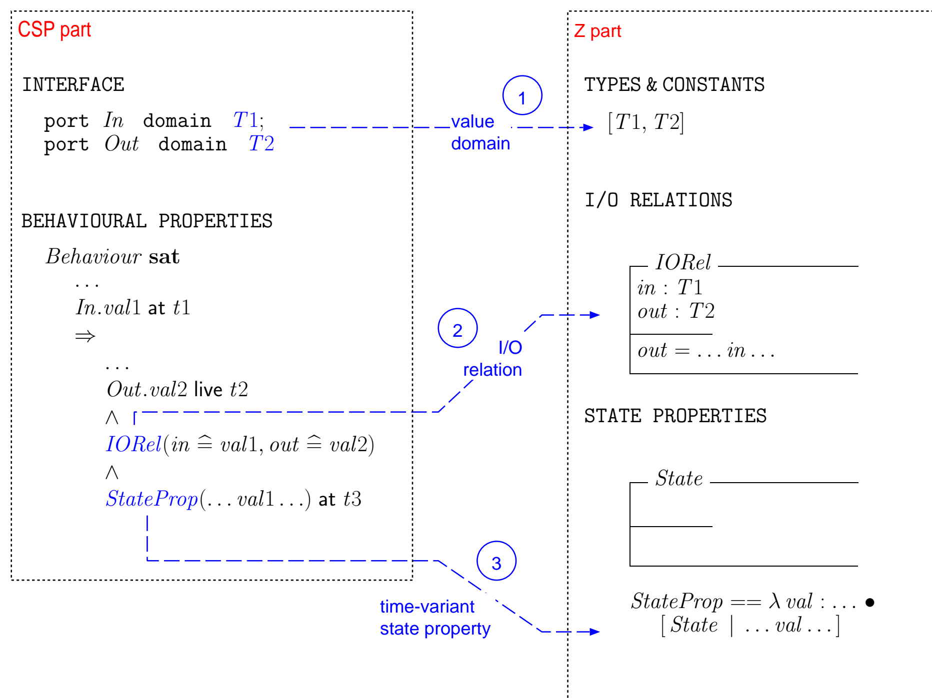


Figure 6.2: Links between the parts.

meet in the context of particular external interactions. To this end, we employ the macro language introduced in this section.

The meaning of an RT-Z specification in the abstract model, as formally defined in Chapter 9, is the meaning of its CSP part. The Z part serves to establish the context for the CSP part.

6.2 Specifying Models (Concrete Model of Integration)

The aim of the later phases of the development process, i.e., the system and software design, is to fix a more or less detailed model of the implementation of the considered artifact, which has to satisfy properties, abstractly specified as outlined in the previous section.

Consider first the Z part of an RT-Z specification in the concrete model. Its purpose is the specification of the data-related characteristics and fragments of the static structure of the artifact under consideration. The following aspects are to be covered.

- The structure and invariant properties of the valid data states.
- Operations on that data state, which define state transitions.
- Predicates on that data state, expressing conditions to be evaluated with respect to the current data state.

- Constants denoting parameters of the artifact under consideration.
- Data types associated with the channels of the interface, restricting the set of data values that can be communicated.

In the current, concrete model of integrating Z and timed CSP, we apply the predominant state-oriented conventions of using Z to formulate the above aspects. This is a contrast to the abstract model dealt with in the previous section.

Following Broy et al. [1992], the specification of a data state and its operations does not necessarily constrain the final implementation to have an implementation of the data state and all its operations. Rather, the state machine model is a tool for specifying the data-related characteristics at the external interface more succinctly and conveniently. The current data state of an artifact can be interpreted as representing the relevant properties of the past interaction that determine the artifact's ability to interact with the environment in the present and future.

The CSP part, on the other hand, serves to specify the dynamic behaviour and the remaining fragments of the static structure of the considered artifact, including the following aspects.

- The external stimulus–response behaviour, i.e., its reaction to external stimuli (input events at the external interface which can carry arbitrarily complex values) in terms of emitting responses (output events at the external interface which can also carry arbitrarily complex values) into the environment.
- The internal control behaviour, i.e., how it distributes and controls internal actions in order to implement the external interaction with the environment.
- The architecture of the considered artifact, i.e., its decomposition into particular components including their interaction via particular channels.
- The structure of its interface.

In the current, concrete model we use the process term language of timed CSP, which allows us to fix concrete models covering the above aspects.

Links. Basically, there are four types of links between the Z part and the CSP part, which are shown in the Figures 6.3 and 6.4.

The first type of link is also present in the abstract model of integration: channels in the CSP part are associated with value domains defined in the Z part.

Second, timed CSP provides indexed forms of some operators, e.g., of the interleaving operator $|||$. They are parametrised by an index set, whose members are used to identify the individual processes being composed. However, timed CSP does not provide an adequate notation to specify these index sets formally. In RT-Z, by contrast, each index set is formally defined in terms of an expression that, evaluated in the context of the Z part, denotes a set of index values.

Third, timed CSP extends (plain) CSP with real-time operators, which are operators with real-valued parameters. The Z part serves, among other things, to declare and constrain

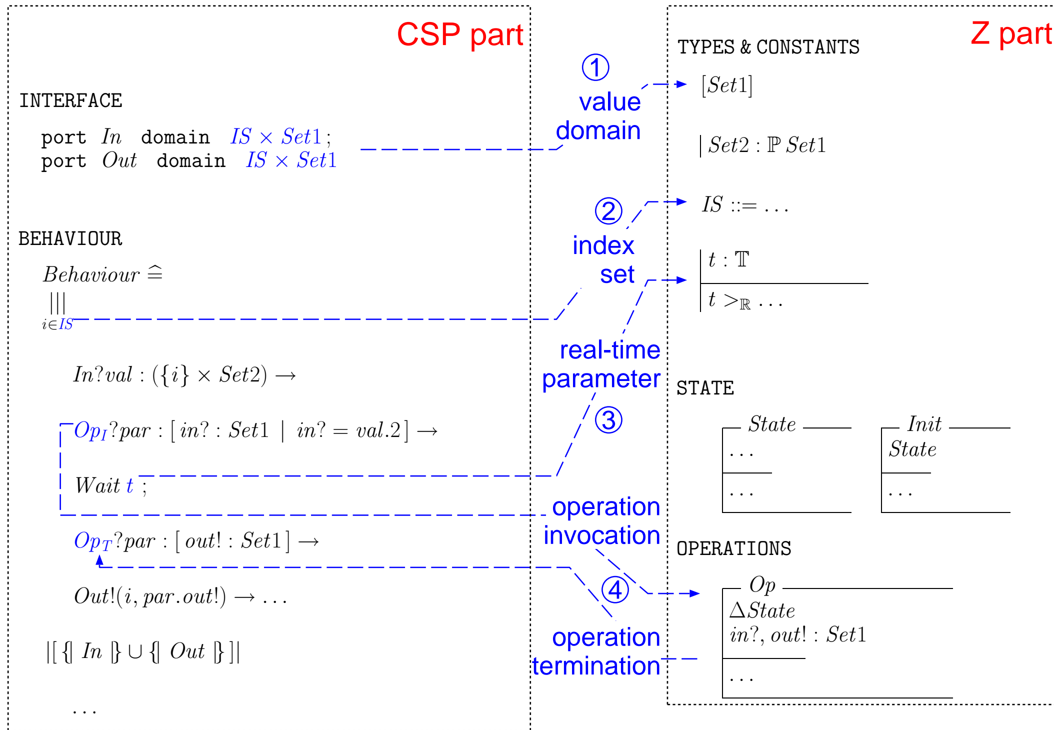


Figure 6.3: Links between the parts (double event approach).

constants of type $\mathbb{T}$, which denotes the time domain of timed CSP and also of RT-Z. Each real-valued parameter occurring in the CSP part must be given in terms of an expression that, evaluated in the context of the Z part, denotes a value of type $\mathbb{T}$.

Finally, one of the main tasks of the CSP part, as already indicated, is to control the execution of the operations specified in the Z part. To model this control relationship, there are two alternatives for relating Z operations and CSP events. The choice depends on the required abstraction level; more specifically, it depends on whether or not the specifier intends to abstract from the time needed to execute an operation.

In the case in which the time interval of executing an operation, say *Op*, is relevant, the operation schema *Op* of the Z part is related to the pair of dedicated channels Op_I and Op_T, on which the CSP part communicates in order to control the execution of the operation. This case is depicted in Figure 6.3. Events on the channel Op_I represent the invocation of the operation *Op* with a particular assignment of values to the input parameters. The data value that is communicated with such an invocation event between the invoking CSP process and the invoked Z operation is a schema binding mapping each input parameter of the corresponding operation schema to a value. It denotes the agreement of the two parts on particular values of the input parameters with which the operation is invoked.[6] Events on the second channel of the pair, Op_T, represent the termination of the invoked operation *Op*. The produced output parameters are transmitted with such a termination event between the

[6] Each channel Op_I is implicitly associated with the value domain that includes all bindings of the input parameters of the operation schema *Op*.

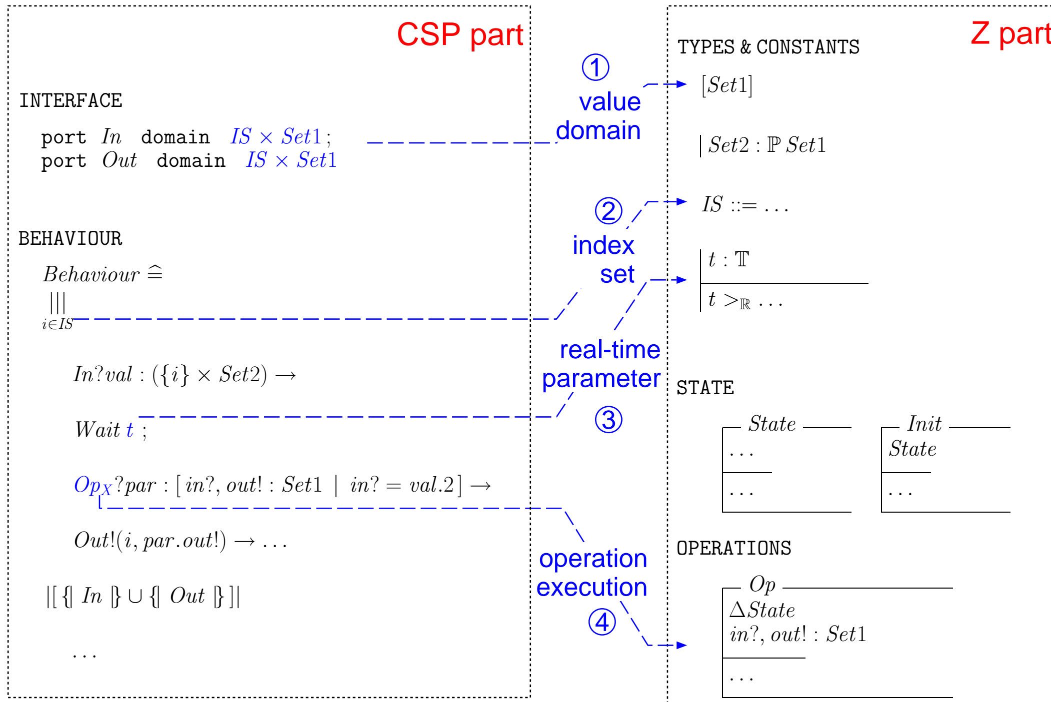


Figure 6.4: Links between the parts (single event approach).

terminated operation and the invoking CSP process.[7]

Relating an operation to a pair of channels and an operation execution to a pair of events, respectively, allows us to specify requirements concerning the interval of an operation execution, for instance, constraints on its duration.

In the other case, in which these time intervals are not relevant, an operation schema *Op* of the Z part is related to a single channel Op_X. This case is depicted in Figure 6.4. Events on this channel represent the *instantaneous* execution of the operation. The data value that is communicated with such an execution event between the executed Z operation and the executing CSP process is a schema binding mapping each input and output parameter of the corresponding operation schema to a value. It denotes the agreement of the two parts on particular values of the input parameters and the sole fixing of the output parameter values by the Z operation.[8]

According to the terminology proposed by Fischer [1998], these cases are called the double and single-event approach, respectively. They can be combined arbitrarily in a single RT-Z specification. That is, some operations of the Z part can be controlled by means of the single-event approach, while the remaining operations are controlled by using the double-event approach.

[7] Each channel Op_T is implicitly associated with the value domain that includes all bindings of the output parameters of the operation schema *Op*.

[8] Each channel Op_X is implicitly associated with the value domain that includes all bindings of the input and output parameters of the operation schema *Op*.

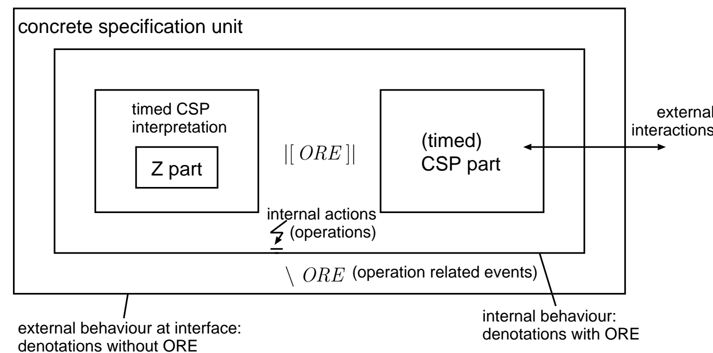


Figure 6.5: Principle of composing the Z part and the CSP part.

Main Principle. Now, we are in a position to discuss the main principle underlying the integration of Z and timed CSP in the concrete model. This principle follows an idea of Smith [1997], which he applied to the integration of plain CSP and Object-Z. Following this idea and adapting it to our specific context, we are able to interpret a Z specification as a timed CSP process: a process that controls the events related to its operations as follows. It accepts the invocation/execution of an operation with particular input parameters if, and only if, the precondition of the corresponding operation schema is satisfied with respect to its current data state and the supplied input parameters; otherwise, it blocks the invocation/execution of the operation. The termination of an invoked operation with particular output parameters is accepted under analogous conditions.

By interpreting the Z part of an RT-Z specification as a timed CSP process, the meaning of the overall specification is defined as the parallel composition of the CSP part and the timed CSP interpretation of the Z part, where both parts synchronise on the set of operation-related events, i.e., on the execution, invocation and termination of operations. These operation-related events are defined to be internal by hiding them from the environment of the parallel composition. That is, the occurrence of operation-related events is completely controlled by the considered artifact and not visible to the environment. This principle of composing the parts is outlined in Figure 6.5.

As a consequence of composing the Z and the CSP part in the chosen way, an operation is invoked/executed only if the corresponding invocation/execution event is accepted by both parts simultaneously; when both parts accept an event, they also agree on a specific assignment of values to the input parameters. This means that the specific values with which an operation is invoked/executed are equally influenced by both the corresponding Z operation schema and the value denotation of the corresponding CSP event.

Note that we adopt a view of the application of Z operations that is usually called the 'blocking view': an operation can be invoked/executed only within its precondition; outside its precondition, the invocation/execution is blocked.

Due to the chosen way of composing the two parts, we are able to express another relationship between the Z and the CSP part by employing only the links already described. The dynamic behaviour of an artifact, specified in the CSP part, usually depends on its current data state, specified in the Z part. In RT-Z, such a kind of dependence is modelled in terms

of an operation schema, say *Pred*, that does not change the data state ($\Xi State$) and whose precondition is equivalent to the condition on the data state to be evaluated. According to the integration principles just discussed, the occurrence of the execution event $Pred_X$ implies the successful evaluation of the precondition of the operation schema with respect to the current data state, in other words, the successful evaluation of the condition to be evaluated.

Parameter Exchange. Let us have a closer look at the interaction between the Z and the CSP part that takes place when an operation is executed (single event approach). An execution event, representing the instantaneous application of an operation, communicates the input and output parameters of the operation. The values of the input parameters must be agreed on by both the Z operation and the executing CSP process, whereas the values of the output parameters are solely determined by the executed Z operation. We discuss the structure of the value denotation of an execution event within the CSP part.

The value communicated with an execution event is a binding, mapping each operation parameter to a value. Since the output parameters should be solely determined by the Z operation, the value denotation of the execution event in the CSP process must allow arbitrary values for the output parameters. In contrast, the value denotation of the execution event must allow us to express arbitrary constraints on the values of the input parameters. These constraints usually depend on values bound by the CSP process during prior event occurrences. The Z operation, by definition, accepts all combinations of input parameter values that satisfy its precondition with respect to the current data state.

Let us consider the value denotation of an execution event Op_X as an example.

$$Op_X?par : [\, in? : T1;\ out! : T2 \mid in? = in\dagger \,]$$

The above schema expression defines a set of bindings of the input parameter $in?$ and the output parameter $out!$. In these bindings, $in?$ is uniquely determined by the value $in\dagger$, which can be thought of as being a variable bound during a prior event occurrence. The Z part accepts all input parameter values $in?$ satisfying the precondition of *Op* with respect to the current data state. That is, the execution event Op_X occurs if, and only if, the precondition is satisfied with respect to the combination of the value $in\dagger$ and the current data state. The value of $out!$, by contrast, is left unspecified by the schema expression and hence by the CSP part.

Analogous remarks apply to the double event approach. Let us consider the value denotation of an invocation event Op_I as an example.

$$Op_I!\langle\!|\ in? == in\dagger\ |\!\rangle$$

Since only input parameters are communicated with invocation events (no output parameters!), the binding set associated with an invocation event is singleton if, and only if, the values of all input parameters are uniquely fixed by the invoking CSP process. This is the case in the above example, where the channel output operator is used to fix this unique binding of input parameters.

On the other hand, the value denotation of a termination event, e.g.,

$$Op_T?par : [\, out! : T2 \,]$$

must allow arbitrary output parameter values, which are solely determined by the invoked operation.

This completes our discussion of the basic principles underlying the abstract and concrete model of integrating Z and timed CSP. In the next chapter, we deal with the notation of the two models of integration: abstract and concrete specification units.

Chapter 7

Specification Units

To manage the inherent complexity of systems, software and systems engineering[1] interpret a system as a hierarchical composition of interacting subsystems and system components.[2] Accordingly, the specification of a complex system in RT-Z is composed of a hierarchy of units, which we call *specification units*. A *basic* specification unit defines the behaviour of a single system component, and the appropriate composition of specification units, forming *compound* specification units, produces subsystem and ultimately system specifications. In other words, specification units are a means for structuring system and subsystem specifications. Specification units correspond to system units. A system unit can be thought of as being an instance/implementation of a specification unit, which defines its structure and behaviour. Each specification unit can have any number of instances. Hence, the relationship between specification units and system units in RT-Z can be compared with the relationship between classes and objects in an object-oriented approach. See also Figure 7.1 for an illustration of the used terminology.

Each RT-Z specification consists of two fragments: the *global definitions*, introducing constants and data types that are global with respect to the whole specification, and the definition of the structure and behaviour of the system under development, consisting of a hierarchy of specification units.

Hierarchical decomposition is a means for coping with the complexity of large and highly concurrent systems. First, the hierarchical decomposition of a system specification in RT-Z can reflect conceptual or physical structures within the problem domain, and it supports encapsulation and separation of concerns. Thus, it is a prerequisite for a clear and problem-oriented system description and hence for the intelligibility of the specified system and the ability to analyse it. Second, specification units in RT-Z constitute the frames within which concurrent activities working on a common (data) state space can be encapsulated. That is, the hierarchical decomposition of a system specification is the only way in RT-Z to model concurrency on a common data state space.

Syntactically, a specification unit is a named and delimited piece of an RT-Z specification. Its

[1] For a detailed account of the concepts underlying systems engineering consult [Kronlöf, 1993, Chapter 1] and [Leveson, 1995].

[2] System components are subsystems that are not decomposed any further.

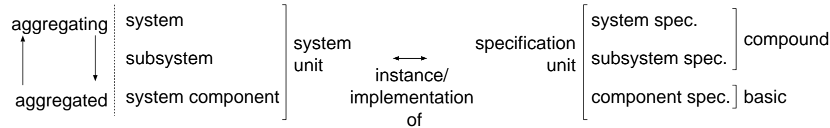


Figure 7.1: Used terminology.

coarse-grained structure is given by two parts (the Z and the CSP part), and its fine-grained structure is given by a number of sections, where each section belongs to one part.

All specification languages that aim to cope with complex systems provide syntax to compose specifications of simpler units in a hierarchical manner. Generally speaking, the RT-Z concept "specification unit" corresponds to the B concept "machine" [Abrial, 1996], the TLA$^+$ concept "module" [Lamport, 2000], the LOTOS concept "process" [Bolognesi et al., 1995], the ASTRAL concept "process type definition" [Coen-Porisini et al., 1997], the Object-Z concept "class" [Smith, 2000] etc.

In this chapter, we introduce the notation for defining single specification units. The structuring mechanisms for building compound specification units are the subject of the next chapter. As described in Chapter 6, there are two different models of integrating Z and timed CSP. The structure of a specification unit depends on the chosen model. We first discuss the structure of specification units in the concrete model; afterwards, we describe this structure for the abstract model.

7.1 Concrete Specification Units

We discuss concrete specification units with the help of the example started in Section 6.1, where we have provided the requirements specification of a reliable communication medium. In this section, we prepare the specification of the design of a particular protocol that achieves such a reliable communication, namely of the alternating bit protocol. This design specification can be completed only after the introduction of structuring mechanisms in the next chapter.

The alternating bit protocol (ABP) is a specific design to meet the requirements specification of the reliable communication medium in Section 6.1. The design is outlined in Figure 7.2.

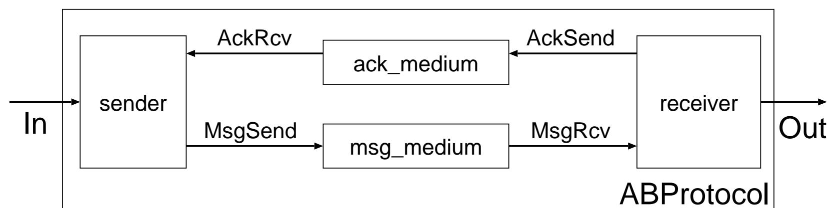


Figure 7.2: Alternating bit protocol: architecture.

There are two protocol components, *sender* and *receiver*, responsible for the protocol implementation at the input and output location and for communicating with an external sender and receiver, respectively. The protocol components are connected by a pair of unreliable, unidirectional communication media, *msg_medium* and *ack_medium*, which are themselves modelled as system components rather than channels, because they have a non-trivial, time-consuming behaviour. The message medium serves to transmit messages with an additional protocol information from the sender to the receiver component, and the acknowledgement medium serves to transmit acknowledgement bits the other way around. The protocol components interact with the communication media through the channels depicted by arrows; the channel *MsgSend*, for example, serves the sender component to deliver message/bit pairs to the message medium.

To illustrate the discussion of concrete specification units, we choose the sender component as an example, whose specification unit is depicted in Figure 7.3. Note that—for reasons of presentation—we have made the specification unit *Sender* more complex than necessary in order to cover most aspects of concrete specification units.

A template of concrete specification units is given in Figure 7.4. In the following, we discuss the sections of concrete specification units in the context of the example.

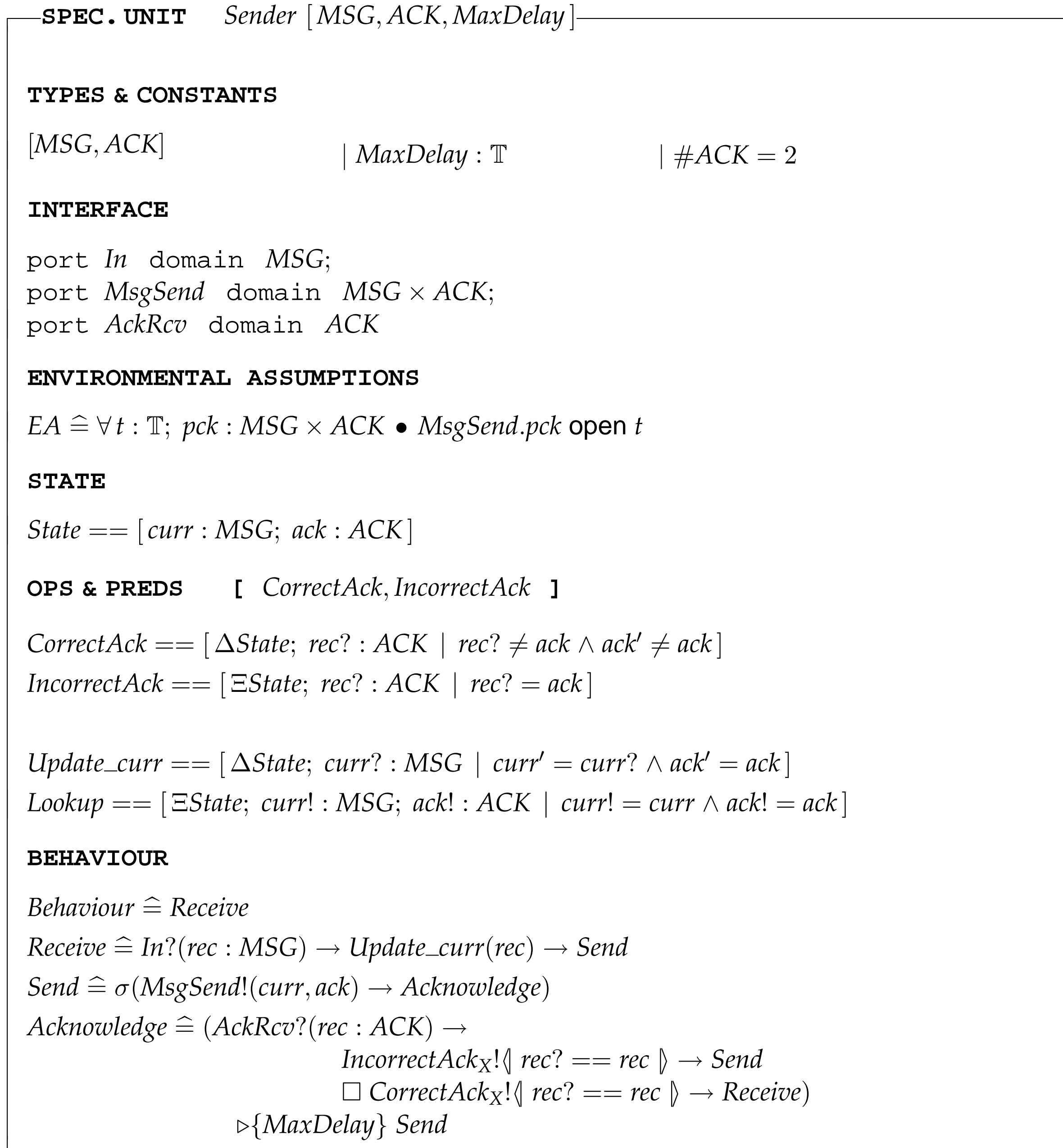
A specification unit, if concrete or abstract, can be parametrised by a list of formal parameters $fp1, \ldots, fpN$. The instantiation of these formal parameters by the context of the specification unit is discussed in detail in Section 8.5. The concrete specification unit *Sender* is parametrised by the identifiers *MSG*, *ACK* and *MaxDelay*, which represent the message and acknowledgement domains the sender component must be able to process and the time period the sender component must wait for an acknowledgement of a sent message before repeating the sending procedure.

Since the considered specification unit is a basic one, it contains neither an `EXTENDS` section nor a `SUBUNITS` section. They are discussed in the next chapter when dealing with structuring mechanisms.

`TYPES & CONSTANTS`: We distinguish between two kinds of types and constants that can occur in a specification unit.

- *Local* types and constants are declared and constrained in the `TYPES & CONSTANTS` section by using the Z notation. Their scope is the current specification unit. Therefore, local types must not be used to define the value domains of interface ports, because the scope of these local types does not encompass the specification unit's context, which must be able to refer to the interface.
- *Global* types and constants are formal parameters of a specification unit. They are instantiated by the context into which the specification unit is embedded. A specification unit can express constraints on the possible instantiations of a formal parameter. This is achieved by additionally declaring the formal parameter in the `TYPES & CONSTANTS` section and by defining appropriate constraints.

  This mechanism of defining formal parameters of specification units in conjunction with constraints on their allowed instantiations makes specification units



SPEC. UNIT $Sender\ [MSG, ACK, MaxDelay]$

TYPES & CONSTANTS

$[MSG, ACK]$ $\quad | \ MaxDelay : \mathbb{T}$ $\quad | \ \#ACK = 2$

INTERFACE

port In domain MSG;
port $MsgSend$ domain $MSG \times ACK$;
port $AckRcv$ domain ACK

ENVIRONMENTAL ASSUMPTIONS

$EA \mathrel{\widehat=} \forall t : \mathbb{T};\ pck : MSG \times ACK \bullet MsgSend.pck$ open t

STATE

$State == [\,curr : MSG;\ ack : ACK\,]$

OPS & PREDS **[** $CorrectAck, IncorrectAck$ **]**

$CorrectAck == [\,\Delta State;\ rec? : ACK \mid rec? \neq ack \wedge ack' \neq ack\,]$
$IncorrectAck == [\,\Xi State;\ rec? : ACK \mid rec? = ack\,]$

$Update_curr == [\,\Delta State;\ curr? : MSG \mid curr' = curr? \wedge ack' = ack\,]$
$Lookup == [\,\Xi State;\ curr! : MSG;\ ack! : ACK \mid curr! = curr \wedge ack! = ack\,]$

BEHAVIOUR

$Behaviour \mathrel{\widehat=} Receive$
$Receive \mathrel{\widehat=} In?(rec : MSG) \rightarrow Update_curr(rec) \rightarrow Send$
$Send \mathrel{\widehat=} \sigma(MsgSend!(curr, ack) \rightarrow Acknowledge)$
$Acknowledge \mathrel{\widehat=} (AckRcv?(rec : ACK) \rightarrow$
$\quad IncorrectAck_X! \langle\!| \ rec? == rec \ |\!\rangle \rightarrow Send$
$\quad \Box \ CorrectAck_X! \langle\!| \ rec? == rec \ |\!\rangle \rightarrow Receive)$
$\quad \triangleright \{MaxDelay\} \ Send$

Figure 7.3: Sender component: specification unit.

more self-contained.[3]

[3] In particular, the information needed to type-check the Z part of a specification unit independently of its context is available within that specification unit.

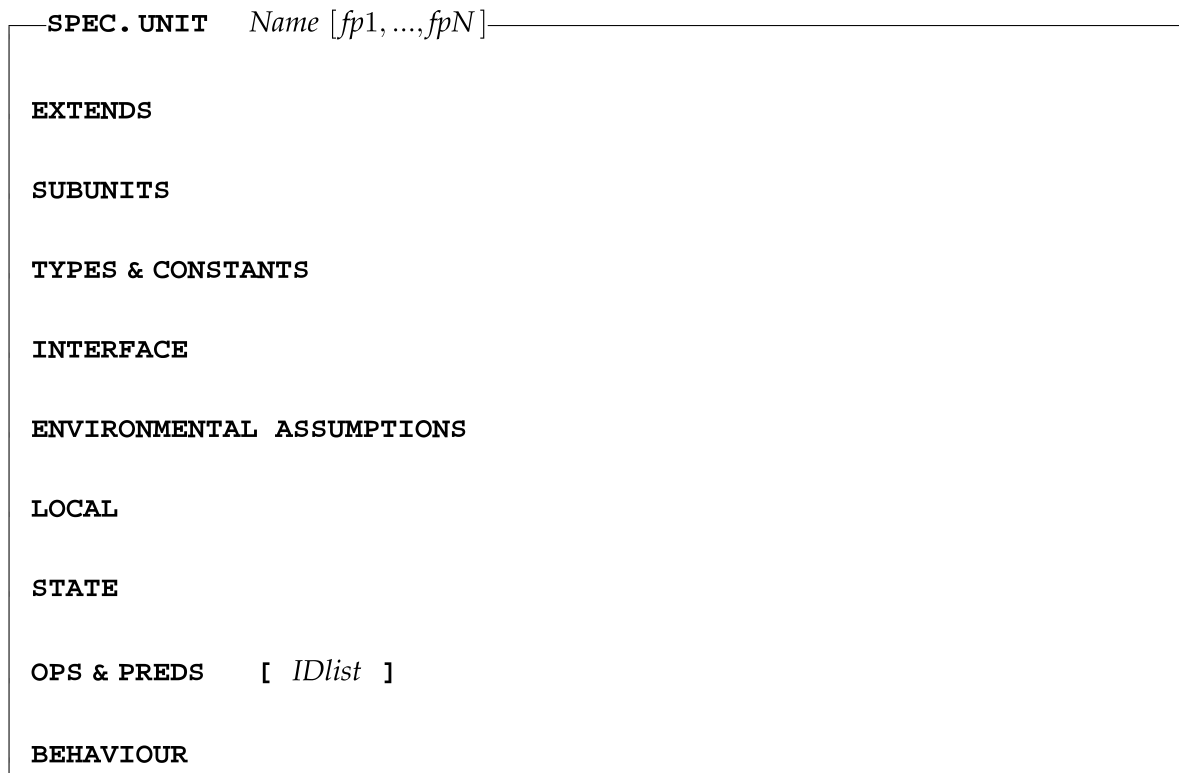


Figure 7.4: Template of concrete specification units.

Global types can be used to define the value domains of the interface ports, because their scope encompasses the context into which the specification unit is embedded.

A particular kind of constant introduced in the `TYPES & CONSTANTS` section are constants of the predefined type $\mathbb{T}$, which are used in the CSP part as parameters of real-time operators.

As indicated, the context into which the specification unit *Sender* of our running example is embedded instantiates the formal parameters. We are, however, able to express constraints on the freedom of the context to perform these instantiations. This is achieved in the `TYPES & CONSTANTS` section. First, the identifiers *MSG* and *ACK* are declared as given sets which constrains the context to instantiate these identifiers by set expressions. Moreover, the identifier *MaxDelay* is declared as a time value. Finally, *ACK* is required to consist of exactly two elements, which is vital for the chosen design.

`INTERFACE`: The interface of a specification unit consists of the communication channels, called ports, via which the communication and synchronisation with the environment is achieved. The declaration

`port` *chan* `domain` *S*

associates the Z expression *S* with the port *chan*, which is a communication channel in CSP terms. The Z expression *S*, evaluated in the context of the corresponding Z part, denotes the set of data values that can be communicated with events on that channel. The predefined Z type *SYNC* is used for channels that serve mere synchronisation without data transmission purposes.

The scope of the declared ports is the `ENVIRONMENTAL ASSUMPTIONS` and the `BEHAVIOUR` section of the current unit.

The declarations in the `INTERFACE` section of the specification unit *Sender* are self-explanatory.

`LOCAL`: This section is used to introduce internal channels, which are not part of the external interface but used only for internal communication and synchronisation purposes. The declaration

`channel` *chan* `domain` *S*

associates the Z expression *S* with the channel *chan*, where *S* denotes the set of data values that can be communicated with events on that channel. Internal channels are implicitly hidden from the environment of the process definition given in the `BEHAVIOUR` section.

The scope of the declared channels is the `BEHAVIOUR` section of the current unit.

The specification unit *Sender* does not have a `LOCAL` section, because there is no need for an internal communication and synchronisation. In fact, a `LOCAL` section is needed only for compound specification units that aggregate other ones and that need to control the interaction between the aggregated units; for more details see the next chapter.

`ENVIRONMENTAL ASSUMPTIONS`: This section serves to document the assumptions that underlie the implementation of a specification unit with respect to the behaviour of its environment at the interface. If the environment violates these assumptions, an implementation of the specification unit is free to behave arbitrarily.

Environmental assumptions are formalised with the help of an extension of the predicate language of timed CSP, defined in Section 9.2.4. The environmental assumptions of a specification unit are defined by the dedicated predicate *EA* in the `ENVIRONMENTAL ASSUMPTIONS` section. It is the task of this predicate to characterise a set of legal interactions at the interface, which are formalised in RT-Z as timed failures $(s, \aleph)$.

The `ENVIRONMENTAL ASSUMPTIONS` section of the example unit defines the predicate *EA*, which induces a relation between timed traces s and timed refusals $\aleph$. Informally, the channel *MsgSend* is always required to accept any package consisting of an arbitrary message and acknowledgement bit. This prevents the sender from being blocked. The assumption is formulated in terms of the specification macro open, which expresses that the environment must accept a communication on the given channel *MsgSend* at the given time instant t.

`STATE`: The Z notation is used to define the space of valid data states and the subset in which the implementations of a specification unit are allowed to start. This is achieved

by means of the dedicated schemas *State* and *Init*. If no *Init* schema is defined, the initial subset is fixed to be equal to the whole data state space.[4] In addition to these dedicated schemas, the `STATE` section can define further auxiliary schemas whose purpose is to structure the definition. In any case, the `STATE` section can be reduced to the *State* and *Init* schemas by expanding references to auxiliary schemas with their definition.

The context of this section is constituted by the `TYPES & CONSTANTS` section.

The data state of the example unit consists of the message and acknowledgement bit to be currently transmitted to the receiver; the inversion of the bit is expected from the receiver component as the acknowledgement for the reception of the current message. This data state is very simple and hence does not allow us to make use of the features of the Z notation to define invariant relationships between state components. Moreover, the subset of initial data states is not restricted.[5]

`OPS & PREDS`: Transitions and conditions on the data state are specified by operation schemas. All operation schemas are based on the dedicated schema *State*.

The `OPS & PREDS` section is parametrised by a list of identifiers of operations to be controlled and predicates to be evaluated by the CSP part. This list distinguishes schemas defining operations and predicates from auxiliary schemas.

For each operation schema *Op*, specifying an operation or predicate, the schema

$NotOp == \neg \ \mathsf{pre}\, Op \wedge \Xi State$

is supposed to be defined implicitly. It defines the predicate that is the negation of the precondition of the operation schema *Op*.

In addition to the operations defined explicitly, each specification unit defines several derived operations implicitly. First, it is a frequent task when defining the dynamic behaviour in the CSP part to access the current data state in order to communicate parts of it to the environment. To this end, each specification unit defines the operation *Lookup* implicitly, which binds each data state component to an equally named output parameter. The derivation of this operation schema from the *State* schema of a specification unit is straightforward. The operation *Lookup* can be used to access the components of the current data state via the corresponding output parameters. Second, it is also a frequent task when defining the dynamic behaviour to update a particular component of the data state with a value received from the environment. To this end, each specification unit defines the operation *Update_comp* for each component *comp* of the data state, which updates this component with the value of the equally named input parameter *comp*?. The derivation of these operation schemas from the schema *State* is also straightforward. The identifiers of all operations defined implicitly are not included within the identifier list.

Returning to the unit *Sender*, there are two operations *CorrectAck* and *IncorrectAck* that are defined explicitly and two operations *Lookup* and *Update_curr* that are defined implicitly (the third implicit operation *Update_ack* is not needed in the following and

[4] $Init == State$

[5] We will see later that the initial data state of the sender can be constrained only in conjunction with the initial data state of the receiver, with which we can deal when having introduced aggregation in the next chapter.

hence omitted). The operations *CorrectAck* and *IncorrectAck* define the reaction to receiving a correct and incorrect acknowledgement from the receiver component, respectively. The operation schemas *Lookup* and *Update_curr*, which are in fact not visible, demonstrate the straightforward derivation of these two kinds of operation schemas from the state schema.

`BEHAVIOUR`: The base language for specifying the dynamic behaviour of specification units is the process term language of timed CSP, which we have extended in order to accommodate the needs in the context of the integrated formalism. This extended process term language is discussed in detail in Section 9.2.4.

The dynamic behaviour is defined by a process with the dedicated identifier *Behaviour*. To structure its definition, the `BEHAVIOUR` section can define further processes, to which the process *Behaviour*—directly or indirectly—refers. Process definitions in timed CSP

$ID \mathrel{\widehat{=}} ProcTerm$

have two purposes:

- to structure large process expressions and
- to express simple and mutual recursion.

We add the ability to parametrise process definitions by a list of identifiers

$ID(fp1, \ldots, fpN) \mathrel{\widehat{=}} ProcTerm$

to the syntax of timed CSP. *ProcTerm* in the above definition can contain occurrences of $fp1, \ldots, fpN$ at arbitrary places. Each reference to such a parametrised process definition must be supplied with a list of actual parameters of the same length. Formal and actual parameters are identified according to their positions in both lists.

For parametrised process definitions to be well-defined, however, we must exclude the possibility that they are involved in recursive definitions. That is, for each parametrised process definition

$P(fp1, \ldots, fpN) \mathrel{\widehat{=}} ProcTerm$

ProcTerm must not contain—directly or indirectly—references to P. Having excluded recursive definitions of parametrised processes, the meaning of referring to a parametrised process definition is given by a simple textual substitution: the reference is substituted by the right hand side of the process definition where all free occurrences of formal parameters are simultaneously substituted by the corresponding actual parameters, supplied in the reference.

As already indicated, the dynamic behaviour consists of two aspects. The external stimulus–response behaviour at the external interface defines the response to external stimuli in terms of emitting responses into the environment. The internal control behaviour, on the other hand, defines the control of the application of internal operations that have to take place in reaction to external interactions.

Let us return to the example. The `BEHAVIOUR` section of the unit *Sender* defines four processes, which are mutually recursive. The process *Receive* first awaits a new message on the channel *In*. Having received such a message it must update its current data

state. This update is defined by the notation $Update_curr(rec)$ which is an abbreviation of the event $Update_curr_X!\langle\!|\ curr? == rec\ |\!\rangle$. According to the discussion in the `OPS & PREDS` section, the operation schema *Update_curr* is defined implicitly, and it models the update of the state component *curr*. Thus, the above execution event defines the update of this state component with the received value *rec*.

The task of the process *Send* is to deliver the package consisting of the current message and acknowledgement bit to the channel *MsgSend*. Therefore, the current data state must be accessed, before its components can be delivered. The notation

$\sigma(MsgSend!(curr, ack) \rightarrow \ldots)$

combines these two steps; it is a shorthand for the process

$Lookup_X?st : [\,curr! : MSG;\ ack! : ACK\,] \rightarrow MsgSend!(st.curr!, st.ack!) \rightarrow \ldots$

which accesses the current data state in the first step and uses the variable *st*, to which the components of the current data states are bound, in the second step.

The purpose of the process *Acknowledge* is to await and process the acknowledgement of the current message, which must be re-sent if an incorrect acknowledgement has been received or if no acknowledgement has been received for *MaxDelay* time units. After an acknowledgement has been received on the channel *AckRcv*, the subsequent behaviour of the unit *Sender* depends on the relationship between the received acknowledgement bit and the current data state. This dependence on the current data state is expressed by the external choice whose two branches are guarded by the execution events of the operations *IncorrectAck* and *CorrectAck*. As stipulated in the previous chapter, the execution event of an operation is accepted by the "Z process" if, and only if, the precondition of the operation schema is satisfied with respect to the current data state and the supplied input parameters. Note further that the preconditions of the operation schemas *CorrectAck* and *IncorrectAck* partition the data state. Thus, for each combination of the current data state and received acknowledgement bit, the precondition of exactly one operation is satisfied, which resolves the external choice accordingly.

7.2 Abstract Specification Units

A template of abstract specification units is given in Figure 7.5. We have already presented an example of abstract specification units in Section 6.1, namely the requirements specification of a reliable communication medium.

We concentrate only on the differences between the structure of abstract and concrete specification units. Since it is not reasonable in the abstract model to stipulate the internal structure of an artifact, there is neither a `SUBUNITS` nor a `LOCAL` section. The purpose of the `TYPES & CONSTANTS`, the `INTERFACE` and the `ENVIRONMENTAL ASSUMPTIONS` sections is identical to that of the corresponding sections of concrete specification units.

`I/O RELATIONS`: This section serves to introduce Z schemas expressing relationships that must hold between data values communicated with external events. They are referred

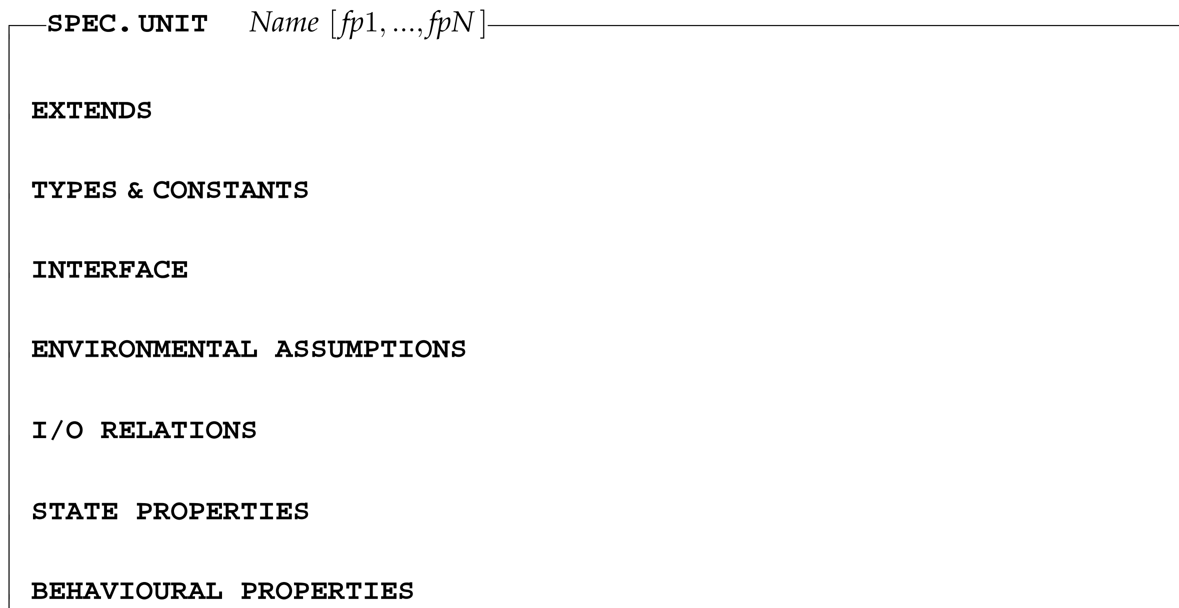


Figure 7.5: Template of abstract specification units.

to in the BEHAVIOURAL PROPERTIES section. Within such a schema reference, data values communicated with particular events are bound to the components of the referenced schema.

STATE PROPERTIES: If present, this section defines the dedicated schema *State*, which, like in the concrete model, defines the space of valid data states. At first glance, this seems to contradict the real purpose of abstract specification units. In the context of the *system* development phases, however, in which one is often faced with a physically fixed state, the model of this system state should be taken into account already in the abstract requirements specification. Based on this state model one is interested to express further properties of the state that must hold in particular situations depending on the external interaction. Additional Z schemas specified in the current section define time-variant state properties, i.e., properties that the evolution of the system state must meet in the context of particular external interactions. They are referred to in the BEHAVIOURAL PROPERTIES section, too.

BEHAVIOURAL PROPERTIES: This section is the core of abstract specification units, specifying all relevant types of system properties discussed in Section 6.2 by referring to the definitions of the previous two sections. Its purpose is to specify the properties of the dynamic behaviour of the artifact under consideration, including aspects of the temporal ordering of external interactions, real-time constraints and properties of the data values communicated with these external interactions. The language for specifying the dynamic behaviour in abstract specification units is the predicate language of

the timed failures/states model, which we introduce in Section 9.2.5. The dedicated predicate *Behaviour* defined in the BEHAVIOURAL PROPERTIES section induces a relation between timed failures $(s, \aleph)$ and timed states *tst*, corresponding to legal timed observations at the interface of the artifact at hand.

Chapter 8

Structuring Mechanisms

A basic specification unit as discussed in Chapter 7 is suited to specify a single system component with a relatively low complexity from scratch. Basic specification units have thus several shortcomings. First, they are not appropriate to specify large and complex systems; to be able to cope with such systems, we must provide adequate structuring concepts in order to stepwise reduce a complex problem to simpler problems and to compose the resulting component specifications to obtain a system specification. Second, a single specification unit is not able to model parallel activities on a common data state space. To model such parallel threads of control as well, we need a means for composing individual sequential threads of control, encapsulated within different specification units. Third, it would be useful to reuse the definition of existing specification units within different system specifications. Therefore, a mechanism is needed that allows us to build a system specification on a library of RT-Z specification units.

To overcome all three shortcomings of single specification units as just discussed, we basically provide two structuring constructs which compose specification units to more complex ones. Aggregation copes with the first two limitations and extension with the last.

Aggregation models physical or conceptual 'part-of' relationships: an aggregating unit contains several smaller or simpler units, called aggregated units. The data state and the dynamic behaviour of an aggregating unit is the (individually defined) composition of the data states and dynamic behaviour patterns of its aggregated units, respectively. Any specification unit within a hierarchy that is induced by the aggregation relationship encapsulates the structure and behaviour of an entire system, of a subsystem or of a basic system component, depending on its specific position within the hierarchy. The process term language of timed CSP is used by an aggregating specification unit to fix the configuration of its aggregated units including their interaction via particular communication channels. Since each aggregated unit encapsulates its separate data state, composing two aggregated units in parallel allows us to model parallel threads of control that work on disjoint parts of the overall, composed data state. This issue of parallelism is dealt with in the next section.

Extension, on the other hand, is a concept that supports reuse. It is orthogonal to aggregation: it allows any specification unit within an aggregation hierarchy to be defined as an extension of previously defined specification units, for instance, constituents of a specification library.

Extension can be regarded as an import mechanism for general-purpose definitions, but also as a means for incremental specification.

In Chapter 9, we define the formal meaning of single specification units. In this chapter, we informally explain the relationships between aggregating and aggregated specification units on the one hand and between extending and extended specification units on the other hand. To formally define the meaning of aggregation and extension, we develop transformation rules that reduce a hierarchy, constituted by the aggregation or extension relationship, to a single specification unit; and we identify the meaning of a hierarchy of specification units with the meaning of the single specification unit resulting from the process of reduction. Both transformations—concerning aggregation and extension—do not mean that we generally drop the idea of decomposing a specification; they have only the purpose to define the semantics. This "transformational approach" to defining the semantics of structured specifications is not uncommon: the semantics of Object-Z [Smith, 2000], e.g., is defined in terms of transformation rules from Object-Z to (plain) Z syntax.

This process of reducing a hierarchy of specification units into a single one is a recursive process from the leaves of the hierarchy towards the root. Its elementary step is to reduce an aggregating (extending) and all its aggregated (extended) specification units to a single specification unit. This elementary step is part of the subject of the subsequent discussion.

8.1 Aggregation

Aggregation, as just indicated, is a structuring mechanism that defines the internal—physical or conceptual—structure of the artifact under consideration. It is therefore applicable only in the concrete model, i.e., to concrete specification units. The only mechanism to structure abstract specification units is extension.

8.1.1 Example

To illustrate the aggregation mechanism and the consequences of applying it, we consider a simple example: fragments of the requirements specification of a ticket sale system, which is shown in Figure 8.1. The sale of tickets might concern, e.g., a particular performance with a restricted capacity. We suppose that the ticket sale is distributed to two offices, both specified by the specification unit *TicketOffice*. The overall sale system, *TicketSale*, is the aggregation of the two offices.

The data state of each office is given by the number of tickets sold so far, where each office has a fixed maximum allocation of cards (*Alloc*) that must not be exceeded. The behaviour of both offices is cyclic, where each cycle concerns a single card request: a request to issue a particular number of cards (*ticket_req*) is responded successfully (*ticket_out*) if the respective office is allowed to issue the requested number; otherwise the request fails (*failure*). The overall ticket sale system is the composition of the two offices. Its data state is the union of the data states of the offices, represented by the state components *officeA* and *officeB*, and its dynamic behaviour is the interleaving of the behaviour patterns of the offices.

There are two main aspects related to aggregation we would like to indicate before going into further details. The first aspect is the concurrency inherent in the ticket sale system: the

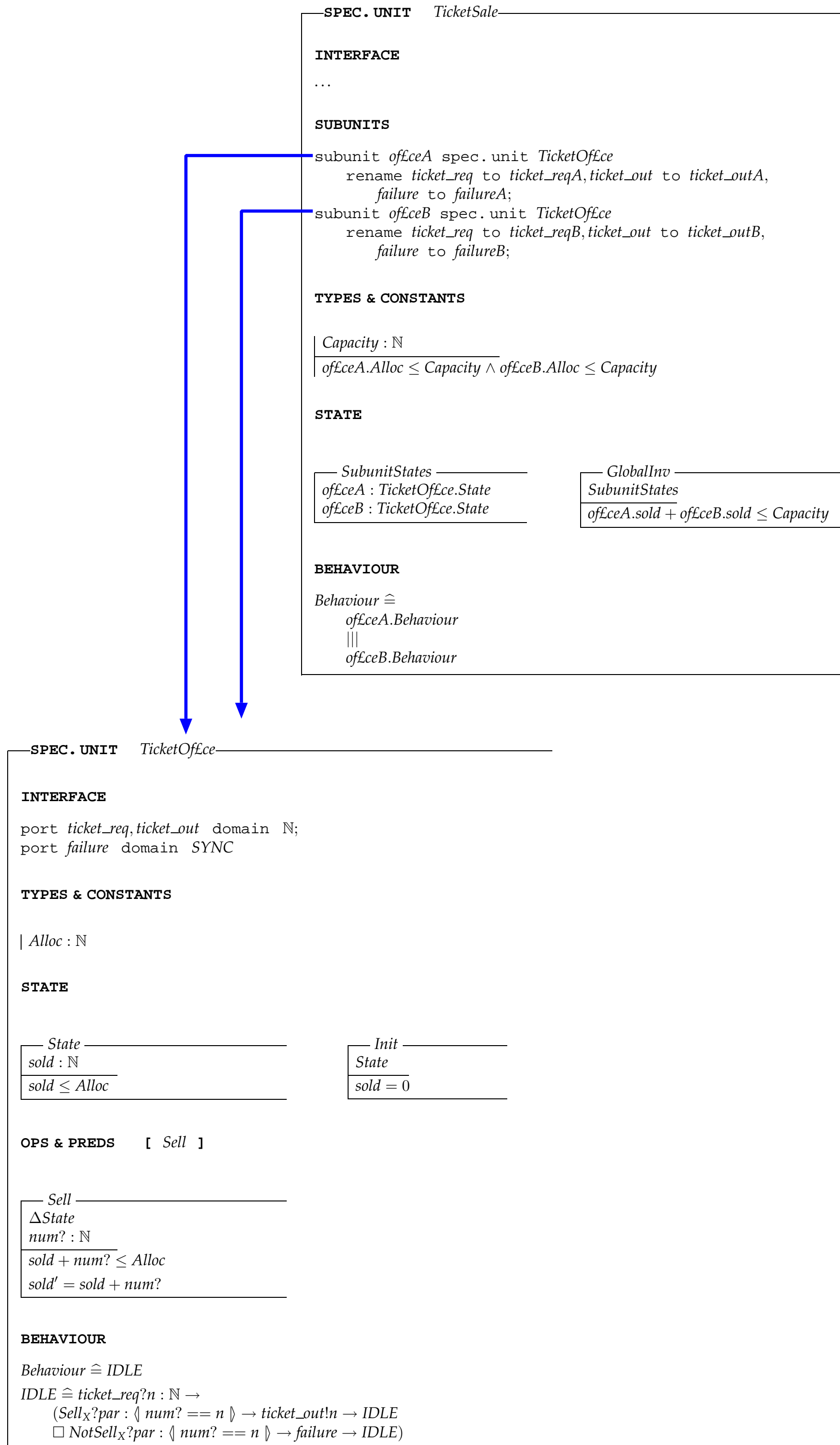


Figure 8.1: Example: ticket sale system.

aggregated offices are composed by the interleaving operator (|||) and thus they evolve (in principle) independently of each other. In particular, this means that the operations defined by the two offices may be applied concurrently. The second aspect is the concept of so-called *global invariants*. In addition to the local invariants of the data states of the offices, the schema *GlobalInv* of the ticket sale system specifies a constraint on the relationship between the state components of the two offices: both offices together must not sell more cards than the overall capacity of the performance. This global invariant has some important consequences with respect to the interaction of the aggregated offices.

Consider the data state of the overall system in which the two offices together have almost exhausted the capacity. If the offices receive requests simultaneously that together but not alone will exceed the capacity, then the considered RT-Z specification does not prevent the parallel execution of the corresponding operations. The interpretation prescribed by the RT-Z semantics is that the system will diverge in this case, i.e., it will behave in an arbitrary way. This means that an implementation of the ticket sale specification is required to respect the overall capacity unless there is a simultaneous request to the two offices whose cumulative amount exceeds the capacity. The requirements specification simply abstracts from the consideration of this specific situation. It is the task of a more concrete design specification to devise mechanisms preventing the parallel execution of two requests under such circumstances. For instance, the offices might be connected by a channel via which they could coordinate the issue of cards; or the allocations might be fixed such that $\mathit{officeA.Alloc} + \mathit{officeB.Alloc} \leq \mathit{Capacity}$ holds.

8.1.2 Global Invariants

Global invariants arise in conjunction with aggregation.[1] A global invariant is a constraint that an aggregating unit imposes on the relationship between state components of several aggregated units, see the example in the previous section. To understand the implications of global invariants, we first elaborate on the mentioned process of reducing an aggregation hierarchy of specification units, which is used to define the meaning of aggregation.

Global invariants are a concept that have a counterpart in the B language [Abrial, 1996], in which they are called glue invariants,[2] and in the formal specification language ASTRAL [Coen-Porisini et al., 1997].

Let us first consider the structure of the data state of a compound RT-Z specification. In the context of an RT-Z specification that is composed of an aggregation hierarchy of speci-

[1] We will see later that global invariants also arise in conjunction with extension.

[2] In brief, B specifications are composed of *machines*. Machines can *include* other machines. Each B machine encapsulates a data state, which is specified in a way similar to Z. Glue invariants are state invariants that refer to the state variables of several of the included machines.

Although the two concepts are very similar, the consequences they have in B and RT-Z are absolutely different. The B system model is sequential: it does not take into account the concurrent execution of operations. Thus, the interaction between operations of different machines working on different subspaces of the overall data state but all affecting the satisfaction of the glue invariant, need not be considered in B.

As we will see later in this thesis, the ability of RT-Z to relate the data states of different units through global invariants while allowing operations working on the data states of different units to be executed concurrently (under particular conditions) is a major achievement of the model underlying RT-Z, but at the same time a major source of complexity.

fication units, the overall data state of the specified system is hierarchically partitioned into subspaces according to the hierarchical decomposition of the RT-Z specification into specification units, where each specification unit defines its associated data state.

As mentioned in the beginning of this chapter, the meaning of aggregation is defined in terms of transformation rules, reducing a hierarchy of specification units to a single one. These rules are the subject of Section 8.1.4 and Section 8.1.5. During this reduction process, each operation, defined in a particular specification unit of the aggregation hierarchy, is lifted (promoted) from the data state of its associated specification unit to the overall data state of the specified system. That is, each operation, which is originally—in the context of its associated specification unit—interpreted with respect to the corresponding part of the overall data state, is lifted to a *system* operation working on the overall data state of the system. This lifting is defined such that the state transitions characterised by the resulting system operation satisfy the following conditions.

- The projection of the state transitions to the subspace of the associated specification unit coincides with the interpretation of the operation in the context of the associated specification unit.
- The projection of the state transitions to the remaining subspaces of the overall data state yields no change.

At this level of consideration, the concurrent application of system operations originating from different specification units entails no problem: since the "real" state changes they cause concern disjoint parts of the overall data state, they do not interfere each other and their state transitions can be combined in a consistent way. Things, however, become more complicated when global invariants are taken into consideration: a global invariant relates state components of different specification units, and consequently the independence of system operations originating from different specification units is no longer guaranteed. In other words, global invariants have the consequence that the application of a system operation, originally defined on the data state of its associated specification unit, may now depend on the overall data state.

Let us discuss global invariants in more detail with the help of the example depicted in Figure 8.2, in which we consider the aggregation of two components $Comp1$ and $Comp2$. $Comp1$ consists of a state variable x and an operation $Op1$ that increases x, and $Comp2$ consists of a state variable y and an operation $Op2$ that decreases y. We (alternatively) consider two global invariants defined in the aggregating unit, which relate the state variables x and y. The first one requires their equality, and the second one requires that x is less than y.

Consider the first global invariant, the equality of the state components x and y of the aggregated units. It turns out that this overly strong global invariant does not allow the operations of $Comp1$ and $Comp2$ to be lifted (promoted) in a consistent manner, because both operations insist on one state component to be changed while the lifting process insists that the other state component remains unchanged; this is an obvious contradiction to the global invariant. Thus, the lifted system operations are inconsistent which means that they define no transition on the overall data state at all. The problem caused by this global invariant is the strong coupling between the state components of the aggregated units.

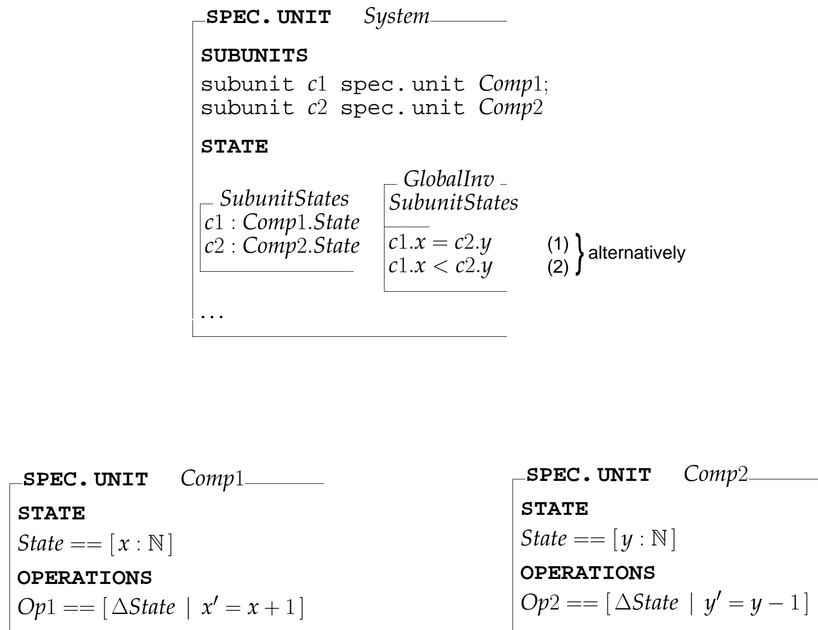


Figure 8.2: Example of global invariants.

One rationale behind specification units is "separation of concerns." Specification units should satisfy the criterion of *cohesion*: the elements of a unit should be related as close as possible, and as few as possible (globally formulated) relationships to elements of other specification units should exist. In the above example, it is questionable if the decomposition of the data state space into the units *Comp*1 and *Comp*2 is reasonable due to the strong equality relationship between x and y. Dispensing with the decomposition into the units *Comp*1 and *Comp*2 would relieve us of employing global invariants. In the case that the decomposition into specification units is used in order to fix physical aspects, e.g., distribution, (strong) global invariants are even more problematic. If, in the example, the specification units represented distributed system units, the global equality invariant would be infeasible: there is no way to guarantee that distributed system units have always the identical state of a variable.

Consider now the second global invariant. Since the state components of the specification units are related via an inequality relationship, their operations, *Op*1 and *Op*2, are more independent of each other than in the context of the first global invariant. According to the semantics of RT-Z defined in Chapter 9, the concurrent application of the two operations to a particular data state is well-defined if, and only if, the set of final data states that the relational composition of the operations produces is not empty and independent of their order. The concurrent application of operations that is not well-defined results in divergence, i.e., in an arbitrary behaviour. If *Op*1 and *Op*2 are applied concurrently to a data state satisfying $x = y - 2$, the first operation terminates in a state satisfying $x = y - 1$, in which the precondition of the other operation is not fulfilled. The set of final data states that can be produced by their relational composition is hence empty. Thus, the concurrent application of the two operations to the data state $x = y - 2$ results in divergence. The invalidation of the precondi-

tion of one operation by the termination of the other one, both changing disjoint subspaces of the overall data state, is the consequence of the global invariant: taking the global invariant into account, the preconditions of the lifted system operations do not depend only on their original part of the data state but also on the remaining part. The preconditions of the lifted operations $Op1$ and $Op2$ are both $x < y - 1$.

The approach taken in RT-Z is to allow scenarios of concurrent operation applications that are not well-defined and result in an arbitrary behaviour. It depends on the level of abstraction whether such nondeterminisms are acceptable. In an abstract requirements specification, it is frequently the deliberate intention of the specifier to abstract from the consequences of such a kind of concurrency in order to allow a design to exhibit an arbitrary behaviour. A sufficiently concrete design specification, on the other hand, must eliminate such nondeterminisms by defining measures that prevent the concurrent application of operations that interfere each other.

To conclude, a great amount of information about the potential effects of applying operations can potentially be distributed to several specification units. This distribution leads to a relatively strong coupling between specification units when global invariants are involved, making it harder to understand and review a specification, because the 'separation of concerns' principle is more or less compromised, depending on how closely components of different units are related by the used global invariants. Global invariants can lead to close relationships between operations of different specification units and can introduce a lot of implicit properties, which are not (necessarily) immediately discernible. They should hence be used very carefully. On the other hand, they are a very powerful notation that can be used to structure complex invariants more clearly. The concept of global invariants is also supported by Object-Z [Smith, 2000].

A discussion of the interaction between global invariants and the concurrent application of operations from a more semantical point of view can be found in Chapter 9.

8.1.3 Syntax

The `SUBUNITS` section of concrete specification units is used to express the aggregation relationship: it declares the system units that aggregate each instance of the specification unit under consideration. These declarations associate each aggregated system unit with a specification unit, determining its structure and behaviour.

Any aggregation hierarchy of specification units is induced by the information contained in the `SUBUNITS` section. A specification unit (and its corresponding system units) containing a `SUBUNITS` section is called *aggregating* unit, and the specification units (and its corresponding system units) recorded therein are called *aggregated* units.

The declaration

`subunit` *agg* `spec.unit` *Spec*

introduces a single system unit *agg*, whose structure and behaviour is determined by the specification unit *Spec*, as a part of each instance of the current specification unit. By declaring

`subunit` $agg(IS)$ `spec.unit` *Spec*

a collection of system units is introduced, one for each member of the index set *IS*. Their structure and behaviour is determined by *Spec*. Such an indexed collection models a number of equally structured and equally behaving system units, each of which identified by a member of *IS*, which must be a *finite* set of index values.

8.1.4 Simple Aggregation

As already indicated in the introduction to this chapter, the meaning of aggregation is defined in terms of transformation rules that reduce a hierarchy of specification units to a single one: the meaning of an aggregation hierarchy is defined to be the meaning of the single specification unit resulting from the reduction. This reduction is a recursive process from the leaves towards the root of a hierarchy. Its elementary step is to reduce an aggregating and all its aggregated specification units. We first explain this elementary step for a specification unit whose `SUBUNITS` section uses only simple aggregation. Indexed aggregation is dealt with afterwards.

This elementary step is described separately for each section of concrete specification units. We suppose in the following that the concrete specification unit *CSU* contains the `SUBUNITS` section

`subunit` agg_1 `spec.unit` $Spec_1$;
...
`subunit` agg_N `spec.unit` $Spec_N$.

The reduction process described in the following takes the aggregated units $agg_1, \dots, agg_N$ and the aggregating unit *CSU* as input and produces a single specification unit, whose meaning is the meaning of the whole aggregation.

To understand the following transformation rules, we must have in mind that the aggregating unit *CSU* can aggregate several instances of a single specification unit, i.e.,

$$Spec_i = Spec_j \quad \text{for} \quad i \neq j;\ i, j \in \{1, \dots, N\}.$$

Constants and types declared in a specification unit need not be uniquely fixed. That is, there can be different interpretations of a constant or type. Concerning the constants and types declared in $Spec_i (= Spec_j)$, both aggregated units agg_i and agg_j should have the freedom to choose different interpretations. We must hence ensure that there are two instances of each constant and type with distinct identifiers. For this reason, the reduction process prefixes each constant and type declared in $Spec_i$ by the identifier of the respective aggregated system unit, e.g.,

$$agg_i.c \quad \text{vs.} \quad agg_j.c.$$

Because of the transformation of constant and type identifiers, all Z expressions within an aggregated unit, which could refer to those identifiers, must be transformed syntactically. This syntactic transformation is denoted by the syntactic meta function `lift_simple_Z`.

The expressions of the CSP part must undergo a similar transformation. As will be discussed later, the reduction process prefixes the identifiers of internal channels of aggregated units

by the identifier of the respective aggregated unit. Thus, each CSP expression referring to an internal channel must be transformed syntactically. This syntactic transformation is denoted by the syntactic meta function `lift_simple_CSP`.

`TYPES & CONSTANTS`. A constant, declared and constrained by an axiomatic definition, needs not be uniquely fixed, because several values might satisfy the constraints associated with that constant. Different instances of a single specification unit can hence choose distinct values for that constant. Accordingly, for each axiomatic definition in the aggregated unit $agg_i(Spec_i)$

$$\begin{array}{|l} c : T \\ \hline Pred \end{array}$$

the specification unit resulting from the reduction process will contain the axiomatic definition

$$\begin{array}{|l} agg_i.c : \texttt{lift_simple_Z}(agg_i, T) \\ \hline \texttt{lift_simple_Z}(agg_i, Pred) \end{array}$$

where $\texttt{lift_simple_Z}(agg_i, Pred)$ is obtained from $Pred$ by substituting each occurrence of a constant or type, say a, by $agg_i.a$.

In Z, the declaration of a given set does not constrain the set that denotes the introduced identifier—not even the structure of the objects that are members of this set is constrained. Different instances of a single specification unit can hence choose distinct interpretations for such a given set. Accordingly, for each given set declaration in the aggregated unit $agg_i(Spec_i)$

$[S]$

the specification unit resulting from the reduction process will contain the declaration

$[agg_i.S]$.

Free type definitions

$$\begin{array}{l} T_1 ::= c_{11} \mid \ldots \mid c_{1N_1} \mid i_{11}\langle\langle S_{11}\rangle\rangle \mid \ldots \mid i_{1M_1}\langle\langle S_{1M_1}\rangle\rangle \\ \& \ldots \& \\ T_K ::= c_{K1} \mid \ldots \mid c_{KN_K} \mid i_{K1}\langle\langle S_{K1}\rangle\rangle \mid \ldots \mid i_{KM_K}\langle\langle S_{KM_K}\rangle\rangle \end{array}$$

are expanded during the reduction process into axiomatic definitions and declarations of given sets corresponding to the semantic transformation rules defined in the Z Standard [ISO, 2002, Chapter 14], see also [Woodcock and Davies, 1996, p. 139].

$$
\begin{array}{|l}
c_{11}, \ldots, c_{1N_1} : T_1 \\
\ldots \\
c_{K1}, \ldots, c_{KN_K} : T_K \\
i_{11} : S_{11} \rightarrowtail T_1;\ \ldots;\ i_{1M_1} : S_{1M_1} \rightarrowtail T_1 \\
\ldots \\
i_{K1} : S_{K1} \rightarrowtail T_K;\ \ldots;\ i_{KM_K} : S_{KM_K} \rightarrowtail T_K \\
\hline
\langle \{c_{11}\}, \ldots, \{c_{1N_1}\}, \mathrm{ran}\, i_{11}, \ldots, \mathrm{ran}\, i_{1M_1} \rangle \text{ partition } T_1 \\
\ldots \\
\langle \{c_{K1}\}, \ldots, \{c_{KN_K}\}, \mathrm{ran}\, i_{K1}, \ldots, \mathrm{ran}\, i_{KM_K} \rangle \text{ partition } T_K \\
\forall S_1 : \mathbb{P}\, T_1;\ \ldots;\ S_K : \mathbb{P}\, T_K \mid \\
\quad \{c_{11}, \ldots, c_{1N_1}\} \cup i_{11} (\!| S_{11}[S_1, \ldots, S_K / T_1, \ldots, T_K] |\!) \cup \ldots \cup \\
\qquad\qquad i_{1M_1} (\!| S_{1M_1}[S_1, \ldots, S_K / T_1, \ldots, T_K] |\!) \subseteq S_1 \\
\quad \land \ldots \land \\
\quad \{c_{K1}, \ldots, c_{KN_K}\} \cup i_{K1} (\!| S_{K1}[S_1, \ldots, S_K / T_1, \ldots, T_K] |\!) \cup \ldots \cup \\
\qquad\qquad i_{KM_K} (\!| S_{KM_K}[S_1, \ldots, S_K / T_1, \ldots, T_K] |\!) \subseteq S_K \bullet \\
\quad S_1 = T_1 \land \ldots \land S_K = T_K
\end{array}
$$

Thus, when an aggregated specification unit contains a free type definition, it is transformed into the corresponding given set declaration and axiomatic definitions, which are in turn subject to the corresponding transformation rules already explained.

Analogously, each definition by syntactic equivalence

$S == Expr$

is expanded during the reduction process into an axiomatic definition corresponding to the syntactic transformation rule defined in the Z Standard [ISO, 2002, Chapter 12].

$S : \{Expr\}$

To summarise, the specification unit resulting from the reduction process contains all types and constants of the aggregating unit *CSU* and all types and constants of the aggregated units $Spec_1, \ldots, Spec_N$, transformed as just described.

`STATE`. We have already outlined the structure of the data state in connection with aggregation: the data state space of an aggregating unit encompasses the data state spaces of the aggregated units, including their local invariants. An aggregating unit does not define its individual data state space. That is, a unit containing a `SUBUNITS` section must not define a *State* schema, because it is automatically defined as a result of the considered reduction process.

$$
\begin{array}{|l}
\textbf{— } State \textbf{ —} \\
SubunitStates \\
\hline
GlobalInv
\end{array}
$$

The data state spaces of the aggregated units are introduced by *SubunitStates*, which is itself a schema derived by the reduction process as follows.

SubunitStates

$agg_1 : \texttt{lift_simple_Z}(agg_1, Spec_1.State)$
...
$agg_N : \texttt{lift_simple_Z}(agg_N, Spec_N.State)$

For each aggregated system unit recorded in the SUBUNITS section, an equally named state variable is declared whose type is the schema type of the state schema of the associated specification unit.[3] Note that this schema can be immediately derived from the SUBUNITS section of the aggregating unit.

Global invariants, constraining the relationships between state components of different aggregated units, are defined in the dedicated schema *GlobalInv* in the STATE section of an aggregating unit. The concept of global invariants has been treated in Section 8.1.2.

The schema

Init

SubunitInit

GlobalInit

defines the initialisation of the overall data state space of the aggregation. The initialisation of the subspaces of the aggregated system units is defined by the schema

SubunitInit

State

$agg_1 \in \texttt{lift_simple_Z}(agg_1, Spec_1.Init)$
...
$agg_N \in \texttt{lift_simple_Z}(agg_N, Spec_N.Init)$

which is derived as another result of the reduction process. Each aggregated system unit must be in a state as defined by the *Init* schema of the corresponding aggregated specification unit.

Additional constraints on the initialisation of the overall data state space can be stipulated in the dedicated schema *GlobalInit*. This concerns constraints on the relationship between state components of different aggregated units, which must be met in addition to the local initialisation constraints.

OPS & PREDS. As already outlined with regard to a whole aggregation hierarchy, the operations (and predicates) of the aggregated units $agg_1, \ldots, agg_N$ are originally defined on the data state space of the corresponding specification unit, and they are lifted to the overall data state space during the reduction process. This lifting is achieved by applying the Z concept of *promotion*.

As a further part of the definition of the reduction process, the promotion schema Φagg_i is derived for each aggregated system unit agg_i recorded in the SUBUNITS section as follows.

---

[3] We use schema variables denoting states of aggregated system units rather than simply importing the corresponding state schemas, because several aggregated system units may be associated with a single specification unit.

$agg_i State == \texttt{lift_simple_Z}(agg_i, Spec_i.State)$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\Phi agg_i} \\
\hline
\Delta State \\
\Delta agg_i State \\
\hline
\theta agg_i State = agg_i \\
agg_i' = \theta agg_i State' \\
agg_1' = agg_1 \land \ldots \land agg_{i-1}' = agg_{i-1} \\
agg_{i+1}' = agg_{i+1} \land \ldots \land agg_N' = agg_N \\
\hline
\end{array}
$$

This promotion schema serves to lift the operations defined on the data state space of the aggregated system unit agg_i ($\Delta agg_i State$) to the overall data state space ($\Delta State$). It stipulates that the subspaces of the other aggregated system units are left unchanged, while the transformation within the subspace of the aggregated system unit agg_i is in accordance with the operation to be promoted.

Based on this preliminary definition, the single specification unit resulting from the reduction process contains the promoted operation schema

$agg_i.Op == (\Phi agg_i \land \texttt{lift_simple_Z}(agg_i, Spec_i.Op)) \setminus \Delta agg_i State$

for each operation schema *Op* contained in the identifier list of the aggregated specification unit $Spec_i$. The operation schema $Spec_i.Op$ can contain references to other operation schemas of the considered unit, which are themselves lifted during the reduction process. Accordingly, the meta function `lift_simple_Z` substitutes each reference to an operation schema *ref* by the new identifier $agg_i.ref$, which the referenced operation schema will have after the lifting.

This syntactical transformation has essential consequences. Using this approach to deal with references to operation schemas, each operation schema is lifted in accordance with the lifting of the context into which it is embedded.

An alternative approach would be to expand references to operation schemas immediately. However, this would not allow us to refer to properties of operation schemas that they have in the context (of the overall data state) into which they are embedded during the reduction process, as will become apparent by revisiting the ticket selling example of p. 67. In the aggregated unit *TicketOffice*, the process term *Behaviour* refers to the operation schema *NotSell*, which is defined implicitly as follows.

$NotSell == \neg \mathsf{pre}\, Sell \land \Xi State$

Following the alternative approach (of immediate reference expansion), the reference $\neg \mathsf{pre}\, Sell$ would be expanded immediately in the local context of the unit *TicketOffice* and its data state: *NotSell* would have as its precondition the negation of the precondition of *Sell*, computed within the local context of *TicketOffice*, namely that the local capacity *Alloc* must not be exceeded. After the reduction process, however, both lifted operation schemas would not have the intended properties in the global context. For example, the property that the preconditions of the two operations partition the data state space. This property is not satisfied because of the additional global invariant,

which affects the precondition of the lifted operation *TicketOffice.Sell*: an additional conjunct of the precondition is that the total capacity must not be exceeded. Since the reference $\neg$ pre *Sell* of the definition of *NotSell* is already expanded in the local context, the precondition of *TicketOffice.NotSell* does not incorporate this case of the invalidation of the additional global invariant. Thus, those data states in which the application of *Sell* would exceed the total but not the local capacity would not be covered by the preconditions of the lifted operations and hence the process *Behaviour* would deadlock in these data states.

Following the first approach (of delayed reference expansion), on the other hand, the reference $\neg$ pre *Sell* is expanded only in the global context, so the lifted operation schema *NotSell* does take the invalidation of the global invariant into account as intended. Note that all identifiers of local entities of aggregated specification units (such as operations and predicates) are prefixed during the reduction process by the respective unit identifier. This is elaborated in the discussion of the `INTERFACE` and the `LOCAL` section.

Note also that we lift only operation schemas contained in the identifier list of the `OPS & PREDS` section. Auxiliary schemas need not be lifted because they are eliminated by means of normalisation.

Since we treat predicates as a specific kind of operations, there is no need to distinguish between operation schemas defining operations and those defining predicates.

`INTERFACE`. The interface of the specification unit resulting from the reduction process is identical with the interface of the aggregating specification unit *CSU*. Declarations of the `INTERFACE` sections of the aggregated specification units must be contained either in the `INTERFACE` or the `LOCAL` section of the aggregating specification unit. This depends on whether or not these interface elements are hidden in the behaviour definition of the aggregating specification unit.

The identifiers of external ports (in contrast to the identifiers of internal channels) of the aggregated units are not prefixed by the respective unit identifier. Therefore, by default, equally named external ports of different aggregated units are identified during the reduction process. When this is not intended, they can be appropriately renamed by using the renaming operator defined for specification units, see Section 8.3. We have already used this renaming operator in the example on p. 67.

`LOCAL`. All internal channels declared by the aggregating unit *CSU* are part of the `LOCAL` section of the specification unit resulting from the reduction process. Moreover, for each declaration

`channel` *chan* `domain` *Set*

in the `LOCAL` section of the aggregated unit $Spec_i$, the declaration

`channel` $agg_i.chan$ `domain` *Set*

is part of the `LOCAL` section of the resulting unit. That is, the identifiers of internal channels of the aggregated units are prefixed by the identifier of the respective aggregated system unit. This allows us to distinguish equally named internal channels of different aggregated system units (e.g., operation-related events).

ENVIRONMENTAL ASSUMPTIONS. The environmental assumption of the specification unit resulting from the reduction process is the environmental assumption of the aggregating specification unit *CSU*. The environmental assumptions of the aggregated units must be ensured by the aggregation context, which is a verification condition that should be checked for each aggregation.

BEHAVIOUR. Finally, we discuss how the definition of the behaviour patterns of the aggregating and of the aggregated specification units are composed to obtain the behaviour definition of the specification unit resulting from the reduction process. The behaviour definition of the aggregating unit *CSU* (process term *Behaviour*) refers to the behaviour of the aggregated unit agg_i by means of $agg_i.Behaviour$. This reference denotes the application of the syntactic meta function lift_simple_CSP to the behaviour definition of the specification unit $Spec_i$.

$$agg_i.Behaviour \mathrel{\widehat{=}} \texttt{lift_simple_CSP}(agg_i, Spec_i.Behaviour)$$

This syntactic transformation concerns the renaming of internal channels, discussed with respect to the LOCAL section.

The process *Behaviour* of the aggregating unit *CSU* can compose the behaviour patterns of the aggregated units by arbitrary process operators. The combination of units by the parallel operators enables the separation of concurrent behaviour patterns working on disjoint parts of the overall data state within different system units. Hence, while a basic specification unit can define only sequential state-dependent behaviour patterns, aggregation makes it possible to consider concurrency (and distribution). Thereby, the process term *Behaviour* of an aggregating unit characterises the (physical or conceptual) configuration of the aggregated units to constitute the whole aggregation.

8.1.5 Indexed Aggregation

In this section, we describe the reduction process for a specification unit whose SUBUNITS section contains indexed aggregation. We describe only the differences to the reduction process for simple aggregation. When a specification unit contains simple and indexed aggregation, the described transformations are combined in the obvious way.

We assume that the considered specification unit, say *CSU*, contains (among others) the declaration

subunit *agg*(*IS*) spec. unit *Spec*

in its SUBUNITS section where *IS* is a finite set of index values. Again, the reduction process is described separately for each section of concrete specification units.

As we will shortly argue in the description of the transformation rules, each constant $c : T$ declared in the aggregated unit *Spec* must be lifted to the function $agg.c : IS \rightarrow T$, because we have to take into account that each instance of the indexed aggregation chooses a different value for that constant. Consequently, each expression of the Z part must be transformed syntactically: each occurrence of a constant c introduced by the aggregated unit must be substituted by the function application $(agg.c\ index)$, where *index* is a dedicated identifier that is assumed not to occur in the aggregated unit; it identifies a particular instance of the

indexed aggregation. The syntactical transformation of Z expressions is represented by the syntactic meta function `lift_indexed_Z`.

A similar syntactical transformation must be applied to the expressions of the CSP part of the aggregated unit. Since the interaction of different instances of the indexed aggregation via external and internal channels must be distinguished, the values transmitted via these channels by different instances must be distinguishable. Basically, this is achieved by extending each communicated value with the identifier of the involved instance. The corresponding syntactic transformation working on process terms is represented by the syntactic meta function `lift_indexed_CSP`.

TYPES & CONSTANTS. A constant, declared and constrained by an axiomatic definition, needs not be uniquely fixed. When building an indexed collection of a specification unit, we have to take into account that each instance of such a collection can choose its own value for such a constant. This means that the reduction process must lift each constant to a function mapping each instance to a particular value.

Accordingly, for each axiomatic definition in the aggregated specification unit *Spec*

$$\begin{array}{|l} c : T \\ \hline Pred \end{array}$$

the specification unit resulting from the reduction process will contain the definition

$$\begin{array}{|l} agg.c : IS \rightarrow \texttt{lift_indexed_Z}(agg, T) \\ \hline \forall\, index : IS \bullet \texttt{lift_indexed_Z}(agg, Pred) \end{array}$$

where $\texttt{lift_indexed_Z}(agg, Pred)$ is obtained from *Pred* by substituting each occurrence of a constant, say *a*, by the function application $(agg.a\ index)$.

Analogously to the procedure described in the context of simple aggregation, the reduction process expands each definition by syntactic equivalence

$S == Expr$

into the axiomatic definition

$| \; S : \{Expr\}$

which in turn is subject to the transformation rule just described.

Specification units instantiated by means of indexed aggregation must not contain type declarations (given sets or free types). The technical reason for this decision is that the Z notation does not allow us to lift type declarations in accordance with our approach to lifting constants. However, also from a conceptual point of view it would not be

reasonable if different instances of an indexed collection had different interpretations of a type, which would be the consequence of lifting the types.[4]

`STATE`. The state schema of the aggregated unit may depend on any constant, say $c : T$, declared by this unit. As discussed, each such constant is lifted to the function $agg.c : IS \rightarrow T$. Therefore, the state schema must be lifted, too. Each occurrence of a constant, say c, is transformed into the function application $(agg.c\ index)$, where $index$ is an additional state component characterising the index by which the considered instance of the indexed collection is identified.

$$aggState == \texttt{lift_indexed_Z}(agg, Spec.State) \wedge [\, index : IS \,]$$

The data state space of the specification unit resulting from the reduction process is constituted by the data states of all instances of *agg* indexed by members of *IS*. Hence, the derived schema *SubunitStates* contains a function mapping each identifier in *IS* to a data state in *aggState*.

SubunitStates

$agg : IS \rightarrow aggState$
$\ldots$

$\forall\, id : IS \bullet (agg\ id) \in [\, aggState \mid index = id \,]$
$\ldots$

Each occurrence of a constant in the initialisation schema of the aggregated specification unit is lifted analogously.

$$aggInit == \texttt{lift_indexed_Z}(agg, Spec.Init) \wedge [\, index : IS \,]$$

After initialisation, each instance of the indexed collection must be in an initial state as defined by the *Init* schema of the aggregated unit. The derived schema *SubunitInit* thus contains the following definitions.

SubunitInit

$State$

$\forall\, id : IS \bullet (agg\ id) \in [\, aggInit \mid index = id \,]$
$\ldots$

`OPS & PREDS`. Similar to the case of simple aggregation, a promotion schema is defined, which lifts each operation schema defined on the individual data state space of a single instance of the indexed collection to the overall data state space.

[4] Simply preventing the lifting of types, however, would lead to an inconsistency, because free types are defined in terms of the introduction of constants and injective functions. Thus, the declaration of free types would indirectly imply the lifting of the corresponding constants and injections, which would in turn imply that different instances of an indexed aggregation would have different interpretations of the free type.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\Phi agg} \\
\Delta State \\
\Delta aggState \\
id? : IS \\
\hline
\theta aggState = (agg\ id?) \\
(agg'\ id?) = \theta aggState' \\
index' = index = id? \\
\forall\, id : IS \mid id \neq id? \bullet (agg'\ id) = (agg\ id) \\
\hline
\end{array}
$$

Then, the specification unit resulting from the reduction process contains the lifted operation

$agg.Op == (\Phi agg \wedge \texttt{lift_indexed_Z}(agg, Spec.Op)) \setminus \Delta aggState,$

for each operation schema *Op* in the aggregated specification unit *Spec*. The identifier *id*? of the involved instance is an additional input parameter.

`INTERFACE`. The interface of the resulting specification unit is identical with the interface of the aggregating unit. Thus, for each declaration

`port` *chan* `domain` *Set*

in the `INTERFACE` section of the aggregated unit *Spec*, either the declaration

`port` *chan* `domain` $IS \times Set$

must be part of the `INTERFACE` section or the declaration

`channel` *chan* `domain` $IS \times Set$

must be part of the `LOCAL` section of the aggregating unit.

The extension of the domains with the set *IS* reflects the remark made in the beginning of this section: the particular instance of the indexed aggregation responsible for an interaction must be identified by the transmitted value.

`LOCAL`. For each declaration

`channel` *chan* `domain` *Set*

in the `LOCAL` section of the aggregated unit *Spec*, the declaration

`channel` *agg.chan* `domain` $IS \times Set$

is part of the `LOCAL` section of the specification unit resulting from the reduction process. Note that, in contrast to external ports, the names of internal channels are prefixed by the respective unit identifier.

`BEHAVIOUR`. The behaviour definition of the aggregating unit (process term *Behaviour*) refers to the behaviour of a particular instance of the aggregated collection with identifier $id \in IS$ by means of $agg.Behaviour(id)$, e.g.,

$$Behaviour \mathrel{\widehat=} \mathop{|||}_{id \in IS} agg.Behaviour(id).$$

The notation $agg.Behaviour(id)$ abbreviates the application of the syntactic meta function `lift_indexed_CSP` to the behaviour definition of the aggregated unit *Spec*:

$$agg.Behaviour(id) \mathrel{\widehat=} \texttt{lift_indexed_CSP}(agg, Spec.Behaviour)[id/index].$$

As discussed, the dedicated identifier *index*, which is substituted by the value *id*, represents the index by which an instance of the indexed collection is identified.

The meta function `lift_indexed_CSP` causes the following syntactic transformations.

First, each set or value expression (of an instance of the channel input or output operator) must be adapted in order to be consistent with the change of the domains associated with external and internal channels. Each expression

$$chan?x : S$$

is transformed into

$$chan?x : \{index\} \times S \qquad \text{or} \qquad agg.chan?x : \{index\} \times S$$

depending on whether *chan* is an external or internal channel. Further, each expression

$$chan!v$$

is transformed into

$$chan!(index, v) \qquad \text{or} \qquad agg.chan!(index, v)$$

again depending on whether *chan* is an external or internal channel.

For operation-related events, in contrast, the transformation is slightly different, because the identifier of the involved instance is transmitted as an input parameter of the communicated schema binding. Each expression

$$Op_X?par : S$$

is transformed into

$$agg.Op_X?par : (S \land [\, id? : IS \mid id? = index \,])$$

where $id?$ is the input parameter used in the promotion schema Φagg.[5] Thus, each binding of input and output parameters is extended with the pair $id? \mapsto index$, taking into account the additional parameter introduced by the lifting process.[6]

This completes our description of the transformation rules for simple and indexed aggregation. In this way, we have defined the meaning of aggregation in a semi-formal yet systematic manner.

[5] We assume that the set of bindings is expressed in terms of a schema expression.

[6] The transformation of invocation and termination events (double-event approach) is defined analogously.

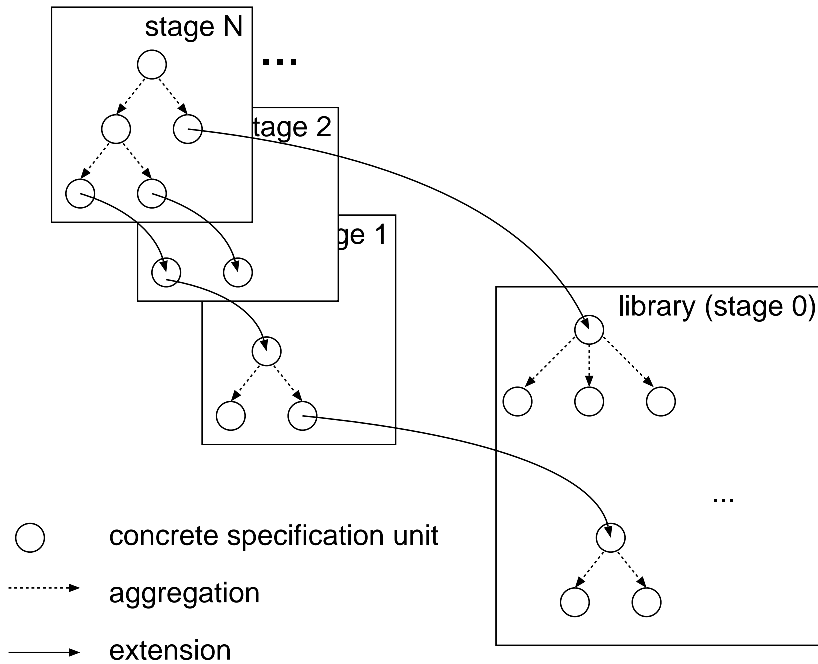


Figure 8.3: Structure of concrete specifications in RT-Z.

8.2 Extension

Extension is a structuring mechanism that is orthogonal to aggregation: any specification unit within an aggregation hierarchy can extend other specification units outside this aggregation hierarchy. Informally speaking, an extending specification unit imports the definitions of the specification units it extends and is able to add new entities, to add new constraints on imported entities or to redefine imported entities.[7] Extension is intended to support the reuse of "libraries of specification units" and an incremental way of specification. It is in some sense comparable with inheritance in object-oriented approaches, but more restricted. In this section, we explain the use of the extension mechanism and its meaning.

Extension applies to concrete and abstract specification units. We first deal with the extension of concrete specification units.

Figure 8.3 outlines the structure of concrete specifications in RT-Z. From a structural point of view, a concrete specification is a finite directed graph whose nodes are constituted by concrete specification units and whose edges represent aggregation and extension relationships. Aggregation is depicted by dotted arrows, and extension is depicted by solid arrows. Such a graph must have the property that it can be partitioned into a set of trees, each of which connected solely by aggregation relationships: these trees are what we have called up to now aggregation hierarchies. Nodes of different trees can be connected by extension relationships. Finally, the graph must be acyclic, i.e., there must be no path from a specification unit to itself via aggregation and/or extension edges. This latter property is vital for the

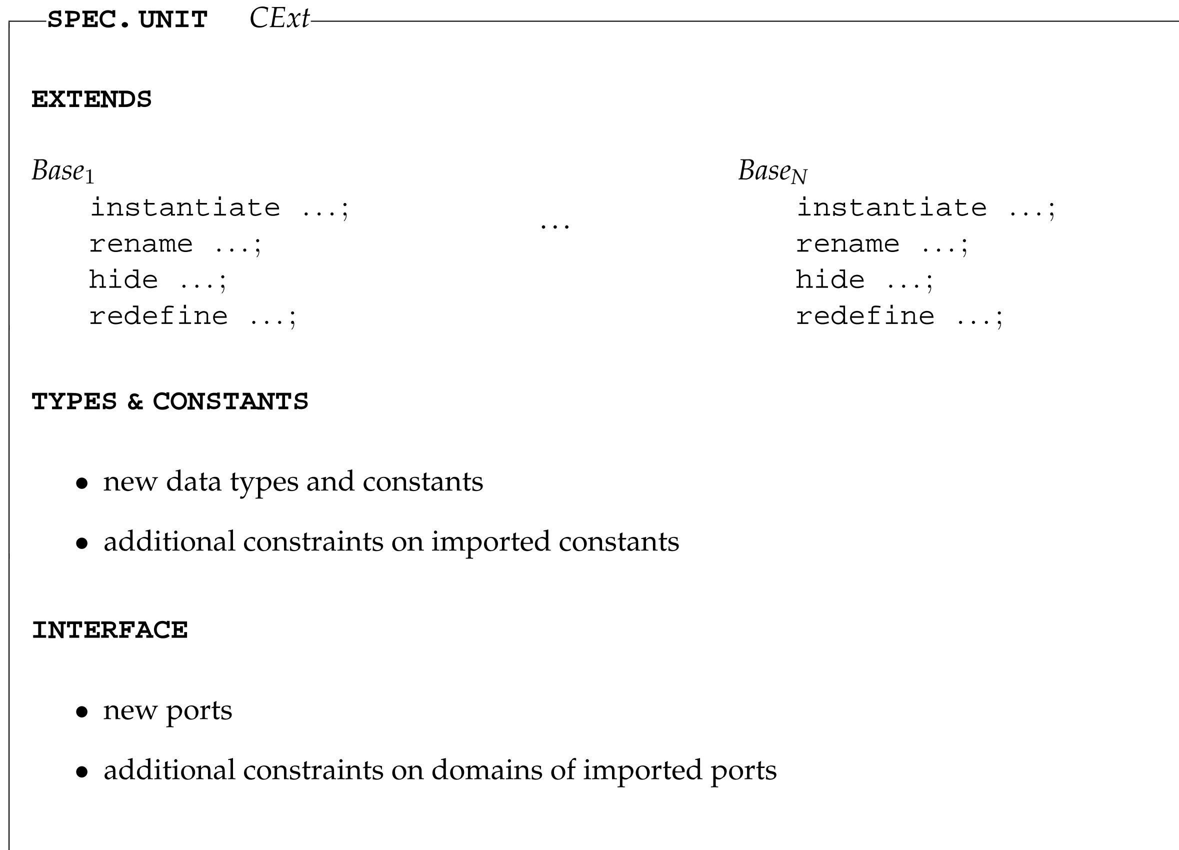
[7] The term 'entity' encompasses ports, channels, processes, constants, operations, etc.

definition of the meaning of the structuring mechanisms.

From a more conceptual point of view, the structure of a graph forming a concrete specification is interpreted as follows. The core of a concrete specification is a single aggregation hierarchy (tree) of specification units. In the figure, it is enclosed by the rectangle *stage* N. This aggregation hierarchy defines the structure and behaviour of the system to be designed. Using extension relationships to connect this core aggregation hierarchy with other ones, the concrete specification is further structured along a chronological dimension: the ultimate result of the design process need not be set up in a single step, but can be defined in several stages by means of stepwise extension; this allows us to pursue an incremental way of specification. Each stage $1, \ldots, N$ of the design process is depicted in the figure by a rectangle. Specification units of a particular stage can extend specification units of a lower stage.

The rectangle *stage* 0 has a special meaning: it represents a library of specification units, which can be used to collect general-purpose, frequently used specification units that can be instantiated during a variety of system development tasks. This library can be regarded as the stage preceding *stage* 1 of each design process.

Let us consider the schematic definition of the concrete specification unit *CExt* as an example for the following discussion.


SPEC. UNIT *CExt*

EXTENDS

Base$_1$
```
instantiate ...;
rename ...;
hide ...;
redefine ...;
```
...

Base$_N$
```
instantiate ...;
rename ...;
hide ...;
redefine ...;
```

TYPES & CONSTANTS

- new data types and constants
- additional constraints on imported constants

INTERFACE

- new ports
- additional constraints on domains of imported ports

ENVIRONMENTAL ASSUMPTIONS

- environmental assumptions concerning new ports
- additional environmental assumptions concerning imported ports (EA_{Ext}).

SUBUNITS

- new subunits

STATE

- new state components
  $State_{Ext} == \ldots$ $Init_{Ext} == \ldots$
- additional global invariants
  $GlobalInv == \ldots$ $GlobalInit == \ldots$

OPS & PREDS [ ]

- new operation schemas
  $Op == [\Delta State; \ \ldots \mid \ldots]$
- redefinition of imported operation schemas (Op is member of `redefine` list)
  $Op == [\Delta State; \ \ldots \mid \ldots]$
- extension of imported operation schemas
  $Op_{Ext} == [\Delta State; \ \ldots \mid \ldots]$

LOCAL

- new channels

BEHAVIOUR

- redefinition
  $Behaviour \mathrel{\widehat{=}} \ldots$ references to processes defined in $Base_1, \ldots, Base_N \ldots$
- extension
  $Behaviour_{Ext} \mathrel{\widehat{=}} \ldots$

Analogously to aggregation, the meaning of extension is defined in terms of transformation rules that reduce the definitions of the extending (*CExt*) and of the extended ($Base_1, \ldots, Base_N$) specification units to a single specification unit. Again, we discuss this reduction process separately for each section.

To understand the following transformation rules related to extension, we must emphasise one important difference between aggregation and extension. Aggregation is a structuring mechanism on the level of specification unit *instances*, i.e., it defines the structure of the instances of specification units. An instance of a specification unit can aggregate instances of other specification units. In particular, it can aggregate several instances of a single specification unit. Extension, in contrast, is a structuring mechanism on the level of specification units, i.e., it does not fix the structure of the corresponding instances. A specification unit can extend another specification unit, but an instance of a specification unit cannot extend an instance of another specification unit. This has important consequences on the following transformation rules: since extension refers to specification units directly, these rules need not lift definitions of the extended specification units like the transformation rules for aggregation.

`TYPES & CONSTANTS`. The extending specification unit *CExt* is able

- to introduce new constants and data types and
- to add new constraints on those constants and data types imported by the extended units $Base_1, \ldots, Base_N$.

The specification unit resulting from the reduction process will contain all constants and data types defined by the extending and the extended specification units. We discuss in the following how the declarations of and constraints on a single constant or data type, which may be distributed to several of the considered specification units, are composed in the specification unit resulting from the reduction process.

First, several declarations of a single given set

$[T]$

in the considered specification units are simply identified with each other and hence result in a single declaration.

Second, axiomatic definitions of a single constant in several specification units

$$\begin{array}{|l} c : S1 \\ \hline Pred1 \end{array} \quad \ldots \quad \begin{array}{|l} c : SK \\ \hline PredK \end{array}$$

are transformed into

$$\begin{array}{|l} c : S1 \cap \ldots \cap SK \\ \hline Pred1 \wedge \ldots \wedge PredK \end{array}$$

That is, the constraints on the constant c contained in different specification units are conjoined. For this transformation to yield a well-typed result, the set expressions $S1, \ldots, SK$ must have the same type.

Third, the reduction process expands free type definitions

$T ::= c1 \mid \ldots \mid cK \mid i1\langle\langle S1\rangle\rangle \mid \ldots \mid iM\langle\langle SM\rangle\rangle$

into axiomatic definitions and declarations of given sets corresponding to the approach described in Section 8.1.4 in the context of aggregation. Thus, when an extended or the extending specification unit contains a free type definition, it is transformed into the corresponding given set declaration and axiomatic definitions, which are in turn subject to the corresponding composition rules already explained.

Analogously, each definition by syntactic equivalence

$S == Expr$

is expanded into the axiomatic definition

$| \; S : \{Expr\}$

which is semantically equivalent.

`INTERFACE`. The extending specification unit *CExt* is able

- to introduce new ports and
- to further restrict the domains of imported ports.

The specification unit resulting from the reduction process will contain all ports declared by the extending and extended specification units. Declarations of a single port in several of the considered specification units

`port` *chan* `domain` $S1$ $\ldots$ `port` *chan* `domain` SK

are transformed into

`port` *chan* `domain` $S1 \cap \ldots \cap SK$

That is, the derived domain is the intersection of the domains associated with *chan*. Again, for this transformation to yield a well-typed result, the set expressions $S1, \ldots, SK$ must have the same type.

`ENVIRONMENTAL ASSUMPTIONS`. The extending specification unit *CExt* can define the predicate EA_{Ext}, which is, by convention, interpreted as defining additional environmental assumptions. This predicate is able to refer to the interface ports of the extending *and* of the extended specification units and is therefore able to specify assumptions relating the interface ports of all considered specification units.

$$EA \mathrel{\widehat{=}} (\bigwedge_{j \in \{1,\ldots,N\}} Base_j.EA) \wedge EA_{Ext}$$

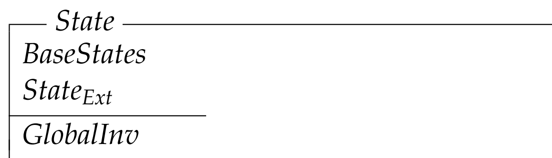
The environmental assumption resulting from the reduction process is the conjunction of the environmental assumptions of the extending and the extended units. That is, one is not free in composing the environmental assumptions of the extended units. This reflects the nature of extension where existing definitions should not be ignored but only be augmented in a consistent way.

SUBUNITS. The extending specification unit *CExt* is able to introduce new aggregated subunits in addition to those introduced by the extended specification units.

Declarations of a single subunit in several of the considered specification units are identified with each other, so there must not be any difference between these declarations (concerning the instantiations, the renamed and hidden ports).[8]

STATE. The resulting data state space encompasses the data states of the extended units. Optionally, the extending specification unit *CExt* can define its individual data state, introducing new state components in addition to the state components of the extended specification units. By convention, the individual data state, if any, is defined by the schema $State_{Ext}$.

The specification unit resulting from the reduction process contains the state schema



where the subspaces of the extended units are introduced by *BaseStates*, which is derived as follows.[9]

Global invariants, constraining the relationships between state components of several extended units and the extending unit, can be defined by the dedicated schema *GlobalInv* in the STATE section of the extending specification unit.

Also the definition of the overall data state space reflects the idea of extension: the state invariants of the extended specification units cannot be ignored but only be extended.

The initialisation of the overall data state space is derived as follows.

The extending specification unit can define the schemas $Init_{Ext}$ and *GlobalInit* to stipulate relationships between the state components of the extending and the extended specification units that must be satisfied by the initial data state.

OPS & PREDS. There are three alternatives for the extending specification unit *CExt* for defining operations and predicates:

[8] The only case where this identification of multiply declared subunits should occur is when there are different paths from an extending to an extended specification unit that declares a subunit.

[9] State components of different extended units with identical name are identified and must thus have compatible types.

Introduction of new operation schemas: In this case, the extending unit defines an operation schema, say *Op*, for which there is no corresponding definition in the extended units.

Redefinition of imported operation schemas: In this case, the extending unit defines an operation schema, say *Op*, for which there are corresponding definitions in the extended units; but these are ignored. The redefinition is made explicit by including the identifier *Op* in the `redefine` list associated with the corresponding extended unit.

Extension of imported operation schemas: In this case, the extending unit defines an operation schema, say Op_{Ext}, for which corresponding operation schemas *Op* are defined in the extended units.

The following discussion is carried out with respect to the operation/predicate *Op*. In the first two of the above cases, the extending unit *CExt* must define the operation schema

$$Op == [\,\Delta State;\ \ldots \mid \ldots]$$

with respect to the overall data state. In the last case, the extending unit *CExt* must define the operation schema

$$Op_{Ext} == [\,\Delta State;\ \ldots \mid \ldots]$$

which defines further constraints on the operation *Op*. Note that Op_{Ext} is defined with respect to $\Delta State$ rather than $\Delta State_{Ext}$: this allows Op_{Ext} to define constraints on the data state components of the extended units not defining *Op*.

Assume that *Op* is defined by the extended units $Base_{i1}, \ldots, Base_{iM}$. Then, the specification unit resulting from the reduction process contains the definition

$$Op == (\bigwedge_{j \in \{i_1, \ldots, i_M\}} Base_j.Op) \wedge Op_{Ext}$$

If, for an operation schema *Op* defined by an extended unit, the extending unit does neither define *Op* nor Op_{Ext}, the last case applies with the schema Op_{Ext} being set to $[\mid true]$ by default.

`LOCAL`. The transformation of the declared channels is analogous to that discussed with respect to the `INTERFACE` section.

`BEHAVIOUR`. The extending specification unit *CExt* is able

- to extend the imported behaviour patterns of the extended units or
- to completely redefine the dynamic behaviour.

In the first case, the extending unit defines, by convention, the process term $Behaviour_{Ext}$. The defined process is composed in parallel with the behaviour patterns of the extended units, which corresponds to a logical conjunction.

$$Behaviour \mathrel{\widehat=} (\mathop{\Big\|}_{i \in \{1, \ldots, N\}} Base_i.Behaviour) \parallel Behaviour_{Ext}$$

The synchronising parallel operator $\|$ enforces a synchronisation of all processes on the operation-related events.

In the second case, the dynamic behaviour is defined from scratch in the extending unit by means of the process term *Behaviour*. This can be done by referring to processes defined in the extended units $Base_1, \ldots, Base_N$.

Let us turn our attention to abstract specification units. The structure of abstract specifications in RT-Z is less complex than the structure of concrete specifications, because extension is the only structuring mechanism applicable to abstract specification units. Therefore, an abstract specification in RT-Z is simply a tree, where the nodes are constituted by abstract specification units and where the edges represent extension relationships.

We now address the extension of abstract specification units. Let us consider the schematic definition of the abstract specification unit *AExt* as an example for the subsequent discussion.

SPEC. UNIT *AExt*

EXTENDS

$Base_1$	$\ldots$	$Base_N$
`instantiate ...;`		`instantiate ...;`
`rename ...;`		`rename ...;`

TYPES & CONSTANTS

- new data types and constants
- additional constraints on imported constants

INTERFACE

- new ports
- additional constraints on domains of imported ports

ENVIRONMENTAL ASSUMPTIONS

- environmental assumptions concerning new ports
- additional environmental assumptions concerning imported ports (EA_{Ext})

I/O RELATIONS

- new I/O relations
- extension of imported I/O relations
- redefinition of imported I/O relations

STATE PROPERTIES

- new state components

  $State_{Ext} == \ldots$ $Init_{Ext} == \ldots$
- additional global invariants

  $GlobalInv == \ldots$ $GlobalInit == \ldots$
- new time-variant state properties
- additional constraints on imported time-variant state properties

BEHAVIOURAL PROPERTIES

- redefinition

  $Behaviour \mathrel{\widehat=} \ldots$
- extension

  $Behaviour_{Ext} \mathrel{\widehat=} \ldots$

Again, the meaning of the extension of abstract specification units is defined in terms of transformation rules, which we explain separately for each section.

TYPES & CONSTANTS, INTERFACE, ENVIRONMENTAL ASSUMPTIONS. The rules for reducing these sections of the extending specification unit *AExt* and of the extended specification units $Base_1, \ldots, Base_N$ are identical with those in the context of concrete specification units.

I/O RELATIONS. There are three alternatives for the extending specification unit *AExt* for defining I/O relations:

Introduction of new I/O relations: In this case, the extending unit defines a schema, say *IORel*, for which there is no corresponding definition in the extended units.

Redefinition of imported I/O relations: In this case, the extending unit defines a schema, say *IORel*, for which there are corresponding definitions in the extended units; but these are ignored.

Extension of imported I/O relations: In this case, the extending unit defines a schema, say $IORel_{Ext}$, for which corresponding schemas *IORel* are defined in the extended units.

The following discussion is carried out with respect to the I/O relation *IORel*. In the last of the above cases, the extending specification unit must define the schema $IORel_{Ext}$, which is interpreted as extending the I/O relation schema *IORel*. We assume that *IORel* is defined by the extended specification units $Base_{i1}, \ldots, Base_{iM}$. Then, the specification unit resulting from the reduction process will contain the definition

$$IORel == (\bigwedge_{j \in \{i_1, \ldots, i_M\}} Base_j.IORel) \wedge IORel_{Ext}$$

If, for a schema *IORel* defined by an extended unit, the extending unit does neither define *IORel* nor $IORel_{Ext}$, the last case applies with the schema $IORel_{Ext}$ being set to $[\mid true]$ by default.

`STATE PROPERTIES`. For abstract specification units, the derivation of the *State* schema from the data states of the extending unit *AExt* and of the extended units $Base_1, \ldots, Base_N$ is identical with the corresponding derivation in the context of concrete specification units. Among other things, this means that the extending unit *AExt* can define the schemas $State_{Ext}$ and *GlobalInv* in its `STATE PROPERTIES` section.

There are three alternatives for the extending specification unit *AExt* for defining time-variant state properties:

Introduction of new time-variant state properties: In this case, the extending unit defines a schema, say *Prop*, for which there is no corresponding definition in the extended units.

Redefinition of imported time-variant state properties: In this case, the extending unit defines a schema, say *Prop*, for which there are corresponding definitions in the extended units; but these are ignored.

Extension of imported time-variant state properties: In this case, the extending unit defines a schema, say $Prop_{Ext}$, for which corresponding schemas *Prop* are defined in the extended units.

The following discussion is carried out with respect to the time-variant state property *Prop*. In the first two cases, the extending unit *AExt* must define the schema

$$Prop == [State;\ \ldots \mid \ldots]$$

with respect to the overall data state. In the last case, the extending unit *AExt* must contain the schema

$$Prop_{Ext} == [State;\ \ldots \mid \ldots]$$

which defines further constraints on the time-variant state property *Prop*. Note that $Prop_{Ext}$ is defined with respect to *State* rather than $State_{Ext}$: this allows $Prop_{Ext}$ to define constraints on the data state components of the extended units not defining *Prop*.

Assume that *Prop* is defined by the extended units $Base_{i1}, \ldots, Base_{iM}$. Then, the specification unit resulting from the reduction process will contain the definition

$$Prop == (\bigwedge_{j \in \{i1,\ldots,iM\}} Base_j.Prop) \wedge Prop_{Ext}$$

If, for a schema *Prop* defined by an extended unit, the extending unit does neither define *Prop* nor $Prop_{Ext}$, the last case applies with the schema $Prop_{Ext}$ being set to $[\mid true]$ by default.

BEHAVIOURAL PROPERTIES. The extending specification unit *AExt* is able

- to extend the imported behaviour patterns of the extended units or
- to completely redefine the dynamic behaviour.

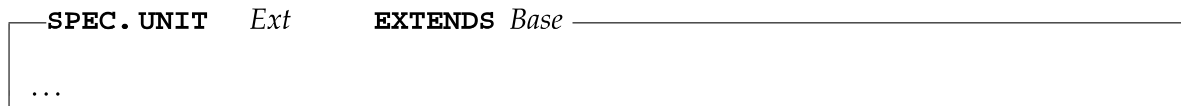
In the first case, the extending unit defines, by convention, the predicate $Behaviour_{Ext}$. The defined predicate is conjoined with the behaviour definitions of the extended units.

$$Behaviour \mathrel{\widehat{=}} (\bigwedge_{i \in \{1,\ldots,N\}} Base_i.Behaviour) \wedge Behaviour_{Ext}$$

In the second case, the dynamic behaviour is defined from scratch in the extending unit by means of the predicate *Behaviour*. This can be done by referring to predicates defined in the extended units $Base_1, \ldots, Base_N$.

Having described the transformation rules for extension, we have defined the meaning of extension in a semi-formal yet systematic manner.

The syntax for extension used so far is the vertical form. In cases in which the extension of a specification unit is carried out without instantiating or renaming identifiers, the horizontal form can be used.



In the horizontal form, the identifier of the specification unit to be extended is recorded in the headline of the extending specification unit.

In the current and previous section, we have treated the transformation rules for aggregation and extension separately. In general, however, an RT-Z specification contains both aggregation and extension relationships. So, we must address the interplay between the two kinds of transformation rules.

According to our previous remarks, an RT-Z specification is a finite, directed and acyclic graph, whose nodes are constituted by (simpler) graphs and whose edges represent extension relationships. Each graph constituting a node of the top-level graph is a tree of

specification units whose edges represent only aggregation relationships (i.e., aggregation hierarchies). Since the top-level graph is finite and acyclic, there must exist nodes without outgoing edges. These leaf nodes, constituted by aggregation hierarchies, can be reduced to single specification units by applying the transformation rules for aggregation. Then, the extension relationships leading to these leaf nodes can be eliminated by applying the transformation rules for extension. The resulting top-level graph is smaller than the original graph, so finitely many of the above steps suffice to reduce the whole RT-Z specification to a single specification unit whose meaning is the meaning of the whole specification by definition.

8.3 Renaming

When instantiating a specification unit by means of aggregation or extension, the elements of its interface can be renamed. The syntax in the context of aggregation is as follows.

`subunit` *agg* `spec.unit` *Spec*
    `rename` $chan_1$ `to` $chan_1'$, ..., $chan_N$ `to` $chan_N'$

The meaning of the renaming operator for specification units (which should not be confused with the CSP renaming operator for processes) is the renaming of the involved external channels in the `INTERFACE`, `BEHAVIOUR` and `ENVIRONMENTAL ASSUMPTIONS` sections of the specification unit *Spec*. The renaming of the channel identifiers must represent an injective function.

8.4 Hiding

When instantiating a specification unit by means of aggregation, arbitrary elements of the interface can be hidden. The syntax is as follows.

`subunit` *agg* `spec.unit` *Spec*
    `hide` $chan_1, \ldots, chan_N$

The meaning of the hiding operator is given by transferring the listed channels from the `INTERFACE` to the `LOCAL` section within the specification unit *Spec*.

8.5 Parametrisation

To support the reuse of specification units in different contexts, any specification unit may contain a list of formal parameters that must be instantiated by the respective aggregation/extension context with appropriate actual parameters. Formal parameters are placeholders for data types or constants. They are introduced in the head of a specification unit within square brackets. Formal parameters denoting data types allow us to define generic specification units.

Each formal parameter introduced by a specification unit must additionally be declared in its `TYPES & CONSTANTS` section in order to allow us to type-check specification units independently of their context. A parametrised specification unit can define constraints on the allowed instantiations of its formal parameters in its `TYPES & CONSTANTS` section by expressing constraints on the formal parameters with the help of the Z notation.

When the specification unit

SPEC. UNIT $Agg[fp1, \ldots, fpN]$

$\ldots$

is aggregated by another specification unit

`subunit` *agg* `spec. unit` *Agg*
`instantiate` fp_1 `with` ap_1, ..., fp_N `with` ap_N

the formal parameters are substituted by the supplied actual parameters.

Note that the top-level specification unit of a specification must not contain formal parameters, because it is not embedded into a context which provides actual constants and types.

8.6 Example: Alternating Bit Protocol

To illustrate the structuring mechanism aggregation, we proceed with the example started in Chapter 7: the alternating bit protocol. There, we have discussed the requirements specification of a reliable communication medium and one component of the design specification of the AB protocol. In the following, we present the top-level specification unit *ABProtocol* of the design, which aggregates the four components that are depicted in Figure 7.2.

SPEC. UNIT *ABProtocol*

INTERFACE

`port` In, Out `domain` *Message*

TYPES & CONSTANTS

$Bit ::= true \mid false$

$MaxTransDelay, MaxAckDelay : \mathbb{T}$

$MaxAckDelay >_{\mathbb{R}} MaxTransDelay +_{\mathbb{R}} MaxTransDelay$

ENVIRONMENTAL ASSUMPTIONS

$EA \mathrel{\widehat=} \forall t : \mathbb{T};\ msg : Message \bullet$
$\quad Out.msg$ live from $t \Rightarrow$
$\quad \exists\, dt : \mathbb{T} \mid dt \leqslant_{\mathbb{R}} (MaxTransDelay +_{\mathbb{R}} MaxTransDelay) \bullet Out.msg$ at $(t +_{\mathbb{R}} dt)$

SUBUNITS

subunit *sender* spec. unit *Sender*
  instantiate *MSG* with *Message*, *ACK* with *Bit*,
    MaxDelay with *MaxAckDelay*;
subunit *receiver* spec. unit *Receiver*
  instantiate *MSG* with *Message*, *ACK* with *Bit*,
    MaxDelay with *MaxAckDelay*;
subunit *msg_medium* spec. unit *UnreliableMedium*
  instantiate *MSG* with *Message* × *Bit*, *MaxDelay* with *MaxTransDelay*;
  rename *Send* to *MsgSend*, *Rcv* to *MsgRcv*;
subunit *ack_medium* spec. unit *UnreliableMedium*
  instantiate *MSG* with *Bit*, *MaxDelay* with *MaxTransDelay*;
  rename *Send* to *AckSend*, *Rcv* to *AckRcv*;

LOCAL

channel *MsgSend*, *MsgRcv* domain *Message* × *Bit*;
channel *AckSend*, *AckRcv* domain *Bit*

STATE

GlobalInit
$sender.ack = receiver.ack$

BEHAVIOUR

$$
\begin{aligned}
Behaviour \mathrel{\widehat=}\ & (sender.Behaviour \\
& |[\ \{\!|\ MsgSend\ |\!\} \cup \{\!|\ AckRcv\ |\!\}\ ]| \\
& \quad (msg_medium.Behaviour\ |||\ ack_medium.Behaviour) \\
& |[\ \{\!|\ AckSend\ |\!\} \cup \{\!|\ MsgRcv\ |\!\}\ ]| \\
& \quad receiver.Behaviour)
\end{aligned}
$$

Consider first the SUBUNITS section. It introduces the four components depicted in Figure 7.2 by means of simple aggregation. The aggregated units *msg_medium* and *ack_medium* are both instances of the specification unit *UnreliableMedium*, because they have the same

generic behaviour. To take their differences into account, both declarations associate the formal parameter *MSG* with different sets (`instantiate` statement), and the identifiers of the external channels are renamed appropriately in order to embed the media into their respective context (`rename` statement).

The data state of the protocol is constituted by the data states of the aggregated units, in this case of the units *sender* and *receiver*. The schema *State* of the aggregating unit *ABProtocol* is derived as discussed in Section 8.1.4.

State
sender : *Sender*.*State*
receiver : *Receiver*.*State*

Since there are no constraints on the relationship between the data states of the sender and receiver components, the schema *GlobalInv* is omitted. There is, however, a constraint on the relationship between the states of the acknowledgement bits of the components at initialisation, stipulated by the schema *GlobalInit*: the bits must be initialised with the identical value, but the particular value is irrelevant.

Finally, the `BEHAVIOUR` section fixes the configuration of the aggregated units, including their interaction via internal channels. This is achieved by means of the process term language of timed CSP. Note that all channels that are declared in the `LOCAL` section are hidden from the interface by definition.

The specification units *Receiver* and *UnreliableMedium*, which define the behaviour of the receiver component and of the unidirectional, unreliable message media, are given for the sake of completeness.

The crucial aspect of the definition of the specification unit *UnreliableMedium* is how it makes explicit the possible error modes of the message medium in the behaviour definition: after receiving a message on the channel *Send*, it chooses an $i \in \mathbb{N}$ nondeterministically. If $i = 0$, then the interleaving process expression simplifies to $|||_{i \in \varnothing} Trans(i, m)$, which is equivalent to *Skip*. This case represents the loss of the received message. If $i = 1$, then the interleaving process expression simplifies to $|||_{i \in \{1\}} Trans(i, m)$, which is equivalent to $Trans(1, m)$. This case represents the correct behaviour of the message medium, because one copy of the received message is correctly delivered. The cases in which $i > 1$ represent the duplication of the received message.

SPEC. UNIT *UnreliableMedium* [*MSG*, *MaxDelay*]

TYPES & CONSTANTS

[*MSG*] | $MaxDelay : \mathbb{T}$

INTERFACE

`port` *Send*, *Rcv* `domain` *MSG*

BEHAVIOUR

$Behaviour \mathrel{\widehat=} Cycle$

$$Cycle \mathrel{\widehat=} Send?(m : MSG) \rightarrow \bigsqcap_{i \in \mathbb{N}} \left(\mathop{|||}_{j \in 1..i} Trans(j, m) \right) ;\ Cycle$$

$Trans(i, m) \mathrel{\widehat=} Wait((i-1) *_{\mathbb{R}} MaxDelay, i *_{\mathbb{R}} MaxDelay];\ Rcv!m \rightarrow Skip$

The specification unit *Receiver* is defined analogously to the specification unit *Sender* discussed in detail in Section 7.1.

SPEC. UNIT $Receiver\ [MSG, ACK, MaxDelay]$

TYPES & CONSTANTS

$[MSG, ACK]$

$\#ACK = 2$
$MaxDelay : \mathbb{T}$

INTERFACE

port Out domain MSG;
port $MsgRcv$ domain $MSG \times ACK$;
port $AckSend$ domain ACK

STATE

$State == [ack : ACK]$

OPS & PREDS **[** $CorrectAck, IncorrectAck$ **]**

$CorrectAck == [\Delta State;\ rec? : ACK \mid rec? = ack \wedge ack' \neq ack]$
$IncorrectAck == [\Xi State;\ rec? : ACK \mid rec? \neq ack]$

BEHAVIOUR

$Behaviour \mathrel{\widehat=} REC$

$$\begin{aligned} REC \mathrel{\widehat=}\ & (MsgRcv?(rec : MSG \times ACK) \rightarrow \\ & \quad (CorrectAck_X!\langle\!| rec? == rec.2 |\!\rangle \rightarrow \\ & \quad\quad (Out!(rec.1) \rightarrow Skip \mathbin{|||} \sigma(AckSend!ack \rightarrow Skip));\ REC \\ & \quad \Box\ IncorrectAck_X!\langle\!| rec? == rec.2 |\!\rangle \rightarrow \sigma(AckSend!ack \rightarrow REC))) \\ & \rhd\{MaxDelay\}\ \sigma(AckSend!ack \rightarrow REC) \end{aligned}$$

This completes our informal discussion of the integrated formalism RT-Z. We have introduced two models of integrating Z and timed CSP, we have described the links between the base formalisms in the two models, we have discussed the structure of abstract and concrete specification units and we have introduced and defined the structuring mechanisms for organising large and complex specifications.

In the next part, we back up formally the RT-Z notation introduced in this part.

Part III

Formal Foundation

Chapter 9

Denotational Semantics

According to the classification of approaches to combining complementary formalisms proposed in Chapter 5, a conserving integration must be equipped with a common semantics. In this chapter, we define a denotational semantics for single specification units of RT-Z. Corresponding to the distinction between abstract and concrete specification units in Chapter 7, we have to define two different semantic functions: in Section 9.2, we define the meaning of concrete specification units; abstract specification units are dealt with in Section 9.3.

9.1 Basics

In correspondence with the denotational semantics of timed CSP, the denotational semantics of RT-Z maps each RT-Z specification unit to the set of timed observations that could be made during the evolution of the specified artifact. In the semantic model of RT-Z, such a timed observation consists of two components.

- The external interaction of the artifact in the observation interval, consisting of
  - the record of which events have occurred in which order and at which time instants at the interface between the artifact and its environment (where an event may involve the communication of a complex data value) and
  - the record of which offered events have been refused by the artifact at particular time instants.

  This corresponds to the structure of process denotations in timed CSP, being composed of timed traces and timed refusals.
- The evolution of the artifact's data state in the observation interval. We call such a record *timed state*. It has no counterpart in the semantic model of timed CSP.

The above description already incorporates an essential design decision, namely that the evolution of the data state should be part of the denotations. This decision is in accordance with the semantic model of TCOZ [Mahony and Dong, 1999a] but in contrast to the one of

CSP-OZ [Fischer, 2000], see Chapter 4. Let us first briefly discuss the decision to include the evolution of the data state in the denotations of an RT-Z specification.

The semantics of CSP-OZ maps each CSP-OZ class to a set of failure/divergence pairs[1], thereby abstracting from the evolution of the data state. From our point of view, too much information is lost by ignoring the data state evolution, because some kinds of requirements with respect to the behaviour of an object cannot be expressed in this case. More precisely, it is not possible in a satisfactory manner to restrict the allowed behaviour patterns of an object that has performed a nondeterministic operation whose nondeterminism is not visible to the environment (not visible via an external interaction). In other words, in CSP-OZ, the ability to express internal nondeterminisms related to the data state of the object under consideration stands in contradiction to the ability to make any statements about its required external behaviour.

We provide an example of such a situation in Figure 9.1.

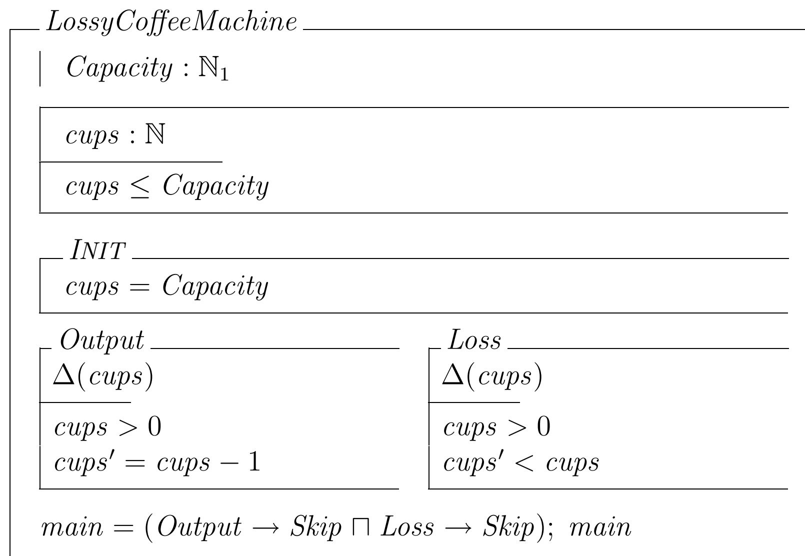


Figure 9.1: Lossy coffee machine.

The CSP-OZ class models a lossy coffee machine with two operations: *Output* models the output of one cup of coffee to the user, and *Loss* models a malfunction of the machine, loosing an arbitrary number of cups of coffee nondeterministically. It is important to note that this nondeterminism—the amount of coffee lost—is not visible to the environment. According to the semantics of CSP-OZ, the failure $(\langle Loss \rangle, \{Output, Loss\})$ is an observation of the specified machine: after a loss has occurred, the machine is able to refuse to output further cups of coffee, regardless of whether or not it has still coffee available. The reason is that the *Loss* operation could potentially lead to a completely empty coffee machine, which is unable to output further coffee. Since the data state that actually resulted from the *Loss* event is ignored by the semantics, one cannot make the acceptance of the output of coffee dependent on the

[1] in the sense of the failures–divergences model of (untimed) CSP

real amount of coffee available. This means that, on the basis of the CSP-OZ semantics, it is not possible to express requirements with respect to the behaviour of the machine like: “the machine is ready to output coffee as long as it is not empty.”

To model such systems properly, including nondeterminisms not visible to the environment, the evolution of the data state must be taken into account.

As discussed in Chapter 2, we must distinguish between two system views in the course of the system development process. In the open system view, the artifact at hand is considered as a white box, i.e., its internal properties are visible, whereas it is considered as a black box in the closed system view. The intention pursued by the above argument was to justify the need for the open system view from the semantic point of view. This, however, does not imply that we can dispense with a semantic model reflecting the closed system view.

Accordingly, the approach taken in this chapter is as follows. We first deal with the timed failures/states model of RT-Z in Sections 9.2 and 9.3, which reflects the open system view and is the more complex model. Afterwards, we treat the extended timed failures semantics of RT-Z specification units in Section 9.4, which reflects the closed system view and which can be easily derived from the former.

9.2 Concrete Specification Units (Open System View)

The structure of concrete specification units and the principles of how they integrate the Z and timed CSP notation are discussed in Section 7.1. In this section, we define a common semantic model unifying the semantic models of the base formalisms in order to map each concrete specification unit of RT-Z to a unique meaning.

In Section 9.2.1, we elaborate the consequences of the potential concurrency of operations working on different parts of the data state space. In particular, we define constraints on the possibility to apply operations concurrently. They ensure that scenarios of concurrent operation applications can be related to corresponding evolutions of the underlying data state in a reasonable manner. Then, in Section 9.2.2, we give an overview of the semantic integration within concrete specification units and outline its major steps. Sections 9.2.3 and 9.2.4 introduce intermediate semantic models that are the basis for the final timed failures/states model dealt with in Section 9.2.5. The formal definition of the semantic integration of the Z part and the CSP part of a concrete specification unit in Section 9.2.6 on the grounds of the timed failures/states model concludes the current section.

In Section 9.3, we define the timed failures/states semantics of abstract specification units, which is done much more succinctly, given their less complex structure and the definitions in the current section.

9.2.1 Concurrency on Common Data State

Let us consider in the following an RT-Z system specification that is composed of an *aggregation hierarchy* of specification units. In this case, the overall data state of the specified system

can be regarded as being hierarchically partitioned into subspaces according to the hierarchical decomposition of the system specification into specification units, where each specification unit defines its associated data state. Roughly speaking, each operation defined in one of the specification units depends on and transforms only the subspace of the overall data state that is associated with its specification unit. When applied, it should leave the other subspaces of the overall data state unchanged and its transformation should depend only on the current values of variables within this subspace of the data state. However, when we take the concept of *global invariants* into account, this clear separation between different subspaces is dropped: a global invariant establishes relationships between variables in different subspaces of the overall data state, which must be respected when an operation transforms one subspace of the data state.

As a consequence, RT-Z addresses the concurrent application of operations only under two specific conditions that are discussed in the following paragraph; if these conditions are not met, RT-Z refuses to make any statements about the result of the concurrent application. More specifically, the behaviour that is specified if these conditions are not met is that of the diverging process, which denotes the set of all timed failures and timed states.

Roughly speaking, the concurrent application of operations is addressed by RT-Z only if these operations do not interfere with each other, that is, if their corresponding transformations of the data state do not affect each other. More precisely, the effect of concurrently applying a set of operations, say *OpSet*, with particular input parameters is well-defined with respect to the current data state only if the following two conditions are satisfied:

1. The subspaces of the data state potentially changed by applying the operations of *OpSet* must be pairwise disjoint. In other words, there must not be any state component that might be subject to change by more than one of the concurrently applied operations.

2. The particular order in which the individual transformations of the data state take place must not be relevant. That is, the set of final data states and output parameters the concurrent application of the operations in *OpSet* may produce must be independent of the particular order in which the corresponding transformations of the data state really take place. Moreover, this set of final data states and produced output parameters must not be empty. This excludes the case that the application of some of the operations of *OpSet* violates the preconditions of others, ensuring that each operation of *OpSet* can be applied to the data state resulting from the application of any combination of other operations of *OpSet*.

In brief, these conditions allow operations transforming disjoint subspaces of the overall data state to be applied concurrently if they do not interfere with each other via global invariants.

If the concurrent application of a set of operations meets the above 'non-interference' conditions with respect to the current data state, then its effect on the current data state is the sequential composition of the individual state transformations in an arbitrary order. If, on the other hand, operations are applied concurrently without satisfying these conditions, the effect on the data state is left unspecified: RT-Z does not really prevent the concurrent application of operations interfering with each other, but refuses to define its result. It is the

$State == [\, n, m : \mathbb{N} \mid n \leq m \,]$
$Op1 == [\, \Delta State \mid n' = n + 1 \land m' = m \,]$
$Op2 == [\, \Delta State \mid m' = m - 2 \land n' = n \,]$

t	$t1$	$t2$	$t3$	$t4$
invocation / termination	$Op1_I$	$Op2_I$	$Op1_T$	$Op2_T$
(n, m)	$(2, 5)$	$(2, 5)$	$(3, 5)$	$(3, 3)$
	$(3, 5)$	$(3, 5)$	$(4, 5)$	divergence
	$(4, 5)$	X	$(5, 5)$	X

Figure 9.2: Concurrent application of operations.

responsibility of the specifier to guarantee that the concurrent application of interfering operations is prevented. This gives rise to a specific kind of validation condition.

As already explained, there are two alternatives for modelling state transitions (operations); either an operation is modelled to be applied during a time interval where this interval is delimited by the events of invocation Op_I and termination Op_T, or an operation is modelled to be applied instantaneously, marked by the event of execution Op_X. These two cases must be distinguished regarding the above 'non-interference' conditions.

In the double-event approach, the application of an operation, say *Op*, starts with the occurrence of the invocation event Op_I. For the operation to be invoked, the precondition of the operation must be satisfied with respect to the current data state, where the evaluation of the precondition may depend on the overall data state, not only on the subspace associated with the operation (see the example on p. 66). The application of the operation is completed with the occurrence of the termination event Op_T; the caused state transition takes effect at this time instant: it is based on the data state that is present at termination. It is important to note that the overall data state may have changed during the application—due to the termination of other operations applied concurrently. In the double-event approach, the above 'non-interference' conditions apply to operation applications whose application intervals—the intervals between invocation and termination—overlap.

In the single-event approach, on the other hand, the instantaneous application of an operation, say *Op*, is marked by the occurrence of the execution event Op_X. For the operation to be executed, its precondition must be evaluated successfully with respect to the current data state. The above 'non-interference' conditions apply to operations executed simultaneously, i.e., at the same time instant.

We illustrate the above discussion with the help of the example in Figure 9.2. The underlying data state consists of two components n and m, where n must be less than or equal to m. Further, there are two operations, $Op1$ and $Op2$, incrementing n by 1 and decrementing m by 2, respectively.

We consider three scenarios in which the operations are applied concurrently in the context of different data states.

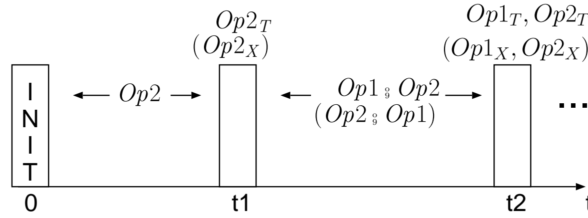


Figure 9.3: Structure of timed states.

In the first scenario, $Op1$ and $Op2$ are invoked at $t1$ and $t2$ in this order with respect to the data state $(n, m) = (2, 5)$. This data state allows the operations to be applied in a well-defined manner, because in whatever order their state transformations take place, the final data state will be $(3, 3)$. Let $Op1$ and $Op2$ terminate at $t3$ and $t4$ in this order. Then, the termination of $Op1$ leads to the data state $(3, 5)$.

Note first that the state transformation that is caused by the termination of $Op2$ relates the data states $(3, 5)$ and $(3, 3)$, where $(3, 5)$ is the data state that is present when $Op2$ terminates. At first glance, this might appear counterintuitive, because the state transformation of $Op2$ is not based on the data state $(2, 5)$ that is present when it is invoked. However, a closer look reveals that any change of the data state that might have taken place between the invocation at $t2$ and the termination at $t4$ cannot have any effect on $Op2$'s transformation within its associated subspace of the overall data state, as long as we consider a well-formed (i.e., non-interfering) application scenario.

Note also that, even though the state transition that is caused by $Op2$ is based on the data state $(3, 5)$ present at termination, it is the data state $(2, 5)$ present at invocation that is used to determine whether $Op2$ can be applied (by computing the precondition). Again, as long as we consider a well-defined application scenario, the fact that the precondition of $Op2$ is satisfied at invocation implies that it must also be satisfied at termination.

In the second application scenario, $Op1$ and $Op2$ are applied concurrently with respect to the data state $(3, 5)$, which does not allow the operations to be applied concurrently in a well-defined manner; because the first operation terminates necessarily in a data state violating the precondition of the other one. In the example, $Op1$ terminates at $t3$ in the data state $(4, 5)$, in which the precondition of $Op2$ is not satisfied. Hence, the termination of $Op2$ at $t4$ results in divergence.

In the last application scenario, we consider a data state $(4, 5)$ that prevents the invocation of $Op2$ at $t2$, because its precondition is not satisfied. Thus, in contrast to the previous scenario, the attempt to apply $Op1$ and $Op2$ concurrently does not result in divergence, but in the application of only $Op1$.

Let us now consider the structure of timed states as outlined in Figure 9.3, each of which is a constituent of a timed observation in RT-Z. A timed state has the purpose to record the changes of the data state caused by the execution or termination of operations. Since arbitrarily many operations may be executed simultaneously (single-event approach) and/or may terminate simultaneously (double-event approach), a timed state must be able to record

arbitrarily many state changes for a single time instant. This is achieved as follows: a timed state assigns a data state to any time instant at which at least one operation is executed or terminates. This data state represents the result of all individual state changes at that time instant. In other words, it combines the individual state changes to a single change. Accordingly, a timed state is a partial function mapping those time instants of the observation interval at which at least one operation is executed or has terminated to the data state that is the result of applying in an arbitrary order all these operations to the previously recorded data state.

This leads to the following interpretation of a timed state. At initialisation (time instant 0), the timed state records the initial data state, which is one of those data states characterised by the *Init* schema of the Z part. For each time instant at which no execution or termination event occurs, the timed state is undefined. The current data state at such a time instant is the last data state recorded in the timed state before. For each time instant at which at least one operation is executed or terminates, the timed state records all state changes that have resulted from these operation executions or terminations.

9.2.2 Semantic Integration: Overview

We define a denotational semantics, which associates a set of mathematical objects with a single concrete specification unit; the denotations represent possible timed observations that could be made when observing the artifact under consideration. Concerning the design of this denotational semantics, a major decision is not to begin from scratch, but to make use of the denotational semantics of the base formalisms as far as possible. For both Z and timed CSP, denotational semantics are provided by the Final Committee Draft of the ISO standard for Z [ISO, 2002] and the *finite timed failures model* [Schneider, 1999b], respectively. Reusing the denotational semantics by encapsulating their semantic functions within the semantic function of RT-Z is not only more economical than defining an entirely new semantic function, but also makes the semantics of RT-Z more robust against potential changes of the semantics of the base formalisms. Following the definition of the Z semantics in [ISO, 2002], we use the Zermelo–Fraenkel (ZF) axiomatisation of set theory [Hamilton, 1982, Enderton, 1977] in combination with logic as the metalanguage for defining the denotational semantics of the integrated formalism. We follow the syntactic conventions of using the ZF set theory that are applied in the Z Standard [ISO, 2002, Chapter 4]. These syntactic conventions lead to a notation that is similar to the Z notation itself. However, the two notations should not be confused.

The main principle of integrating the semantic models of Z and timed CSP follows an idea of Smith [1997] and adapts it to our specific context: each Z specification is given an interpretation in the semantic model of timed CSP. Using this interpretation, the combination of the Z and the (timed) CSP part of a specification unit is interpreted as their parallel composition, where the two parts synchronise on the events related to the operations of the Z part. The definition of the semantics of concrete specification units consists of three steps associated with corresponding intermediate semantic models, which are depicted in Figure 9.4: the history model, the extended timed failures model and the timed failures/states model; the Z Standard and the timed failures model of timed CSP are their foundation.

In Figure 9.4, there are two lines of transformation. The first line deals with the transform-

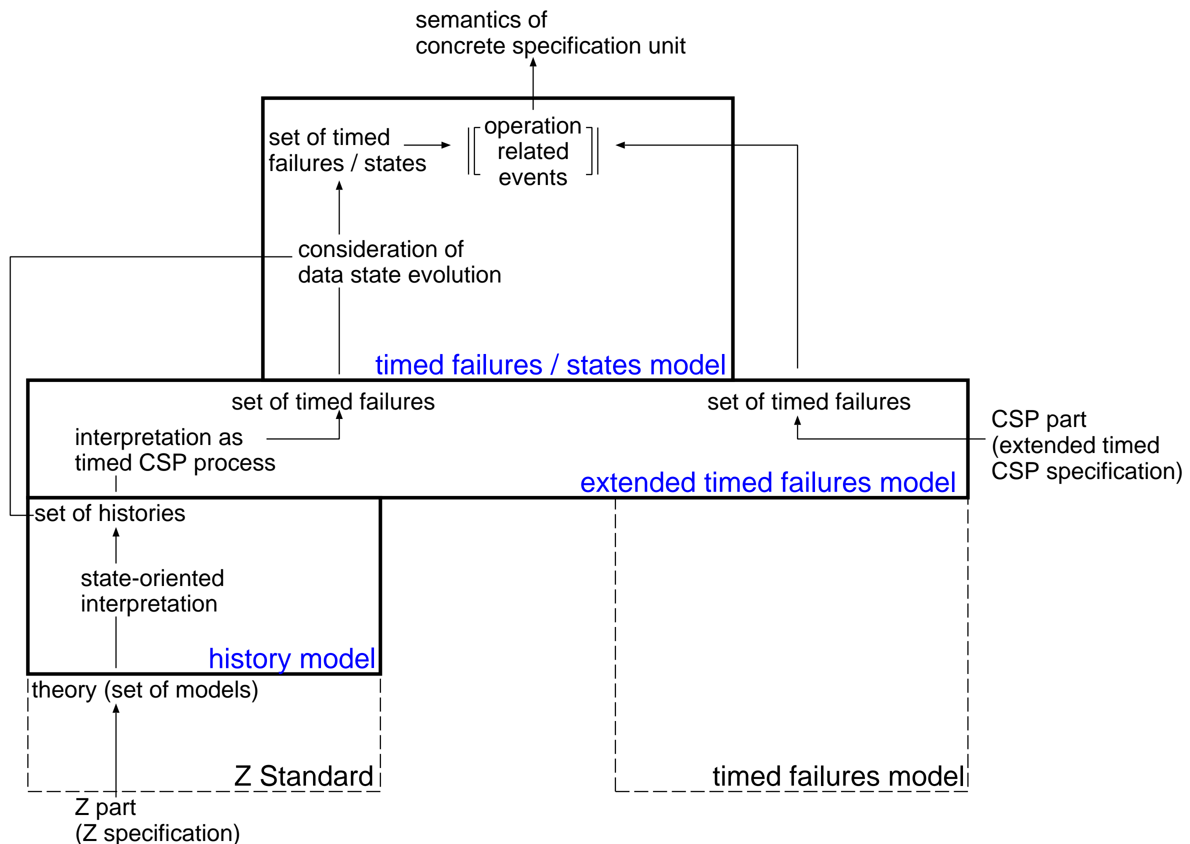


Figure 9.4: Steps of the semantic integration.

ation of the denotation of a Z specification, as defined by the Z Standard, into a form compatible with the denotation of a timed CSP process, i.e., it gives a Z specification a timed failures/states semantics. The second line is about giving a timed CSP specification a meaning in the timed failures/states model of RT-Z. As just discussed, finally, the appropriately transformed interpretations of the Z and the CSP part are composed in parallel to obtain the semantics of the overall concrete specification unit.

The simplest intermediate model is the history model. The denotational semantics given in the Z Standard maps each section of a Z specification to a set of so-called *models*. A model is a function mapping each identifier occurring in the considered specification section to a value, which the identifier may denote according to all constraints defined in the whole specification. The Z Standard does not fix a particular interpretation of Z specifications; in particular, it does not reflect the *state-oriented* conventions of interpreting Z specifications; however, this is the mandatory interpretation in the context of concrete specification units. Therefore, we transform a model into a form reflecting the state-oriented interpretation. This interpretation regards a Z specification as defining a data state and a set of transitions on it. *Histories*, as used by Smith [1997], are mathematical objects reflecting this state-oriented interpretation. A history represents a possible observation of the specified artifact up to a particular time instant. It consists of the sequence of data states the artifact has passed

through, together with the sequence of operation events the artifact has performed. After this transformation, the meaning of a Z specification is expressed in terms of a set of histories.

The purpose of the extended timed failures model is twofold. First, it lifts the syntax and the semantic model of timed CSP such that timed CSP expressions can be embedded within a Z context. Second, it is the underlying model to give a Z specification—via the history model—a 'timed CSP-like' meaning. Hence, it brings together the semantic models of Z and timed CSP.

The main tasks related to the definition of this model are:

- The extension of the timed CSP syntax in order to express the links from the CSP to the Z part of a concrete specification unit.
- The definition of the concepts of the timed failures model in ZF set theory, including
  - the time domain of timed CSP processes
  - the constituents of the denotations in the timed failures model and
  - the properties of the timed failures model stating which sets of timed failures are valid, consistent timed observations of a process.
- The definition of the meaning of the extended syntax based on the denotational semantics of timed CSP.

The final model is the timed failures/states model; it augments the previous model by taking into account the evolution of the data state of the specified artifact in the considered observation interval. The lifting from the extended timed failures to the timed failures/states model concerns only the Z part, because the CSP part does not make any statements about the data state.

So far, the meaning of the Z part of a concrete specification unit is defined in terms of a set of pairs of timed failures and timed states, and the meaning of the CSP part is expressed in terms of a set of timed failures. In the final step, the meaning of the RT-Z specification unit is defined to be the relation between timed failures and timed states as defined by the Z part, where the domain of this relation is restricted to the set of timed failures as defined by the CSP part. More abstractly, the meaning of the specification unit is defined to be the parallel composition of its parts synchronising on the set of operation-related events, see Figure 6.5.[2]

9.2.3 History Model

For the purpose of defining the meaning of concrete specification units, we fix a particular interpretation of their Z parts. Roughly speaking, we adopt the *state-oriented* interpretation of Z specifications. This state-oriented interpretation, which is the most widespread convention of using Z in practice [Woodcock and Davies, 1996, Wordsworth, 1992], regards a

[2] To be more precise, the operation-related events are hidden after the parts have been composed in parallel. The operation-related events serve only to connect the parts; they are not part of the external interface of a specification unit. The information contained in Figure 6.5 is defined formally by the relations $tfs_of_model_{INT}$ and $tfs_of_model_{EXT}$ on p. 144.

Z specification as characterising the specified artifact as a state-transition system. Following this convention, a Z specification must define a dedicated schema that models the data state. Moreover, operation schemas are defined that model transitions on that data state. The meaning of a Z specification as defined by the Z Standard, however, is absolutely independent of any particular kind of interpretation. This semantics associates a *theory* with each (sectioned) Z specification, which is a mapping from section identifiers to sets of models. A *model*, in turn, is a function mapping each introduced identifier to a value that this identifier may denote according to all specified constraints. As already mentioned, it is the task of the first sub-step of the semantic integration to transform a set of models, denoting the meaning of the Z part of a concrete specification unit, into a set of histories, which reflect the state-oriented interpretation of this Z part.

The metalanguage used in this section to define the denotational semantics of RT-Z is based on logic and Zermelo–Fraenkel set theory, which is an *untyped* theory. Regarding the choice of the semantic metalanguage, we follow the approach of the Z Standard to defining the denotational semantics of the Z notation. The concrete syntax used in this section is identical in appearance to that of the base language Z itself, but is grounded only on logic and ZF set theory. Using the Z notation as the concrete syntax allows us to apply the powerful tool ESZ [Büssow and Grieskamp, 1998] for performing syntax checks: all formal definitions in this chapter are type-checked. However, since ZF set theory, in contrast to Z, is untyped, we are forced to "circumvent" the Z type system at different places.

To define the meaning of concrete specification units, we must be able to refer to their syntactical constituents, which are entire Z specifications (Z part), timed CSP process terms (`BEHAVIOUR` section), predicates of the timed failures model (`ENVIRONMENTAL ASSUMPTIONS` section) and Z expressions (channel signatures in the `INTERFACE` and `LOCAL` sections). Let *ZSpec*, *ZExpr*, *ETCSP_PROCESS* and *ETFM_PREDICATE* denote these syntactical units of the base formalisms. They are characterised by the corresponding syntax definitions of Z and timed CSP.

The `BEHAVIOUR` section can define any number of processes, and it must define the dedicated process *Behaviour*, which models the dynamic behaviour of the artifact under consideration. Let *PROCESS* be the set of all identifiers of processes. Then, the information induced by the `BEHAVIOUR` section is a function mapping the identifier of each defined process to the corresponding defining expression. We call that function *process environment*.

$$ProcEnv == PROCESS \twoheadrightarrow\!\!\!\!\!\!\mapsto ETCSP_PROCESS$$

The `INTERFACE` and the `LOCAL` sections of a specification unit associate Z expressions with external and internal channels, which are evaluated in the context of the Z part and denote the value domains of the declared channels. Let *CHANNEL* be the set of all identifiers of external and internal channels. The information induced by these two sections is a function mapping channel identifiers to Z expressions. We call that function *channel signature*.

$$ChanSgn == CHANNEL \twoheadrightarrow\!\!\!\!\!\!\mapsto ZExpr$$

Evaluated in the context of the Z part, each Z expression denotes the set of data values that can be communicated along the corresponding channel.

A concrete specification unit, as introduced in Chapter 7, incorporates

- a Z specification (*ZSpec*) containing its `TYPES & CONSTANTS`, `STATE` and `OPS & PREDS` section,
- two functions (I, L) mapping each external and internal channel, declared in the `INTERFACE` and `LOCAL` sections, to a Z expression,
- a predicate (*EA*) of the extended timed failures model constituting the `ENVIRONMENTAL ASSUMPTIONS` section,
- a behaviour definition (*Behav*) constituting the `BEHAVIOUR` section and
- a set of identifiers (*OPS*) characterising some schemas defined in the `OPS & PREDS` section as specifying operations and predicates (as opposed to auxiliary definitions).

$$\begin{aligned}
CSpecUnit == [&Zpart : ZSpec;\\
&I, L : ChanSgn;\\
&EA : ETFM_PREDICATE;\\
&Behav : ProcEnv;\\
&OPS : \mathbb{F}\, NAME]
\end{aligned}$$

According to the rules for constructing the Z part of a concrete specification unit explained in Chapter 7, each Z specification constituting such a Z part consists of the sections `TYPES & CONSTANTS`, `STATE` and `OPS & PREDS`, which are related via the obvious parent-relationships. The `STATE` section must introduce the dedicated schemas *State* and *Init*, and the `OPS & PREDS` section must define an operation schema for each identifier in the set *OPS*.

The Z Standard introduces the set *NAME* to denote the set of all identifiers of a Z specification and the function *decor* to define the decoration of identifiers. We assume that *NAME*, *PROCESS* and *CHANNEL* are disjoint. The Z Standard also introduces the symbols $\mathbb{W}$ to denote “a world of sets, providing semantic values for Z expressions,” $\mathbb{U}$ to denote “the semantic universe, providing semantic values for all Z expressions, including generic definitions” and *Model* to denote the set of all models.

Furthermore, we refer to the semantic functions $[\![_]\!]^{\epsilon}$ and $[\![_]\!]^{\mathcal{Z}}$ defined by the Z Standard. They define the meaning of Z expressions and sections of Z specifications, respectively.

When following the state-oriented conventions, the name space of each Z operation is partitioned into plain names, primed names, names ending with a ‘?’ (input parameters) and names ending with a ‘!’ (output parameters).

$$\begin{aligned}
&Primed == \{n : NAME \bullet decor\,'\, n\}\\
&Inputs == \{n : NAME \bullet decor\,?\, n\}\\
&Outputs == \{n : NAME \bullet decor\,!\, n\}\\
&PrimedOf == \{n : NAME \bullet n \mapsto (decor\,'\, n)\}\\
&PlainOf == \{n : NAME \bullet (decor\,'\, n) \mapsto n\}
\end{aligned}$$

State Model. We first define the data state space that is induced by the schema *State* of the STATE section of a concrete specification unit. More precisely, each particular model of the meaning of the Z part induces its specific data state. Accordingly, the entire discussion of the transformation from the Standard's semantics to the history semantics is done with respect to a particular model of the meaning of the considered Z part.

A binding, as usual, is defined as a finite function from identifiers to values.

$$Binding == NAME \ffun \mathbb{W}$$

The set of data states associated with a model $\mathcal{M}$ is the set of bindings that is related to the dedicated identifier *State*. The set of initial data states that are related to a model $\mathcal{M}$ are defined analogously.

$$StateModel == \{\mathcal{M} : Model;\ data_state : Binding \mid State \in \mathrm{dom}\,\mathcal{M} \land data_state \in \mathcal{M}\,State\}$$
$$InitModel == \{\mathcal{M} : Model;\ init_state : Binding \mid Init \in \mathrm{dom}\,\mathcal{M} \land init_state \in \mathcal{M}\,Init\}$$

Since we assume that the *Init* schema of each concrete specification unit imports the *State* schema, it follows that $InitModel(\!|\{\mathcal{M}\}|\!) \subseteq StateModel(\!|\{\mathcal{M}\}|\!)$ for all $\mathcal{M} : Model$.

Furthermore, the set of (undecorated) state variables introduced by the state schema is determined by the domain of the bindings that are the denotation of the schema *State*.

$$StateVars == \lambda\,\mathcal{M} : Model \mid State \in \mathrm{dom}\,\mathcal{M} \bullet \mathrm{dom}(\bigcup(\mathcal{M}\,State))$$

Operation Model. As discussed in Chapter 7, the OPS & PREDS section must define an operation schema for each identifier in the set *OPS*.

Each operation schema has a unique identifier and introduces a set of input and output parameters with corresponding types. Hence, an operation application is characterised by the identifier of the applied operation and the assignment of values to the operation's input and output parameters (the transformation of the data state caused by an operation application is considered separately).

$$OP_APP == NAME \times Binding$$

$$op_name == first[NAME, Binding]$$
$$op_params == second[NAME, Binding]$$

The relation *OpModel* characterises, for each operation schema in *OPS*, the set of corresponding operation applications (i.e., the assignment of values to the input and output parameters of the operations) and the set of data state transitions (i.e., the pairs of pre and post states) they are able to cause. Any model $\mathcal{M}$ associates a set of bindings with each operation schema *op*. These bindings contain the state variables of the pre state (in *StateVars*), the variables

of the post state (in *PrimedStateVars*), input variables (in *Inputs*) and output variables (in *Outputs*). A pair of data states is related to a particular assignment of values to input and output parameters if, and only if, their union is a member of the binding set characterised by the model.

$$
\begin{aligned}
&OpModel == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\, NAME;\ op : NAME;\\
&\qquad\qquad inout : Binding;\ delta_state : Binding \times Binding \mid\\
&\qquad op \in OPS \land OPS \subseteq \mathrm{dom}\, \mathcal{M} \land\\
&\qquad \mathrm{dom}\, inout \subseteq Inputs \cup Outputs \land\\
&\qquad (\exists_1\, pre == delta_state.1;\ post == delta_state.2 \bullet\\
&\qquad\quad \mathrm{dom}\, pre = \mathrm{dom}\, post = StateVars\, \mathcal{M} \land\\
&\qquad\quad (\exists\, bind : \mathcal{M}\, op \bullet bind = pre \cup (post \circ PlainOf) \cup inout))\\
&\bullet ((\mathcal{M}, OPS), (op, inout, delta_state))\}
\end{aligned}
$$

More abstractly, each operation schema induces a relation between operation applications, as defined by *OP_APP*, and data state transitions.

The function *AssocSubspace* maps each operation *op* to the subspace of the overall data state whose components are potentially subject to change when the operation is applied. This subspace is needed to formalise the non-interference conditions already presented informally.

$$
\begin{aligned}
&AssocSubspace == \lambda\, \mathcal{M} : Model;\ OPS : \mathbb{F}\, NAME;\ op : NAME \mid\\
&\quad op \in OPS \land OPS \subseteq \mathrm{dom}\, \mathcal{M} \bullet\\
&\quad StateVars\, \mathcal{M} \setminus \{var : StateVars\, \mathcal{M} \mid \forall\, bind : \mathcal{M}\, op \bullet bind\, var = bind(PrimedOf\, var)\}
\end{aligned}
$$

The relation *OpSetNonInterference* formally specifies the conditions that must be met for a set of operations *OpSet* to be applied concurrently with respect to a particular data state *current*, which have been informally stated in Section 9.2.1.

Firstly, the subspaces of the data state associated with the operations by the function *AssocSubspace* must be pairwise disjoint.

Secondly, the order in which the operations of *OpSet* really cause their corresponding state transformations must not be relevant for the set of possible final data states and produced output parameters. More formally, each pair of permutations $(perm1, perm2)$ of operation applications within *OpSet* must result in the same set of final data states and produced output parameters. This set must not be empty, ensuring that the operations within *OpSet* do not violate the preconditions of each other. The relation *OpSeqEffects*, which is an auxiliary definition, associates with each permutation of operation applications the set of final data states *final* and produced output parameters *ops_outpar*.

$$
\begin{array}{l}
OpSeqEffects == \{\mathcal{M} : Model;\ OPS, OpSet : \mathbb{F}\, NAME;\ current, final : Binding; \\
\qquad ops_inpar, ops_outpar : NAME \twoheadrightarrow Binding;\ perm : \mathbb{N} \twoheadrightarrow NAME \mid \\
\quad \mathrm{dom}\, ops_outpar = OpSet \land \\
\quad (\exists\, stseq : 1 \mathinner{..} \#OpSet + 1 \rightarrow Binding \bullet \\
\qquad stseq\, 1 = current \land stseq(\#OpSet + 1) = final \land \\
\qquad (\forall\, i : 1 \mathinner{..} \#OpSet \bullet \\
\qquad\quad (perm\, i, ops_inpar(perm\, i) \cup ops_outpar(perm\, i), \\
\qquad\qquad (stseq\, i, stseq(i + 1))) \in OpModel(\!| \{(\mathcal{M}, OPS)\} |\!))) \\
\bullet\, (((\mathcal{M}, OPS), perm, current, ops_inpar, OpSet), (final, ops_outpar))\}
\end{array}
$$

$$
\begin{array}{l}
OpSetNonInterference == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\, NAME;\ current : Binding; \\
\qquad ops_inpar : NAME \twoheadrightarrow Binding \mid \\
\quad \exists_1\, OpSet == \mathrm{dom}\, ops_inpar \bullet \\
\qquad (\forall\, op1, op2 : OpSet \mid op1 \neq op2 \bullet \\
\qquad\quad AssocSubspace(\mathcal{M}, OPS, op1) \cap AssocSubspace(\mathcal{M}, OPS, op2) = \varnothing) \land \\
\qquad (\forall\, perm1, perm2 : 1 \mathinner{..} \#OpSet \rightarrowtail\!\!\!\!\rightarrow OpSet \bullet \\
\qquad\quad OpSeqEffects(\!| \{((\mathcal{M}, OPS), perm1, current, ops_inpar, OpSet)\} |\!) \\
\qquad\quad = OpSeqEffects(\!| \{((\mathcal{M}, OPS), perm2, current, ops_inpar, OpSet)\} |\!) \\
\qquad\quad \neq \varnothing) \\
\bullet\, ((\mathcal{M}, OPS), (current, ops_inpar))\}
\end{array}
$$

We have to distinguish between the two approaches to relating events and operations discussed in Section 6.2. In the single-event approach, each operation application is related to a single execution event, including the record of the input and output parameter values. In the double-event approach, an operation application is split into the operation invocation, including the record of the input parameter values, and the operation termination, including the record of the output parameter values. The purpose of the set POP_APP is to distinguish between operation invocations, terminations and executions, which we call *partial operation events*.

$$
\begin{array}{ll}
Exec == \{1\} \times OP_APP & POP_APP == Exec \cup Invoc \cup Term \\
Invoc == \{2\} \times OP_APP & pop_name == first[NAME, Binding] \circ \\
Term == \{3\} \times OP_APP & \qquad second[\mathbb{N}, OP_APP] \\
 & pop_params == second[NAME, Binding] \circ \\
 & \qquad second[\mathbb{N}, OP_APP]
\end{array}
$$

Partial operation events are modelled as tuples, where the first component determines the kind of event.

The sets *Invocations*, *Terminations* and *Executions* define relations between a model $\mathcal{M}$, an operation identifier *op* and the set of all corresponding invocations, terminations and executions, respectively.

$$
\begin{aligned}
&\mathit{Invocations} == \{\mathcal{M} : \mathit{Model};\ \mathit{OPS} : \mathbb{F}\,\mathit{NAME};\ \mathit{op} : \mathit{NAME};\ \mathit{invoc} : \mathit{Binding} \mid \\
&\qquad \mathit{op} \in \mathit{OPS} \land \mathit{OPS} \subseteq \mathsf{dom}\,\mathcal{M} \land \\
&\qquad \mathsf{dom}\,\mathit{invoc} \subseteq \mathit{Inputs} \land \\
&\qquad (\exists\, \mathit{term}, \mathit{pre}, \mathit{post} : \mathit{Binding} \bullet \mathsf{dom}\,\mathit{term} \subseteq \mathit{Outputs} \land \\
&\qquad\quad (\mathit{op}, \mathit{invoc} \cup \mathit{term}, (\mathit{pre}, \mathit{post})) \in \mathit{OpModel}(\!| \{(\mathcal{M}, \mathit{OPS})\} |\!)) \\
&\bullet ((\mathcal{M}, \mathit{OPS}), (2, (\mathit{op}, \mathit{invoc})))\}
\end{aligned}
$$

The sets *Terminations* and *Executions* are defined analogously and are hence omitted.

History Model. The state-oriented semantics of the Z part of a concrete specification unit, as outlined in Section 9.2.2, is the set of all possible *histories* in which the specified artifact may participate. A history of an artifact at a particular point of its evolution, modelled by a member of the set *History*, is characterised by the sequence of partial operation events it has undergone and the sequence of states it has passed through. In addition, a history incorporates a component that is derived from the sequence of partial operation events, namely the corresponding sequence of (complete) operation applications. The derivation of this additional component is defined later.

Actually, we do not deal with real-time information in the current history model. However, in order to express the well-formedness conditions of partial operation event sequences in the following, we need a particular aspect of the timing of events: the information which partial operation events have occurred simultaneously. The set *PartialOpSeq* contains all pairs consisting of a sequence of partial operation events and a relation indicating which of the events of the sequence have occurred simultaneously.

$$
\begin{aligned}
&\mathit{PartialOpSeq} == \{\mathit{events} : \mathrm{seq}\,\mathit{POP_APP};\ \mathit{simultan} : \mathbb{N} \leftrightarrow \mathbb{N} \mid \\
&\quad \mathit{simultan} \subseteq \mathsf{dom}\,\mathit{events} \times \mathsf{dom}\,\mathit{events} \land \mathit{simultan} \in \mathit{EquivRel}[\mathbb{N}] \land \\
&\quad (\forall\, i, j : \mathsf{dom}\,\mathit{events} \mid (i, j) \in \mathit{simultan} \bullet \forall\, k : i \,..\, j \bullet (i, k) \in \mathit{simultan})\} \\
&\mathit{History} == \mathit{PartialOpSeq} \times (\mathrm{seq}\,\mathit{Binding} \times \mathrm{seq}\,\mathit{OP_APP})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{events_of} == \mathit{first}[\mathrm{seq}\,\mathit{POP_APP}, \mathbb{N} \leftrightarrow \mathbb{N}] \circ \mathit{first}[\mathit{PartialOpSeq}, \mathrm{seq}\,\mathit{Binding} \times \mathrm{seq}\,\mathit{OP_APP}] \\
&\mathit{simultan_of} == \mathit{second}[\mathrm{seq}\,\mathit{POP_APP}, \mathbb{N} \leftrightarrow \mathbb{N}] \circ \mathit{first}[\mathit{PartialOpSeq}, \mathrm{seq}\,\mathit{Binding} \times \mathrm{seq}\,\mathit{OP_APP}] \\
&\mathit{states_of} == \mathit{first}[\mathrm{seq}\,\mathit{Binding}, \mathrm{seq}\,\mathit{OP_APP}] \circ \mathit{second}[\mathit{PartialOpSeq}, \mathrm{seq}\,\mathit{Binding} \times \mathrm{seq}\,\mathit{OP_APP}] \\
&\mathit{ops_of} == \mathit{second}[\mathrm{seq}\,\mathit{Binding}, \mathrm{seq}\,\mathit{OP_APP}] \circ \mathit{second}[\mathit{PartialOpSeq}, \mathrm{seq}\,\mathit{Binding} \times \mathrm{seq}\,\mathit{OP_APP}]
\end{aligned}
$$

For a sequence of partial operation events to be a consistent record of the invocation, termination and execution of operations, it must satisfy different well-formedness conditions, which are introduced in the following. We introduce these well-formedness conditions in different steps, where each condition includes the conditions of the previous steps.

The first well-formedness condition concerns the relationship between the invocation and termination of operations within a sequence of partial operation events. Roughly speaking, an operation can terminate only if it has been invoked before. Formally speaking, for any prefix of a well-formed sequence, the number of invocations of a particular operation must be greater than or equal to the number of its terminations. Further, the difference between

the numbers must be lower than or equal to one, because at any time at most one instance of any operation can be invoked without being terminated. This is a consequence of the fact that any operation interferes with itself. The set $WellFormed_1$ contains all sequences satisfying this condition at least for the first wf events.

$$
\begin{array}{l}
WellFormed_1 == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\,NAME;\ h : History;\ wf : \mathbb{N} \mid \\
\quad \exists_1\, events == events_of\, h \bullet \\
\qquad (\forall\, i : 1 \mathrel{..} wf \bullet \\
\qquad\quad events\, i \in Invoc \land events\, i \in Invocations(\!|\{(\mathcal{M}, OPS)\}|\!) \\
\qquad\quad \lor events\, i \in Term \land events\, i \in Terminations(\!|\{(\mathcal{M}, OPS)\}|\!) \\
\qquad\quad \lor events\, i \in Exec \land events\, i \in Executions(\!|\{(\mathcal{M}, OPS)\}|\!)) \land \\
\qquad (\forall\, id : OPS;\ n : 1 \mathrel{..} wf \bullet \\
\qquad\quad 0 \leq \#\{i : 1 \mathrel{..} n \mid events\, i \in Invoc \land pop_name(events\, i) = id\} - \\
\qquad\quad \#\{i : 1 \mathrel{..} n \mid events\, i \in Term \land pop_name(events\, i) = id\} \leq 1) \\
\bullet\, ((\mathcal{M}, OPS), (h, wf))\}
\end{array}
$$

The second well-formedness condition deals with the relationship between the sequence of partial operation *events* and the corresponding sequence of data *states*, both being components of a history h. First, the state sequence must contain at least one data state, namely the initial state. Second, each termination/execution event of the component *events* is related to a corresponding pair of data states of the *states* component, so *states* must have exactly one member more than the projection of *events* to termination and execution events.

The function *ev_to_st* is introduced to map each event of the component *events* to the data state of the component *states* that is present when the event occurs (more precisely, their indices are related). The index of the component *states* by which this data state is determined depends on the number of termination and execution events that have occurred before.

$$
\begin{array}{l}
WellFormed_2 == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\,NAME;\ h : History;\ wf : \mathbb{N};\ ev_to_st : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\!+\;\; \mathbb{N} \mid \\
\quad (h, wf) \in WellFormed_1(\!|\{(\mathcal{M}, OPS)\}|\!) \land \\
\quad (\exists_1\, events == events_of\, h;\ states == states_of\, h \bullet \\
\qquad states \neq \langle\rangle \land \\
\qquad (\forall\, i : \mathbb{N} \setminus \{0\} \bullet i + 1 \in \mathrm{dom}\, states \Leftrightarrow i \in \mathrm{dom}(events \upharpoonright (Term \cup Exec))) \land \\
\qquad \mathrm{dom}\, ev_to_st = \mathrm{dom}\, events \land \\
\qquad (\forall\, i : \mathrm{dom}\, events \bullet \\
\qquad\quad ev_to_st\, i = \#\{j : \mathrm{dom}\, events \mid j < i \land events\, j \in Term \cup Exec\} + 1)) \\
\bullet\, ((\mathcal{M}, OPS), (h, wf, ev_to_st))\}
\end{array}
$$

The third well-formedness condition requires that the 'non-interference' conditions are met by the sequence of partial operation *events*. For any index of the sequence, the set of currently invoked or executed operations must satisfy the 'non-interference' conditions with respect to the current data state. The set of currently invoked and executed operations is determined by the auxiliary function *ConcOps*.

$$
\begin{aligned}
&ConcOps == \lambda\, OPS : \mathbb{F}\, NAME;\ h : History;\ index : \mathbb{N} \mid index \in \mathsf{dom}(events_of\ h) \bullet \\
&\quad \mathbf{let}\ events == events_of\ h;\ simultan == simultan_of\ h \bullet \\
&\quad \{op : OPS;\ inpar : Binding \mid \\
&\qquad \exists\, i : \mathsf{dom}\, events \mid op = pop_name(events\ i) \land inpar = Inputs \lhd pop_params(events\ i) \bullet \\
&\qquad\quad events\ i \in Exec \land (i, index) \in simultan \\
&\qquad\quad \lor \\
&\qquad\quad events\ i \in Invoc \land (i < index \lor (i, index) \in simultan) \land \\
&\qquad\quad \neg\, (\exists\, j : \mathsf{dom}\, events \mid events\ j \in Term \land pop_name(events\ j) = op \bullet \\
&\qquad\qquad i < j \land j < index \land (j, index) \notin simultan)\}
\end{aligned}
$$

$$
\begin{aligned}
&WellFormed_3 == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\, NAME;\ h : History;\ wf : \mathbb{N};\ ev_to_st : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{N} \mid \\
&\quad \exists_1\ events == events_of\ h;\ simultan == simultan_of\ h;\ states == states_of\ h \bullet \\
&\quad (h, wf, ev_to_st) \in WellFormed_2(\!|\{(\mathcal{M}, OPS)\}|\!) \land \\
&\quad (\forall\, i : 1 \mathrel{..} wf \bullet \\
&\qquad (states(ev_to_st\ i), ConcOps(OPS, h, i)) \in OpSetNonInterference(\!|\{(\mathcal{M}, OPS)\}|\!)) \\
&\bullet ((\mathcal{M}, OPS), (h, wf, ev_to_st))\}
\end{aligned}
$$

A sequence of partial operation events that is not well-formed models the divergence of the specified artifact, i.e., an arbitrary behaviour. We distinguish between two parts of a sequence: an initial part, which is well-formed, and the remaining part (possibly empty), which is divergent. In the above definitions, *wf* is an index expressing that the corresponding sequence is well-formed at least for the first *wf* events. In the following, we are interested only in the maximal values of this index.

$$
\begin{aligned}
&MaxWellFormed == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\, NAME;\ h : History;\ max_wf : \mathbb{N}; \\
&\qquad\qquad ev_to_st : \mathbb{N} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{N} \mid \\
&\qquad (h, max_wf, ev_to_st) \in WellFormed_3(\!|\{(\mathcal{M}, OPS)\}|\!) \land \\
&\qquad (\forall\, n : \mathbb{N} \mid (h, n, ev_to_st) \in WellFormed_3(\!|\{(\mathcal{M}, OPS)\}|\!) \bullet n \leq max_wf) \\
&\bullet ((\mathcal{M}, OPS), (h, max_wf, ev_to_st))\}
\end{aligned}
$$

The aim of the subsequent relation *InvTermComb* is to determine the sequence of (complete) operation applications (*operations*) that corresponds to a given sequence of partial operation events (*events*), i.e., to define the required relationship between the corresponding components of a history *h*.

$$
\begin{array}{l}
InvTermComb == \{\mathcal{M} : Model;\ OPS : \mathbb{F}\,NAME;\ h : History;\ max_wf : \mathbb{N};\ ev_to_st : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\mapsto \mathbb{N} \mid \\
\quad (h, max_wf, ev_to_st) \in MaxWellFormed(\!|\{(\mathcal{M}, OPS)\}|\!) \land \\
\quad (\exists_1\ events == events_of\ h;\ operations == ops_of\ h \bullet \\
\quad \mathrm{dom}\ operations = 1 \mathinner{..} \#\{i : 1 \mathinner{..} max_wf \mid events\ i \in Term \cup Exec\} \land \\
\quad (\exists_1\ term_exec_of : \mathrm{dom}\ operations \rightarrowtail \mathrm{dom}\ events \mid \\
\qquad (\forall\ i, j : \mathrm{dom}\ operations \mid i < j \bullet term_exec_of\ i < term_exec_of\ j) \land \\
\qquad \mathrm{ran}\ term_exec_of = \{i : 1 \mathinner{..} max_wf \mid events\ i \in Term \cup Exec\} \bullet \\
\qquad (\forall\ i : \mathrm{dom}\ operations \bullet \\
\qquad\quad events(term_exec_of\ i) \in Term \land \\
\qquad\quad op_name(operations\ i) = pop_name(events(term_exec_of\ i)) \land \\
\qquad\quad (\exists_1\ invoc == \max\{j : \mathrm{dom}\ events \mid j < term_exec_of\ i \land \\
\qquad\qquad\quad events\ j \in Invoc \land pop_name(events\ j) = op_name(operations\ i)\} \bullet \\
\qquad\qquad op_params(operations\ i) = \\
\qquad\qquad pop_params(events(term_exec_of\ i)) \cup pop_params(events\ invoc)) \\
\qquad\quad \lor \\
\qquad\quad events(term_exec_of\ i) \in Exec \land \\
\qquad\quad op_name(operations\ i) = pop_name(events(term_exec_of\ i)) \land \\
\qquad\quad op_params(operations\ i) = pop_params(events(term_exec_of\ i))))) \\
\bullet ((\mathcal{M}, OPS), (h, max_wf, ev_to_st))\}
\end{array}
$$

Each (complete) operation application of *operations* is composed either of an invocation and the corresponding termination event or of a single execution event of *events*. Since an operation application is completed only with the occurrence of the termination/execution event, each operation application in *operations* is identified with its corresponding termination/execution event of *events* by means of the injective, monotone function $term_exec_of$. The pairing of corresponding invocation and termination events is done only for the well-formed part of the sequence ($1 \mathinner{..} max_wf$). This is important because the derivation of the possible data state evolutions in later definitions are based on the sequence of (complete) operation applications.

The set *Histories* relates each model $\mathcal{M}$ of the meaning of the Z part of a concrete specification unit to the set of histories of the specified artifact. The component *events* of a history records operation invocations, terminations and executions. The derived component *operations* is the sequence of operation applications resulting from the combination of corresponding invocation and termination events of the former component, as defined by the previous relation.

$$
\begin{array}{l}
\mathit{Histories} == \{\mathcal{M} : \mathit{Model};\ \mathit{OPS} : \mathbb{F}\,\mathit{NAME};\ h : \mathit{History} \mid \\
\quad \exists_1\ \mathit{events} == \mathit{events_of}\ h;\ \mathit{states} == \mathit{states_of}\ h;\ \mathit{operations} == \mathit{ops_of}\ h; \\
\qquad \mathit{max_wf} : \mathbb{N};\ \mathit{ev_to_st} : \mathbb{N} \twoheadrightarrow \mathbb{N} \mid \\
\qquad\quad (h, \mathit{max_wf}, \mathit{ev_to_st}) \in \mathit{InvTermComb}(\!|\{(\mathcal{M}, \mathit{OPS})\}|\!) \bullet \\
\qquad (\forall i : \mathrm{dom}\,\mathit{operations} \cup \{0\} \bullet \mathit{states}(i+1) \in \mathit{StateModel}(\!|\{\mathcal{M}\}|\!)) \wedge \\
\qquad \mathit{states}\ 1 \in \mathit{InitModel}(\!|\{\mathcal{M}\}|\!) \wedge \\
\qquad (\forall i : 1 \mathrel{..} \mathit{max_wf} \mid \mathit{events}\ i \in \mathit{Invoc} \cup \mathit{Exec} \bullet \\
\qquad\quad (\exists \mathit{post}, \mathit{out} : \mathit{Binding} \mid \mathrm{dom}\,\mathit{out} \subseteq \mathit{Outputs} \bullet \\
\qquad\qquad (\mathit{pop_name}(\mathit{events}\ i), (\mathit{Inputs} \lhd \mathit{pop_params}(\mathit{events}\ i)) \cup \mathit{out}, \\
\qquad\qquad\quad (\mathit{states}(\mathit{ev_to_st}\ i), \mathit{post})) \in \mathit{OpModel}(\!|\{(\mathcal{M}, \mathit{OPS})\}|\!))) \wedge \\
\qquad (\forall i : \mathrm{dom}\,\mathit{operations} \bullet \\
\qquad\quad (\mathit{op_name}(\mathit{operations}\ i), \mathit{op_params}(\mathit{operations}\ i), \\
\qquad\qquad (\mathit{states}\ i, \mathit{states}\ (i+1))) \in \mathit{OpModel}(\!|\{(\mathcal{M}, \mathit{OPS})\}|\!)) \\
\bullet\ ((\mathcal{M}, \mathit{OPS}), h)\}
\end{array}
$$

A tuple consisting of a sequence of partial operation events (*events*), a sequence of data states (*states*) and a sequence of operation applications (*operations*) is related to a model $\mathcal{M}$ under the following conditions. First, all data states in the well-formed part of the *states* sequence must satisfy the state invariant, and the first data state must additionally fulfil the initial state predicate. Second, for any invocation or execution event in *events*, the data state in *states* that is present when the event occurs (determined with the help of *ev_to_st*) must satisfy the precondition of the corresponding operation schema.[3] Third, for each operation application in *operations*, the state pair that is constituted by the data state that is present before the termination/execution event occurs and the data state that is the result of the termination/execution must be a member of the operation model, i.e., be related by the corresponding operation schema.

Note, again, that the sequence of partial operation events of a history needs not be completely well-formed; the evolution of the data state corresponding to the diverging part of such a sequence is arbitrary. This is a consequence of the definition of *InvTermComb*, in which only the well-formed part of a sequence of partial operation events is used to build the corresponding sequence of operation applications. Only the latter sequence is the basis for expressing constraints on the data state evolution in the set *Histories*.

Finally, the relation *PreHist* relates two histories if, and only if, the first is a prefix of the second one (component-wise).

$$
\begin{array}{l}
\mathit{PreHist} == \{h1, h2 : \mathit{History} \mid \\
\quad \mathit{events_of}\ h1\ \mathsf{prefix}\ \mathit{events_of}\ h2 \wedge \\
\quad \mathit{states_of}\ h1\ \mathsf{prefix}\ \mathit{states_of}\ h2 \wedge \\
\quad \mathit{simultan_of}\ h1 = (\mathit{simultan_of}\ h2) \cap (\mathrm{dom}(\mathit{events_of}\ h1) \times \mathrm{dom}(\mathit{events_of}\ h1))\}
\end{array}
$$

For each pair of histories $(h1, h2) \in \mathit{PreHist}$, $h1$ is called a *pre-history* of $h2$.

[3] Note that this condition is necessary in addition to the last condition, because the data state present when an execution or invocation event occurs needs not be identical to the data state to which the operation is applied.

This completes the description of the history model, which is depicted in Figure 9.4 by the rectangle with the corresponding name.

9.2.4 Extended Timed Failures Model (ETFM)

The purpose of the extended timed failures model, which is depicted in Figure 9.4 by the rectangle in the centre, is

- to define the constituents (notions) of the timed failures model within ZF set theory,
- to extend the syntax of timed CSP in order to accommodate the needs of the CSP part in the context of the Z definitions within a concrete specification unit and
- to define the semantics of the extended syntax based on the denotational semantics of timed CSP as given in [Schneider, 1999b].

We base RT-Z on the *finite* timed failures model, because it is not reasonable to deal with *infinite* evolutions of the data state: there is simply no possibility to characterise the data state of the artifact under consideration after an infinite sequence of operation applications has occurred. As a consequence, we deal only with finite timed observations (timed traces).

Time Domain

The time domain of timed CSP processes are the positive real numbers. Although real numbers are not primitive objects of ZF set theory, it is well-known (c.f. [Hamilton, 1982, Chapter 1]) how to construct $\mathbb{R}$ from $\mathbb{N}$ via $\mathbb{Z}$ and $\mathbb{Q}$. First, the natural numbers are defined by virtue of the ZF axioms to be specific kinds of sets, and it is shown that these "abstract natural numbers" satisfy all required laws (e.g., Peano axioms). Then, integers are defined as equivalence classes over pairs of natural numbers with identical difference, rational numbers are defined as equivalence classes over pairs of integers with identical quotient, and real numbers are defined as equivalence classes over Cauchy sequences of rational numbers converging at the identical limit. In [Hamilton, 1982] it is demonstrated that real numbers constructed in this way exhibit all required properties, and it is shown how to define the basic constants, relations and functions on real numbers on the grounds of rational numbers and limits of Cauchy sequences. We assume in the following that the tuple ($\mathcal{R}$, $0_{\mathcal{R}}$, $1_{\mathcal{R}}$, $+_{\mathcal{R}}$, $*_{\mathcal{R}}$, $<_{\mathcal{R}}$) is defined accordingly within ZF set theory.

The Z part of each RT-Z specification unit is based on the Z section *Reals*, which is depicted in Figure 9.5, via the parent relationship of Standard Z. That is, the section *Reals* is an implicit prelude for the Z part of any RT-Z specification unit (analogously to the mathematical toolkit). Actually, this prelude introduces only the *symbols* of the carrier set, its elements, its functions and its relation without any definition. The mapping of these symbols to the corresponding entities of the semantic universe ($\mathcal{R}$, $0_{\mathcal{R}}$, $1_{\mathcal{R}}$, $+_{\mathcal{R}}$, $*_{\mathcal{R}}$, $<_{\mathcal{R}}$) is fixed only on the semantic level of RT-Z: the models of the Z part taken into account when computing the meaning of an RT-Z specification unit are restricted to those that map the above symbols to the intended entities of the semantic universe. This is reflected by the definition of the semantic functions $\mathit{timedfailures\,states}_C\,[\![_]\!]$ and $\mathit{timedfailures\,states}_A\,[\![_]\!]$ on pp. 145 and 147,

section *Reals*

$[\mathbb{R}]$

$$
\begin{array}{|l}
0_{\mathbb{R}}, 1_{\mathbb{R}} : \mathbb{R} \\
_ +_{\mathbb{R}} _, _ *_{\mathbb{R}} _ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
_ <_{\mathbb{R}} _ : \mathbb{R} \leftrightarrow \mathbb{R}
\end{array}
$$

$$
\begin{array}{|l}
_ >_{\mathbb{R}} _ : \mathbb{R} \leftrightarrow \mathbb{R} \\
_ \leqslant_{\mathbb{R}} _ : \mathbb{R} \leftrightarrow \mathbb{R} \\
_ \geqslant_{\mathbb{R}} _ : \mathbb{R} \leftrightarrow \mathbb{R} \\
_ -_{\mathbb{R}} _ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
\inf : \mathbb{P}\,\mathbb{R} \rightarrow \mathbb{R} \\
\sup : \mathbb{P}\,\mathbb{R} \rightarrow \mathbb{R} \\
\hline
(\forall x, y, z : \mathbb{R} \bullet \\
\quad x >_{\mathbb{R}} y \Leftrightarrow y <_{\mathbb{R}} x \wedge \\
\quad x \leqslant_{\mathbb{R}} y \Leftrightarrow \neg\, y <_{\mathbb{R}} x \wedge \\
\quad x \geqslant_{\mathbb{R}} y \Leftrightarrow \neg\, x <_{\mathbb{R}} y \wedge \\
\quad x -_{\mathbb{R}} y = z \Leftrightarrow z +_{\mathbb{R}} y = x) \\
\forall S : \mathbb{P}\,\mathbb{R};\ x : \mathbb{R} \bullet \inf S = x \Leftrightarrow \\
\quad (\forall y : S \bullet x \leqslant_{\mathbb{R}} y) \wedge (\forall z : \mathbb{R} \mid (\forall y : S \bullet z \leqslant_{\mathbb{R}} y) \bullet z \leqslant_{\mathbb{R}} x) \\
\forall S : \mathbb{P}\,\mathbb{R};\ x : \mathbb{R} \bullet \sup S = x \Leftrightarrow \\
\quad (\forall y : S \bullet y \leqslant_{\mathbb{R}} x) \wedge (\forall z : \mathbb{R} \mid (\forall y : S \bullet y \leqslant_{\mathbb{R}} z) \bullet x \leqslant_{\mathbb{R}} z)
\end{array}
$$

Figure 9.5: Z section *Reals*.

where only a subset of the models of the meaning of the Z part are taken into consideration. The following expression is only a definition fragment used to illustrate the above statement.

$$
\begin{array}{l}
\ldots \mathcal{M} : [\![\ \ldots\ ]\!]^{\mathcal{Z}} \mid \\
\quad \ldots \\
\quad \{\mathbb{R}, 0_{\mathbb{R}}, 1_{\mathbb{R}}, +_{\mathbb{R}}, *_{\mathbb{R}}, <_{\mathbb{R}}\} \subseteq \mathsf{dom}\, \mathcal{M} \wedge \\
\quad \mathcal{M}\, \mathbb{R} = \mathcal{R} \wedge \mathcal{M}\, 0_{\mathbb{R}} = 0_{\mathcal{R}} \wedge \ldots \\
\quad \ldots
\end{array}
$$

Based on these considerations, the time domain of RT-Z is defined to be the positive real numbers including zero.

$$
\begin{array}{l}
\mathbb{T} == \{x : \mathcal{R} \mid \neg\, x <_{\mathcal{R}} 0_{\mathcal{R}}\} \\
\mathbb{T}^{+} == \{x : \mathcal{R} \mid 0_{\mathcal{R}} <_{\mathcal{R}} x\}
\end{array}
$$

Constituents

In the remainder of this chapter, we use functions and relations working on objects of the timed failures model introduced in [Schneider, 1999b]. They are explained in Appendix E.

In RT-Z specifications, we have to distinguish between two different kinds of events. The first kind of events, which we call *operation-related events*, are related to the invocation, termination and execution of operations. The values transmitted by these events are the values of input and output parameters. The second kind of events are called *plain events* and concern the internal and external synchronisation and communication. These events are pairs of channel identifiers and communicated values, too. The set *ChannelValue* encompasses all values that can be communicated along channels, either parameters of an operation invocation, termination or execution or arbitrary values of plain events.[4]

$$ChannelValue == \mathbb{W}$$

Let the functions *InvOf*, *TermOf* and *ExecOf* map each operation identifier to the identifier of the channel carrying the corresponding invocation, termination and execution events, respectively. The set *ORC* (abbreviates 'operation-related channels') encompasses all identifiers of channels carrying operation-related events.

$$ORC == \operatorname{ran} InvOf \cup \operatorname{ran} TermOf \cup \operatorname{ran} ExecOf$$

An event is a pair of a channel identifier (*CHANNEL*) and a communicated value (*ChannelValue*). According to the semantic models of CSP, a trace is a finite sequence of events, a refusal is a set of events, and a failure is a pair consisting of a trace and a refusal, where the observed process has engaged in the events recorded in the trace and is subsequently able to deadlock if the environment is willing to offer only the events contained in the refusal set.

$$Event == CHANNEL \times ChannelValue$$

$$chan_of == first[CHANNEL, ChannelValue]$$
$$val_of == second[CHANNEL, ChannelValue]$$

$$Trace == \{s : \operatorname{seq} Event \mid \forall i : 1 \mathrel{..} \#s - 1 \bullet s\,i \neq \checkmark\}$$
$$Refusal == \mathbb{P}\, Event$$
$$Failure == Trace \times Refusal$$

According to the timed failures model of timed CSP [Schneider, 1999b], a timed event is a pair consisting of an event and the time instant it is observed. A timed trace is a finite or infinite sequence of timed events subject to the following conditions. The sequence of timed events recorded in a timed trace must be ordered in time, the termination event (✓) is allowed to appear only as the last event in a finite timed trace and any infinite trace must not

[4] This is a place where we must "circumvent" the type system of Z: the values communicated with operation-related events are bindings, which in the untyped ZF set theory can be required to be contained in $\mathbb{W}$. This is not possible in the strongly typed Z notation. We have circumvented the Z type system by means of invisible (in LaTeX) conversion functions.

be bounded in time, which is dictated by the principle of finite variability.[5] In the context of the finite timed failures model, however, we consider only finite timed traces.

$$TimedEvent == \mathbb{T} \times Event \qquad time_of == first[\mathbb{T}, Event]$$
$$ev_of == second[\mathbb{T}, Event]$$

$$\begin{aligned}
&TimedTrace == \{s : \mathrm{seq}_{\infty}[TimedEvent] \mid \\
&\quad (\forall i, j : \mathsf{dom}\, s \mid i < j \bullet time_of(s\,i) \leqslant_{\mathbb{R}} time_of(s\,j)) \land \\
&\quad (\exists n : \mathbb{N} \bullet \mathsf{dom}\, s = 1 \mathinner{..} n) \land \\
&\qquad \checkmark \notin \{i : 1 \mathinner{..} \#s - 1 \bullet ev_of(s\,i)\} \\
&\quad \lor \mathsf{dom}\, s = \mathbb{N} \land \\
&\qquad (\forall t : \mathbb{T} \bullet \exists t^* : \mathbb{T};\ e : Event \mid t <_{\mathbb{R}} t^* \bullet (t^*, e) \in \mathsf{ran}\, s) \land \\
&\qquad \checkmark \notin \{i : \mathbb{N} \bullet ev_of(s\,i)\}\}
\end{aligned}$$

To define the concept of a timed refusal, we need the set *HalfOpenInterval*, containing all bounded half-open intervals of time instants, the set *RefusalToken*, containing all pairs of time intervals and event sets, and the set *FinRefusalTokenSet*, which contains all unions of a finite number of refusal tokens. Using these sets, a timed refusal is defined to be a set of timed events such that its restriction to any finite interval is a member of *FinRefusalTokenSet*, which is again dictated by the principle of finite variability.

$$\begin{aligned}
&HalfOpenInterval == \{TI : \mathbb{P}\,\mathbb{T} \mid \exists b, e : \mathbb{T} \mid 0_{\mathbb{R}} \leqslant_{\mathbb{R}} b \land b <_{\mathbb{R}} e \bullet TI = \{t : \mathbb{T} \mid b \leqslant_{\mathbb{R}} t \land t <_{\mathbb{R}} e\}\} \\
&RefusalToken == \{S : \mathbb{P}\, TimedEvent \mid \exists I : HalfOpenInterval;\ A : \mathbb{P}\, Event \bullet S = I \times A\} \\
&FinRefusalTokenSet == \{tref : \mathbb{P}\, TimedEvent \mid \exists C : \mathbb{F}\, RefusalToken \bullet tref = \bigcup C\} \\
&TimedRefusal == \{tref : \mathbb{P}\, TimedEvent \mid \forall t : \mathbb{T} \bullet \\
&\quad \{t' : \mathbb{T};\ e : Event \mid (t', e) \in tref \land t' <_{\mathbb{R}} t\} \in FinRefusalTokenSet\}
\end{aligned}$$

A timed failure is a pair of a timed trace and a timed refusal.

$$TimedFailure == TimedTrace \times TimedRefusal$$

In Section A.2, we discuss in detail the properties TF1–TF3 that a set of timed failures must meet in order to represent some timed CSP process. To understand the following definition of the set *TimedFailuresModel*, which encompasses all sets of timed failures that satisfy these conditions, read the informal explanation on p. 249.

[5] The principle of finite variability requires a process to undergo only finitely many changes in any finite time interval.

$$
\begin{array}{l}
\mathit{TimedFailuresModel} == \{S : \mathbb{P}\,\mathit{TimedFailure} \mid \\
\quad (\langle\rangle, \varnothing) \in S \qquad\qquad \text{(TF1)} \\
\quad \land \\
\quad (\forall s, s' : \mathit{TimedTrace};\ \aleph, \aleph' : \mathit{TimedRefusal} \bullet \\
\qquad (s, \aleph) \in S \land (s', \aleph') \preceq (s, \aleph) \Rightarrow (s', \aleph') \in S) \qquad\qquad \text{(TF2)} \\
\quad \land \\
\quad (\forall s : \mathit{TimedTrace};\ \aleph : \mathit{TimedRefusal} \mid (s, \aleph) \in S \bullet \qquad\qquad \text{(TF3)} \\
\qquad (\exists \aleph' : \mathit{TimedRefusal} \mid \aleph \subseteq \aleph' \land (s, \aleph') \in S \bullet \\
\qquad\quad (\forall t : \mathbb{T};\ a : \mathit{Event} \bullet \\
\qquad\qquad ((t, a) \notin \aleph' \Rightarrow ((s \upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \parallel\!\upharpoonright_{tref} t) \in S) \qquad \text{(C1)} \\
\qquad\qquad \land \\
\qquad\qquad t >_{\mathbb{R}} 0_{\mathbb{R}} \land \neg\, (\exists \epsilon : \mathbb{T}^{+} \bullet \{t^{*} : \mathbb{T} \mid t -_{\mathbb{R}} \epsilon \leqslant_{\mathbb{R}} t^{*} \land t^{*} <_{\mathbb{R}} t\} \times \{a\} \subseteq \aleph') \\
\qquad\qquad \Rightarrow ((s \parallel\!\upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \parallel\!\upharpoonright_{tref} t) \in S)))\} \qquad \text{(C2)}
\end{array}
$$

The relation $\preceq$ occurring in the above definition represents the information order on timed failures.

$$
\begin{array}{l}
_ \preceq _ == \{s, s' : \mathit{TimedTrace};\ \aleph, \aleph' : \mathit{TimedRefusal} \mid s' \in \mathit{FinTimedTrace} \land \\
\quad (\exists s2 : \mathit{TimedTrace} \bullet s = s' \frown s2 \land \aleph' \subseteq \aleph \parallel\!\upharpoonright_{tref} \mathrm{begin}_{ttr}(s2)) \\
\bullet ((s', \aleph'), (s, \aleph))\}
\end{array}
$$

The timed trace component s' of the first timed failure must be a prefix of the timed trace component s of the second one, and its timed refusal component $\aleph'$ may contain only refusal information also present in the timed refusal component $\aleph$ of the second timed failure up to the time instant at which s extends s'.

Extended Timed Failures Semantics of Z Specifications

As discussed, the main idea underlying the semantic integration of Z and timed CSP is to give a Z specification a timed failures semantics. This allows us to regard the Z part and the CSP part of a concrete specification unit as two parallel processes. The first step towards this goal, achieved in Section 9.2.3, was to associate a set of histories with each model of the meaning of a Z specification. The second step, which is the subject of the current section, is to translate the concept of a history into the concepts of (untimed) CSP, namely traces, refusals and failures.

We first need to define the auxiliary function *ev_of_op*. It maps an operation application to the corresponding event. An operation application, as recorded in a history, corresponds to an event if its operation identifier translates into the channel identifier of the event and if the values transmitted with the operation application and the event occurrence are identical, each embedded in the respective context.

$$
\begin{aligned}
&ev_of_op == \lambda\, op : POP_APP \bullet \\
&\quad \mathbf{let}\ chan == (\mathbf{if}\ op \in Invoc\ \mathbf{then}\ InvOf(pop_name\ op) \\
&\qquad\qquad\qquad \mathbf{else\ if}\ op \in Term\ \mathbf{then}\ TermOf(pop_name\ op) \\
&\qquad\qquad\qquad \mathbf{else}\ ExecOf(pop_name\ op)) \bullet \\
&\quad (chan, (pop_params\ op))
\end{aligned}
$$

The functions *InvOf*, *TermOf* and *ExecOf* map each operation identifier to the identifier of the channel carrying the corresponding event; they have been introduced on p. 124.

The function $traces_Z$ maps histories of the Z part to corresponding traces of operation-related events, which represent possible observations of the CSP interpretation of the Z part. Basically, a trace is mapped to a history if it corresponds to the sequence of partial operation events contained in the history. Both the trace and the history event sequence must have the same length, and for each index of the sequence the operation application and the trace event must be related by *ev_of_op*.

$$traces_Z == \lambda\, h : History \bullet (\lambda\, i : \mathsf{dom}(events_of\ h) \bullet ev_of_op((events_of\ h)i))$$

Failures of operation-related events, which represent possible observations of the CSP interpretation of the Z part, and histories of the Z part are related by $failures_Z$. The trace component *tr* of these failures is determined by $traces_Z$. Regarding the refusals *ref* related to a particular trace *tr* there are two reasons for the membership of an operation-related event *e* in *ref*.

The first reason for the refusal of the event *e* is that the modelled artifact is not able to perform *e* after having performed the operation-related events in *tr*. The inability to perform an operation-related event corresponds to the inability of the modelled artifact to invoke/execute this operation if the operation's precondition is not satisfied with respect to the current data state. Technically speaking, an event cannot be performed after a particular trace has been observed if the concatenation of this trace and the event is not a possible observation of any history of which the considered history is a pre-history.

The other reason for an operation-related event *e* to be a member of *ref* (to be refused) is that there is another operation-related event e' that is not a member of *ref* (not refused) and that is related to *e* such that both events represent the termination or execution of the same operation. If they represent the execution of an operation, then they must assign the same values to the input parameters of this operation. The background of this second alternative is our decision that the Z part chooses the values of the output parameters nondeterministically; they are constrained only by the definition of the Z operation schema. Therefore, given a set of valid assignments of values to the output parameters of an operation, the Z part must be free to refuse all but one assignment.

$$
\begin{array}{l}
failures_Z == \{h : History;\ hset : \mathbb{P}\, History;\ fail : Failure \mid \\
\quad \exists_1\, tr == fail.1;\ ref == fail.2 \bullet \\
\quad\quad tr = traces_Z\, h \wedge \\
\quad\quad (\forall\, e : ref \bullet \\
\quad\quad\quad tr \frown \langle e \rangle \notin \{ext : hset \mid (h, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad\quad \vee \\
\quad\quad\quad (\exists\, e' : Event \mid e' \notin ref \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad (chan_of(e) \in \mathrm{ran}\, TermOf \\
\quad\quad\quad\quad \vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \\
\quad\quad\quad\quad\quad \wedge\, Inputs \lhd ((val_of\ e)) = Inputs \lhd ((val_of\ e'))))) \\
\bullet ((h, hset), fail)\}
\end{array}
$$

Note that we need the set of all histories *hset* that are possible observations of the specified artifact in order to define the failures corresponding to a history *h*, because the refusal information refers to the inability to expand the considered history *h* by another valid history.

Timed failures of operation-related events, which represent possible timed observations of the timed CSP interpretation of the Z part, and histories of the Z part are related by the set $timedfailures_Z$. To understand the association of timed failures with histories, it is important to realise that the Z part does not fix any time aspects of the occurrence or refusal of operation-related events. A (finite) timed failure is related to a history *h* under the following conditions. First, the (untimed) trace that is obtained by ignoring the time information of the timed trace component must be mapped to *h* by $traces_Z$. Second, considering an arbitrary time instant *t*, the set of events refused at that time instant (member of the timed refusal) must be a refusal associated with the (untimed) trace that is obtained by restricting the timed trace to the interval ending at time instant *t* (inclusive) and subsequently ignoring the time information.

$$
\begin{array}{l}
timedfailures_Z == \{h : History;\ hset : \mathbb{P}\, History;\ tf : TimedFailure \mid \\
\quad \exists_1\, s == tf.1;\ \aleph == tf.2;\ simultan == simultan_of\ h \bullet \\
\quad\quad (\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan \bullet time_of(s\, i) = time_of(s\, j)) \wedge \\
\quad\quad \mathrm{strip}\ s = traces_Z\, h \wedge \\
\quad\quad (\forall\, t : \mathbb{T} \bullet \\
\quad\quad\quad (\exists_1\, tr == \mathrm{strip}(s \upharpoonright_{ttr} t);\ ref == \{e : Event \mid (t, e) \in \aleph\} \bullet \\
\quad\quad\quad \exists\, prefix : hset \mid (prefix, h) \in PreHist \bullet \\
\quad\quad\quad\quad (tr, ref) \in failures_Z(\!|\{(prefix, hset)\}|\!))) \\
\bullet ((h, hset), tf)\}
\end{array}
$$

The time information contained in the timed trace component *s* must be consistent with the information of the component *simultan* of the history *h* concerning the simultaneity of operation-related events.

In Appendix B, we prove that the above definitions really map each Z specification to a timed CSP process, i.e., to a set of timed failures satisfying the properties of the timed failures model as defined on p. 126.

Process Term Language

According to the description in Chapter 7, the syntax of the CSP part of concrete specification units slightly extends the syntax of timed CSP processes in order to incorporate the following adaptations.

- Each application of the channel input operator ($c?(x : S) \rightarrow P(x)$) is parametrised by a Z set expression S, which is evaluated in the context of the Z part.
- Each instance of the channel output operator ($c!v \rightarrow P$) is parametrised by a Z value expression v, which is evaluated in the context of the Z part.
- The index sets used as parameters of the indexed nondeterministic choice ($\sqcap_{id \in S}$), interleaving ($|||_{id \in S}$) and synchronised parallel ($\|_{id \in S}$) operators are evaluated in the context of the Z part.
- The real-time operators of timed CSP (timeout ($\triangleright\{t\}$), timed interrupt ($\triangle_t$) and delay ($Wait\ t$)) have time parameters whose values are evaluated in the context of the Z part.

Let *ETCSP_PROCESS* denote the set of extended process terms as specified by the syntax definition in Appendix D.

Process terms in the `BEHAVIOUR` section of concrete specification units can refer to arbitrary channel identifiers declared in the `LOCAL` and `INTERFACE` sections, and they can contain arbitrary time, value and set expressions, which are interpreted in the context of the corresponding Z part. The CSP definitions of concrete specification units can hence be interpreted only in the context of three pieces of information:

- The assignment of value domains to those channel identifiers to which the CSP definitions refer. Such an assignment is characterised by a member of *ChanSgn*.
- The assignment of values to the value, set and time expressions to which the CSP definitions refer. Such an assignment is determined by a model, being a member of the denotation of the corresponding Z part.
- The assignment of timed CSP expressions to free process variables occurring in the process terms. Such an assignment is a member of *ProcEnv*.

The semantic function of the extended timed failures model for process terms has the following signature.

$$timedfailures_{ETCSP}\,[\![_]\!]_ \; : ETCSP_PROCESS \times (ProcEnv \times (Model \times Model) \times ChanSgn) \rightarrow TimedFailuresModel$$

There is a *pair* of parameters of type *Model* for the following reason. Some process operators, e.g., channel input, update the information that is available for evaluating Z expressions. The channel input operator binds the communicated value to its variable, to which Z expressions in the subsequent process expression can refer. The current information used to evaluate Z expressions—including variable bindings—is recorded in the first parameter of the pair.

The scope in which variable bindings are visible is limited. In the process expression,

$P \mathrel{\widehat=} c?x : S \rightarrow d!x \rightarrow Q$

e.g., the variable x, bound by the communication on the channel c, can be referred to only in the process expression subsequent to the channel input: it can be referred to in the channel output but it cannot be used in order to evaluate the process term Q. Variables are visible only in the process term in which they are bound. Whenever a process is instantiated, all variable bindings are discarded. That is, all processes are evaluated with respect to the identical model, which encodes the information of the Z part without containing any variable bindings. This model is called the initial model, and it is the second parameter of the pair.

The definition of the semantic function $timedfailures_{ETCSP}$ is based on the semantic function of the timed failures model, which we define in Section A.2. In the following, we provide the definitions of the extended/adapted operators. The meaning of these operators is defined in the form

$timedfailures_{ETCSP} [\![P(par1, \ldots, parN)]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \ldots$

where the triple $(Env, (\mathcal{M}, \mathcal{M}_{\mathcal{I}}), IL)$ contains the underlying information of the Z part and the `INTERFACE` and `LOCAL` sections needed to interpret the process definition and where $par1, \ldots, parN$ are simple parameters of the process definition, e.g., the channel name of the prefix operator or the index set of the indexed nondeterministic choice.

Channel Input Operator: The channel input operator $chan?(var : Set) \rightarrow P$ has three parameters: the channel identifier $chan$, the identifier var, to which the value that is transmitted with the event occurrence is bound, and the set expression Set, which restricts the set of values that can be communicated with the event occurrence.

The denotation assigned to the channel input operator is split into three subsets. The first subset deals with an inconsistent use of the parameters: either the channel identifier is not declared as an external or internal channel or the set associated with the expression Set is not a subset of the set associated with the considered channel by the channel declaration. In these cases, the semantics refrains from defining the meaning of the operator: it just allows an arbitrary behaviour.

The second and third subsets reflect, in essence, the definition of the channel input operator in timed CSP. The second subset addresses the case that the initial event has already occurred, while the third subset addresses the case that this initial event is still awaited.

$$
\begin{array}{l}
timedfailures_{ETCSP} [\![chan?(var : Set) \rightarrow P]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \\
\quad \{ tf : TimedFailure \mid chan \notin \mathrm{dom}\, IL \vee chan \in \mathrm{dom}\, IL \wedge \\
\qquad\qquad \neg\ [\![Set]\!]^{\epsilon}\, \mathcal{M} \subseteq [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M} \} \\
\quad \cup \{ s : TimedTrace;\ \aleph : TimedRefusal;\ t : \mathbb{T};\ val : \mathbb{W} \mid \\
\qquad\qquad chan \in \mathrm{dom}\, IL \wedge [\![Set]\!]^{\epsilon}\, \mathcal{M} \subseteq [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M} \wedge val \in ([\![Set]\!]^{\epsilon}\, \mathcal{M}) \wedge \\
\qquad\qquad (s, \aleph) \mathbin{\dot{-}}_{tfail} t \in timedfailures_{ETCSP} [\![P]\!]_{(Env,(\mathcal{M} \oplus \{var \mapsto val\},\mathcal{M}_{\mathcal{I}}),IL)} \\
\qquad \bullet (\langle (t, (chan, val)) \rangle \frown s, \aleph) \} \\
\quad \cup \{ \aleph : TimedRefusal \mid chan \in \mathrm{dom}\, IL \wedge [\![Set]\!]^{\epsilon}\, \mathcal{M} \subseteq [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M} \wedge \\
\qquad\qquad \{ t : \mathbb{T};\ val : \mathbb{W} \mid val \in [\![Set]\!]^{\epsilon}\, \mathcal{M} \bullet (t, (chan, val)) \} \cap \aleph = \varnothing \\
\qquad \bullet (\langle \rangle, \aleph) \}
\end{array}
$$

In the definition of the second subset, it is first required that all parameters are used in a consistent way. The data value *val* transmitted with the event occurrence must be a member of the set associated with the expression *Set* by the Z part, and the value *val* is bound to the variable *var*. This procedure of binding the chosen value to the variable is reflected in the definition of the model used to interpret the remaining process expression *P*: the original model $\mathcal{M}$ is updated with the pair $var \mapsto val$ of the variable and the transmitted value.

Channel Output Operator: Again, the denotation of the channel output operator $chan!val \rightarrow P$ is split into three subsets, where the structure is identical to the case of the channel input operator above. The substantial difference is the treatment of the value *val* to be transmitted: since it is now the considered process that determines the concrete value transmitted along the respective channel, the remaining process *P* does not depend on this value and thus the model used to interpret *P* need not be updated like in the above case.

$$
\begin{aligned}
& timedfailures_{ETCSP} [\![chan!val \rightarrow P]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \\
& \quad \{tf : TimedFailure \mid chan \notin \mathrm{dom}\, IL \vee chan \in \mathrm{dom}\, IL \wedge \\
& \qquad\qquad ([\![val]\!]^{\epsilon}\, \mathcal{M}) \notin [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M}\} \\
& \quad \cup \{s : TimedTrace;\ \aleph : TimedRefusal;\ t : \mathbb{T} \mid \\
& \qquad\qquad chan \in \mathrm{dom}\, IL \wedge ([\![val]\!]^{\epsilon}\, \mathcal{M}) \in [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M} \wedge \\
& \qquad\qquad (s, \aleph) \mathbin{\dot{-}}_{tfail} t \in timedfailures_{ETCSP} [\![P]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} \\
& \qquad \bullet (\langle (t, (chan, ([\![val]\!]^{\epsilon}\, \mathcal{M}))) \rangle \frown s, \aleph)\} \\
& \quad \cup \{\aleph : TimedRefusal \mid chan \in \mathrm{dom}\, IL \wedge ([\![val]\!]^{\epsilon}\, \mathcal{M}) \in [\![IL\, chan]\!]^{\epsilon}\, \mathcal{M} \wedge \\
& \qquad\qquad \{t : \mathbb{T} \bullet (t, (chan, ([\![val]\!]^{\epsilon}\, \mathcal{M})))\} \cap \aleph = \varnothing \\
& \qquad \bullet (\langle\rangle, \aleph)\}
\end{aligned}
$$

Indexed Nondeterministic Choice: The index set expression *S* of an indexed nondeterministic choice $\sqcap_{id \in S}$ is interpreted in the context of the Z part.

The definition of the meaning of this operator is divided into two subsets. The first subset addresses the inconsistent use of the index set parameter, in which the Z part assigns the empty set to the index set parameter.

The second subset deals with the consistent case. An arbitrary index *j* is chosen from the index set assigned to the expression *S* by the considered model $\mathcal{M}$, and the parametrised process $P(i)$ is interpreted with parameter *i* substituted by index *j*, which is modelled by updating the model $\mathcal{M}$ with the pair $i \mapsto j$.

$$
\begin{aligned}
& timedfailures_{ETCSP} [\![\sqcap_{i \in S} P(i)]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \\
& \quad \{tf : TimedFailure \mid ([\![S]\!]^{\epsilon}\, \mathcal{M}) = \varnothing\} \\
& \quad \cup \\
& \quad \{tf : TimedFailure \mid \exists j : ([\![S]\!]^{\epsilon}\, \mathcal{M}) \bullet \\
& \qquad\qquad tf \in timedfailures_{ETCSP} [\![P(i)]\!]_{(Env,(\mathcal{M} \oplus \{i \mapsto j\},\mathcal{M}_{\mathcal{I}}),IL)}\}
\end{aligned}
$$

Indexed Interleaving: Analogously to the indexed nondeterministic choice operator, the indexed interleaving operator $|||_{id \in S}$ has an index set parameter, which is interpreted in

the context of the Z part. According to the laws of timed CSP, this set must be finite. Then, the commutativity and associativity of the binary interleaving operator justify the following definition.

$$
\begin{array}{l}
\mathit{timedfailures}_{ETCSP}\, [\![\ |||_{i \in S}\, P(i)\ ]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \\
\quad \mathbf{let}\, IS == (\ [\![\ S\ ]\!]^{\epsilon}\ \mathcal{M}) \bullet \\
\qquad \{tf : \mathit{TimedFailure} \mid IS \notin \mathbb{F}\,\mathbb{W}\} \\
\qquad \cup \\
\qquad \{tf : \mathit{TimedFailure} \mid IS \in \mathbb{F}\,\mathbb{W} \wedge tf \in (\mathit{rec_interleave}\, [\![\ P(i)\ ]\!]_{(Env,\mathcal{M},IL)})IS\}
\end{array}
$$

The first subset addresses the inconsistent case, in which the index set expression is evaluated to an infinite set, while the second subset deals with the consistent case by referring to the function *rec_interleave*. This function is defined recursively with respect to the index set *IS*. The function *rec_interleave*, applied to the set of index values *IS*, yields the inductively computed interleaving of the processes $P(i)$ for $i \in IS$, where *IS* must be finite. Note the difference between the parameter *S* of the above and *IS* of the following definition: *S* is a Z expression whereas *IS* is the corresponding evaluated set of index values; while we cannot define a recursive function based on Z expressions, there is an induction principle for finite sets.

$$
\begin{array}{l}
\mathit{rec_interleave}\, [\![\ P(i)\ ]\!]_{(Env,\mathcal{M},IL)} = \lambda\, IS : \mathbb{P}\,\mathbb{W} \bullet \\
\quad \{tf : \mathit{TimedFailure} \mid IS = \varnothing \wedge tf \in \mathit{timedfailures}_{ETCSP}\, [\![\ \mathit{Skip}\ ]\!]_{(Env,(\mathcal{M},\mathcal{M}),IL)}\} \\
\quad \cup \\
\quad \{s, s1, s2 : \mathit{TimedTrace};\ \aleph 1, \aleph 2 : \mathit{TimedRefusal} \mid \\
\qquad IS \neq \varnothing \wedge (\exists\, j : IS \bullet \\
\qquad\quad (s1, \aleph 1) \in \mathit{timedfailures}_{ETCSP}\, [\![\ P(i)\ ]\!]_{(Env,(\mathcal{M} \oplus \{i \mapsto j\},\mathcal{M}),IL)} \wedge \\
\qquad\quad (s2, \aleph 2) \in (\mathit{rec_interleave}\, [\![\ P(i)\ ]\!]_{(Env,\mathcal{M},IL)})(IS \setminus \{j\})) \wedge \\
\qquad (s1, s2)\ \text{interleaves}\ s\ \wedge \\
\qquad \aleph 1 \setminus (\mathbb{T} \times \{\checkmark\}) = \aleph 2 \setminus (\mathbb{T} \times \{\checkmark\}) \\
\quad \bullet\, (s, \aleph 1 \cup \aleph 2)\}
\end{array}
$$

The first subset deals with the induction base—the empty set—where the indexed interleaving process represents just the terminating process *Skip*.

Accordingly, the second subset deals with the induction step. An index *j* is arbitrarily chosen from the non-empty set *IS*, and the process characterised by this index is composed with the indexed interleaving process that is obtained by deleting the chosen index from *IS*. The meaning of the latter process is determined by a recursive application of the function *rec_interleave*. The timed failures $(s1, \aleph 1)$ and $(s2, \aleph 2)$, being denotations of the derived processes, are combined to the timed failure $(s, \aleph 1 \cup \aleph 2)$ of the whole process, which reflects the definition of the binary interleaving operator in timed CSP.

Indexed Parallel: The indexed (synchronised) parallel operator is defined analogously to the indexed interleaving operator.

$$
\begin{array}{l}
\mathit{timedfailures}_{ETCSP}\, [\![\ \|_{i\in S}[A]P(i)\ ]\!]_{(Env,(\mathcal{M},\mathcal{M}_{\mathcal{I}}),IL)} = \\
\quad \mathbf{let}\ IS == (\ [\![\ S\ ]\!]^{\epsilon}\ \mathcal{M}) \bullet \\
\qquad \{tf : \mathit{TimedFailure} \mid IS \notin \mathbb{F}\,\mathbb{W}\} \\
\qquad \cup \\
\qquad \{tf : \mathit{TimedFailure} \mid IS \in \mathbb{F}\,\mathbb{W} \land tf \in (\mathit{rec_parallel}\, [\![\ (P(i), A)\ ]\!]_{(Env,\mathcal{M},IL)})IS\}
\end{array}
$$

The only difference is the way in which the denotations of the derived processes are combined in the second subset of the recursive function *rec_parallel*, which reflects the induction step.

$$
\begin{array}{l}
\mathit{rec_parallel}\, [\![\ (P(i), A)\ ]\!]_{(Env,\mathcal{M},IL)} = \lambda\, IS : \mathbb{P}\,\mathbb{W} \bullet \\
\quad \{tf : \mathit{TimedFailure} \mid IS = \varnothing \land tf \in \mathit{timedfailures}_{ETCSP}\, [\![\ \mathit{Skip}\ ]\!]_{(Env,(\mathcal{M},\mathcal{M}),IL)}\} \\
\quad \cup \\
\quad \{s, s1, s2 : \mathit{TimedTrace};\ \aleph 1, \aleph 2 : \mathit{TimedRefusal} \mid \\
\qquad IS \neq \varnothing \land (\exists\, j : IS \bullet \\
\qquad\quad (s1, \aleph 1) \in \mathit{timedfailures}_{ETCSP}\, [\![\ P(i)\ ]\!]_{(Env,(\mathcal{M}\oplus\{i\mapsto j\},\mathcal{M}),IL)} \land \\
\qquad\quad (s2, \aleph 2) \in (\mathit{rec_parallel}\, [\![\ (P(i), A)\ ]\!]_{(Env,\mathcal{M},IL)})(IS \setminus \{j\})) \land \\
\qquad (\exists_1\, \mathit{EvSet} ==\ [\![\ A\ ]\!]^{\epsilon}_{\alpha}\ (\mathcal{M}, IL) \bullet \\
\qquad\quad ((s1, s2), \mathit{EvSet})\ \mathrm{synch}\ s \land \\
\qquad\quad \aleph 1 \setminus (\mathbb{T} \times (\mathit{EvSet} \cup \{\checkmark\})) = \aleph 2 \setminus (\mathbb{T} \times (\mathit{EvSet} \cup \{\checkmark\}))) \\
\quad \bullet (s, \aleph 1 \cup \aleph 2)\}
\end{array}
$$

Note also the evaluation of the alphabet expression A by means of the semantic function $[\![\ _\ ]\!]^{\epsilon}_{\alpha}$, which is defined in the next section.

Predicate Language

Let us now address the other fragment of the extended timed CSP notation: the predicate language of the extended timed failures model.

Predicates of the extended timed failures model are used in concrete and abstract specification units to express environmental assumptions and/or to define requirements on the dynamic behaviour. Each predicate is interpreted in the context of a model $\mathcal{M}$, reflecting the definitions of the Z part, and a channel signature IL, reflecting the channel declarations of the `INTERFACE` and `LOCAL` sections; and it induces a relation between timed traces s and timed refusals $\aleph$.

The semantic function for predicates of the extended timed failures model has thus the following signature:

$$\mathit{timedfailures}_{ETFM}\, [\![\ _\ ]\!]_{_}\ : \mathit{ETFM_PREDICATE} \times (\mathit{Model} \times \mathit{ChanSgn}) \rightarrow \mathbb{P}\, \mathit{TimedFailure}$$

We define the semantic function $\mathit{timedfailures}_{ETFM}$ in terms of the function $[\![\ _\ ]\!]^{\mathcal{P}}_{ETFM}$, which is defined in the following. Both functions evaluate predicates of the extended timed failures model. The difference between them is that the latter function does not take into account the information contained in the `INTERFACE` and `LOCAL` sections. That is, it ignores that

the data values communicated with the occurrence of events on particular channels must be members of the value domains associated with these channels. To address this further requirement is the purpose of the semantic function $timedfailures_{ETFM}$:

$$\begin{array}{l} timedfailures_{ETFM} [\![Pred]\!]_{(\mathcal{M},IL)} = \{ s : TimedTrace;\ \aleph : TimedRefusal \mid \\ \quad ((s,\aleph),\mathcal{M},IL) \in [\![Pred]\!]^{\mathcal{P}}_{ETFM} \ \wedge \\ \quad \forall i : \mathrm{dom}\, s \bullet \\ \qquad chan_of(ev_of(s\,i)) \in \mathrm{dom}\, IL \ \wedge \\ \qquad val_of(ev_of(s\,i)) \in [\![IL(chan_of(ev_of(s\,i)))]\!]^{\epsilon}\, \mathcal{M} \} \end{array}$$

It has proved to be convenient to decouple the two functions for reasons of separation of concerns.

Syntax. We define the semantics of the "core" of the predicate language, containing all those predicates in terms of which all other predicates can be expressed equivalently. Thus, the process of computing the meaning of a predicate is divided into two phases. In the first phase, the predicate is transformed into an equivalent "core" predicate (by using transformation rules, which we do not completely treat in this thesis). In the second phase, the "core" predicate is associated with a meaning according to the semantic function to be defined in the following. This approach follows the one taken in the Z Standard [ISO, 2002], which also defines transformation rules between equivalent Z terms and semantic functions only for the "core" language.

The syntax of the predicate language is formally defined in Appendix D. The predicate language provides the usual logical connectives ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$). Its atomic predicates include all those provided by the predicate language of the (original) timed failures model [Schneider, 1999b], which allow one to make statements about the timed trace and timed refusal components s and $\aleph$. They make use of the various functions on timed traces and timed refusals as defined by Schneider [1999b] and described in Appendix E.

Our extensions to the predicate language of the timed failures model concern two aspects. First, we introduce references to Z schemas, say *Prop*,

$$Prop(v1 \mathrel{\widehat=} expr1, \ldots, vN \mathrel{\widehat=} exprN)$$

as additional atomic predicates. They allow us to specify relationships between data values communicated along particular channels with particular events. Second, we add to our predicate language universal and existential quantifications

$$\forall v : S \bullet Pred \qquad\qquad \exists v : S \bullet Pred$$

over arbitrary set expressions S, evaluated in the context of the Z part. Both extensions establish relationships between the model component $\mathcal{M}$ and the timed failure component $(s,\aleph)$.

Semantics. The semantic function $[\![-]\!]^{\mathcal{P}}_{ETFM}$ associates each predicate of the extended timed failures model with its characterising set of timed failures $(s,\aleph)$, models $\mathcal{M}$ and channel signatures IL.

$$[\![_]\!]^{\mathcal{P}}_{ETFM} : ETFM_PREDICATE \rightarrow \mathbb{P}(TimedFailure \times Model \times ChanSgn)$$

Logical Connectives.

$$[\![Pred1 \land Pred2]\!]^{\mathcal{P}}_{ETFM} = [\![Pred1]\!]^{\mathcal{P}}_{ETFM} \cap [\![Pred2]\!]^{\mathcal{P}}_{ETFM}$$
$$[\![Pred1 \lor Pred2]\!]^{\mathcal{P}}_{ETFM} = [\![Pred1]\!]^{\mathcal{P}}_{ETFM} \cup [\![Pred2]\!]^{\mathcal{P}}_{ETFM}$$
$$[\![\neg Pred]\!]^{\mathcal{P}}_{ETFM} = (TimedFailure \times Model \times ChanSgn) \setminus [\![Pred]\!]^{\mathcal{P}}_{ETFM}$$
$$[\![Pred1 \Rightarrow Pred2]\!]^{\mathcal{P}}_{ETFM} = [\![\neg Pred1 \lor Pred2]\!]^{\mathcal{P}}_{ETFM}$$
$$[\![Pred1 \Leftrightarrow Pred2]\!]^{\mathcal{P}}_{ETFM} = [\![Pred1 \land Pred2 \lor \neg Pred1 \land \neg Pred2]\!]^{\mathcal{P}}_{ETFM}$$

Universal Quantification. The predicate language of the ETFM allows us to introduce bound variables *var* universally quantified over arbitrary Z set expressions *S*. The set of values denoted by the Z expression *S* is computed with respect to a particular model $\mathcal{M}$ of the meaning of the Z part by using the semantic function for Z expressions $[\![_]\!]^{\epsilon}$ defined by the Z Standard.

$$[\![\forall var : S \bullet Pred]\!]^{\mathcal{P}}_{ETFM} = \bigcup\{\mathcal{M} : Model \bullet \bigcap\{val : [\![S]\!]^{\epsilon}\mathcal{M} \bullet \{tf : TimedFailure;\ IL : ChanSgn \mid (tf, \mathcal{M} \oplus \{var \mapsto val\}, IL) \in [\![Pred]\!]^{\mathcal{P}}_{ETFM} \bullet (tf, \mathcal{M}, IL)\}\}\}$$

Only those timed failures *tf*, models $\mathcal{M}$ and channel signatures *IL* are part of the denotation of the universal quantification that can be successfully extended by *all* values *val* of the denotation of the set expression *S*; this is reflected by the distributed intersection operator $\bigcap$.

Existential Quantification. Existential quantification is defined by exploiting its duality to universal quantification.

$$[\![\exists var : S \bullet Pred]\!]^{\mathcal{P}}_{ETFM} = [\![\neg \forall var : S \bullet \neg Pred]\!]^{\mathcal{P}}_{ETFM}$$

Local Definitions. Local definitions associate an expression *expr* with an identifier *var* within a local scope. Each local definition extends the model $\mathcal{M}$ used to interpret the subsequent predicate *Pred*.

$$[\![\mathbf{let}\ var == expr \bullet Pred]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid (tf, \mathcal{M} \oplus \{var \mapsto [\![expr]\!]^{\epsilon}_{Zval}(tf, \mathcal{M}, IL)\}, IL) \in [\![Pred]\!]^{\mathcal{P}}_{ETFM}\}$$

The expression *expr* can be an arbitrary expression evaluating to an untimed trace or an untimed refusal. The function $[\![_]\!]^{\epsilon}_{Zval}$ transforms untimed traces and refusals into corresponding Z values, see the next section.[6]

[6] There are also semantic functions evaluating timed/untimed trace, timed/untimed refusal, time value, time interval and natural number expressions. They are defined after we have dealt with the atomic predicates.

Atomic Predicates.

Schema reference. Introducing references to schemas defined in the Z part, say *Prop*, and associating their components, say $v1, \ldots, vN$, with untimed trace or refusal expressions, say $expr1, \ldots, exprN$, is the major link between our predicate language and the Z language.

$$\begin{array}{l}[\![Prop(v1 \mathrel{\widehat{=}} expr1, \ldots, vN \mathrel{\widehat{=}} exprN)]\!]^{\mathcal{P}}_{ETFM} = \\ \quad \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \quad\quad \{v1 \mapsto [\![expr1]\!]^{\epsilon}_{Zval}(tf, \mathcal{M}, IL), \ldots, vN \mapsto [\![exprN]\!]^{\epsilon}_{Zval}(tf, \mathcal{M}, IL)\} \in [\![Prop]\!]^{\epsilon}\, \mathcal{M}\}\end{array}$$

The expressions $expr1, \ldots, exprN$ are evaluated and transformed into Z values by the function $[\![_]\!]^{\epsilon}_{Zval}$. The resulting Z binding must be a member of the denotation of the Z schema, computed with the help of the semantic function $[\![_]\!]^{\epsilon}$ of the Z Standard.

Timed trace inclusion/prefix. One purpose of the predicate language is to restrict the timed trace component s. There are several predicates to achieve this, however, we deal only with the basic predicates, namely inclusion and prefix. The other predicates on timed traces can be derived.

$$\begin{array}{l}[\![s1 \ \mathbf{in}\ s2]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \qquad\qquad [\![s1]\!]^{\epsilon}_{TTr}(tf, \mathcal{M}, IL) \ \mathbf{in}_{ETFM}\ [\![s2]\!]^{\epsilon}_{TTr}(tf, \mathcal{M}, IL)\} \\ [\![s1 \ \mathbf{prefix}\ s2]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \qquad\qquad [\![s1]\!]^{\epsilon}_{TTr}(tf, \mathcal{M}, IL) \subseteq [\![s2]\!]^{\epsilon}_{TTr}(tf, \mathcal{M}, IL)\}\end{array}$$

Note that the base predicate ($s1$ **in** $s2$) is defined in terms of a corresponding relation $\mathbf{in}_{ETFM}$. The formal definition of this relation is part of the definition of the extended timed failures model. For the sake of brevity, we have omitted its presentation. This correspondence between base predicates of the predicate language and relations of the semantic model also applies to subsequent definitions.

Trace inclusion/prefix.

$$\begin{array}{l}[\![tr1 \ \mathbf{in}\ tr2]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \qquad\qquad [\![tr1]\!]^{\epsilon}_{Tr}(tf, \mathcal{M}, IL) \ \mathbf{in}_{ETFM}\ [\![tr2]\!]^{\epsilon}_{Tr}(tf, \mathcal{M}, IL)\} \\ [\![tr1 \ \mathbf{prefix}\ tr2]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \qquad\qquad [\![tr1]\!]^{\epsilon}_{Tr}(tf, \mathcal{M}, IL) \subseteq [\![tr2]\!]^{\epsilon}_{Tr}(tf, \mathcal{M}, IL)\}\end{array}$$

Timed refusal subset. Another purpose of the predicate language is to restrict the timed refusal component $\aleph$. Again, there are several predicates to achieve this; however, we deal only with the basic predicate used to restrict the timed refusal component, namely the subset relationship. The other predicates on timed refusals can be derived.

- Base

$$\begin{array}{l}[\![\aleph1 \subseteq \aleph2]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid \\ \qquad\qquad [\![\aleph1]\!]^{\epsilon}_{TRef}(tf, \mathcal{M}, IL) \subseteq [\![\aleph2]\!]^{\epsilon}_{TRef}(tf, \mathcal{M}, IL)\}\end{array}$$

- Derived

$\aleph 1 \subset \aleph 2 \equiv \aleph 1 \subseteq \aleph 2 \wedge \neg\, \aleph 1 = \aleph 2$
$te \in \aleph 1 \equiv \{te\} \subseteq \aleph 1$

Refusal subset.

- Base

$$[\![\, ref1 \subseteq ref2 \,]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid [\![\, ref1 \,]\!]^{\epsilon}_{Ref}\,(tf, \mathcal{M}, IL) \subseteq [\![\, ref2 \,]\!]^{\epsilon}_{Ref}\,(tf, \mathcal{M}, IL)\}$$

- Derived

$ref1 \subset ref2 \equiv ref1 \subseteq ref2 \wedge \neg\, ref1 = ref2$
$e \in ref \equiv \{e\} \subseteq ref$

Time comparison. The predicate language must also deal with the comparison of time values, related to the occurrence or refusal of particular events. The base predicate is $\leqslant_{\mathbb{R}}$, from which the other predicates can be derived.

Furthermore, we do not consider time intervals as basic entities of the predicate language. Accordingly, the membership of a time value to a time interval is transformed into an equivalent pair of comparisons with the limits of the interval.

- Base

$$[\![\, t1 \leqslant_{\mathbb{R}} t2 \,]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid ([\![\, t1 \,]\!]^{\epsilon}_{\mathbb{T}}\,(tf, \mathcal{M}, IL),\ [\![\, t2 \,]\!]^{\epsilon}_{\mathbb{T}}\,(tf, \mathcal{M}, IL)) \in (\mathcal{M}\ \leqslant_{\mathbb{R}})\}$$

- Derived

$t1 <_{\mathbb{R}} t2 \equiv \neg\, t2 \leqslant_{\mathbb{R}} t1$
$t \in [t1, t2] \equiv t1 \leqslant_{\mathbb{R}} t \wedge t \leqslant_{\mathbb{R}} t2$
$t \in (t1, t2) \equiv t1 <_{\mathbb{R}} t \wedge t <_{\mathbb{R}} t2$
$t \in [t1, t2) \equiv t1 \leqslant_{\mathbb{R}} t \wedge t <_{\mathbb{R}} t2$
$t \in (t1, t2] \equiv t1 <_{\mathbb{R}} t \wedge t \leqslant_{\mathbb{R}} t2$

Natural number comparison. Finally, the predicate language must deal with the comparison of natural numbers, related to the length of timed or untimed traces. The base predicate is $\leq$, from which the other predicates can be derived.

- Base

$$[\![\, n1 \leq n2 \,]\!]^{\mathcal{P}}_{ETFM} = \{tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \mid ([\![\, n1 \,]\!]^{\epsilon}_{Nat}\,(tf, \mathcal{M}, IL),\ [\![\, n2 \,]\!]^{\epsilon}_{Nat}\,(tf, \mathcal{M}, IL)) \in (\mathcal{M}\ \leq)\}$$

- Derived

$n1 < n2 \equiv \neg\, n2 \leq n1$

Semantic Functions on Expressions. According to the previous discussion of the predicate language, there are different kinds of entities to be set into relationship. That is, the expressions occurring in the predicate language evaluate to values of different "types," namely Z values, timed and untimed traces, timed and untimed refusals, time values and natural numbers. For each type, we define a separate semantic function evaluating expressions of this type.

The semantic function

$$[\![_]\!]^{\epsilon}_{Zval} : TimedFailure \times Model \times ChanSgn \to \mathbb{W}$$

is the interface between Z expressions and timed CSP expressions in our predicate language. Its purpose is to transform entities of the timed failures model, such as traces and refusals, into corresponding Z entities. The expressions that are bound to schema components within schema reference expressions are evaluated with the help of this function.

Any expression *tr* evaluating to an untimed trace can be transformed into a Z sequence. This is achieved by omitting the channel identifier from each event in the trace with the help of the projection *val_of*.

$$[\![\mathrm{seq}\, tr]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet val_of \circ [\![tr]\!]^{\epsilon}_{Tr} (tf, \mathcal{M}, IL)$$

Similarly, any expression *ref* evaluating to an untimed refusal can be transformed into a Z set. Again, this is achieved by omitting the channel identifier from each event in the refusal with the help of the projection *val_of*.

$$[\![\mathrm{set}\, ref]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet val_of (\!| [\![ref]\!]^{\epsilon}_{Ref} (tf, \mathcal{M}, IL) |\!)$$

It is also possible to transform single events into Z values. There are only four possibilities to denote untimed events: by applying the functions first, head, last and foot yielding the first and last event of an untimed or timed trace. Since we do not intend to introduce a semantic function to evaluate *event* expressions, we must embed separate event expressions within set expressions, which in turn can be evaluated by means of the function $[\![\ ]\!]^{\epsilon}_{Ref}$. The unique members of the resulting singleton sets are determined by the function *unique_mem*.

$$[\![\mathrm{head}\, tr]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$val_of(unique_mem([\![\{\mathrm{head}\, tr\}]\!]^{\epsilon}_{Ref} (tf, \mathcal{M}, IL)))$$

$$[\![\mathrm{first}\, s]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$val_of(unique_mem([\![\{\mathrm{first}\, s\}]\!]^{\epsilon}_{Ref} (tf, \mathcal{M}, IL)))$$

$$[\![\mathrm{foot}\, tr]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$val_of(unique_mem([\![\{\mathrm{foot}\, tr\}]\!]^{\epsilon}_{Ref} (tf, \mathcal{M}, IL)))$$

$$[\![\mathrm{last}\, s]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$val_of(unique_mem([\![\{\mathrm{last}\, s\}]\!]^{\epsilon}_{Ref} (tf, \mathcal{M}, IL)))$$

Finally, also time expressions can be transformed into corresponding Z values.

$$[\![\mathrm{time}\, t]\!]^{\epsilon}_{Zval} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet [\![t]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL)$$

Time-valued expressions are first-order citizens in our predicate language. They are evaluated with the help of the Z part, because reals including their associated functions and relations are predefined in the Z part; thus, each model $\mathcal{M}$ carries the needed information to interpret these expressions.

The semantic function

$$[\![_]\!]^{\epsilon}_{TTr} : TimedFailure \times Model \times ChanSgn \rightarrow TimedTrace$$

evaluates timed trace expressions.

As mentioned, each predicate of the extended timed failures model induces a relation between the dedicated identifiers s, $\aleph$, $\mathcal{M}$ and IL. The expression s is used to refer to the timed trace component.

$$[\![s]\!]^{\epsilon}_{TTr} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet tf.1$$

There are different constructors for timed traces. First, one can explicitly provide a finite sequence of timed events. The timed events can be composed of arbitrary time and value expressions, which are evaluated by the corresponding semantic functions. The empty timed trace and the concatenation of timed traces are the other possibilities to construct timed traces.

$$[\![\langle (t_1, c_1.val_1), \ldots, (t_N, c_N.val_N) \rangle]\!]^{\epsilon}_{TTr} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$\lambda\, i : 1 \mathrel{..} N \bullet ([\![t_i]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL), (c_i, [\![val_i]\!]^{\epsilon}_{Zval} (tf, \mathcal{M}, IL)))$$
$$[\![\langle\rangle]\!]^{\epsilon}_{TTr} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet \varnothing$$
$$[\![s1 \frown s2]\!]^{\epsilon}_{TTr} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$([\![s1]\!]^{\epsilon}_{TTr} (tf, \mathcal{M}, IL)) \frown_{ETFM} ([\![s2]\!]^{\epsilon}_{TTr} (tf, \mathcal{M}, IL))$$

The meaning of the other functions of the timed failures model yielding timed traces, e.g., tail and init, is defined in terms of their counterparts in RT-Z's semantic model. Since their definition is straightforward, they are omitted.

The semantic function

$$[\![_]\!]^{\epsilon}_{TRef} : TimedFailure \times Model \times ChanSgn \rightarrow TimedRefusal$$

evaluates timed refusal expressions.

The timed refusal component is referred to by the expression $\aleph$.

$$[\![\aleph]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet tf.2$$

There are three constructors for timed refusals: the explicit definition of a finite set of timed events, the empty set and union.

$$[\![\{(t_1, c_1.val_1), \ldots, (t_N, c_N.val_N)\}]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$\{i : 1 \mathrel{..} N \bullet ([\![t_i]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL), (c_i,\ [\![val_i]\!]^{\epsilon}_{Zval} (tf, \mathcal{M}, IL)))\}$$
$$[\![\varnothing]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet \varnothing$$
$$[\![\aleph 1 \cup \aleph 2]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$[\![\aleph 1]\!]^{\epsilon}_{TRef} (tf, \mathcal{M}, IL) \cup [\![\aleph 2]\!]^{\epsilon}_{TRef} (tf, \mathcal{M}, IL)$$

We do not need to define a separate semantic function to evaluate timed event expressions. The functions yielding timed events are evaluated in the context of singleton set expressions with the help of the current semantic function.

$$[\![\{\mathrm{head}\, sexpr\}]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$\{\mathrm{head}_{ETFM}([\![sexpr]\!]^{\epsilon}_{TTr} (tf, \mathcal{M}, IL))\}$$
$$[\![\{\mathrm{foot}\, sexpr\}]\!]^{\epsilon}_{TRef} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChanSgn \bullet$$
$$\{\mathrm{foot}_{ETFM}([\![sexpr]\!]^{\epsilon}_{TTr} (tf, \mathcal{M}, IL))\}$$

The various functions yielding timed refusals are defined in terms of their counterparts of the semantic model of RT-Z. Again, their definition is omitted.

The semantic function

$$[\![_]\!]^{\epsilon}_{Ref} : TimedFailure \times Model \times ChanSgn \to Refusal$$

evaluates (untimed) refusal expressions. A particular kind of refusal expression—expressions denoting set of events—are alphabet expressions.

The alphabet of a channel *chan*, denoted $\{\!|\ chan\ |\!\}$, is the set of all value-carrying events that can occur on this channel. It can be determined only on the basis of the value domain associated with the channel—a piece of information fixed by the channel declarations of the `INTERFACE` and `LOCAL` sections.

Alphabet expressions occur in the predicate language of the extended timed failures model, but also as parameters of some timed CSP operators, e.g., of the parallel composition operator $P \,|[\, A \,]|\, Q$; so the current semantic function is also used to define the meaning of those operators of the process term language, see the previous section. As mentioned, each reference to an external or internal channel ($\{\!|\ chan\ |\!\}$) denotes its alphabet. The `INTERFACE` and `LOCAL` sections induce the channel signature *IL* that maps each channel identifier to a Z expression, which is evaluated in the context of a particular model $\mathcal{M}$ of the Z part.

$$[\![\{\!|\ chan\ |\!\}]\!]^{\epsilon}_{Ref} = \lambda\, tf : TimedFailure;\ \mathcal{M} : Model;\ IL : ChannelType \mid chan \in \mathrm{dom}\, IL \bullet$$
$$\{chan\} \times ([\![IL\, chan]\!]^{\epsilon}\ \mathcal{M})$$

Note that the expression $\{\!|\ chan\ |\!\}$ is defined only if *chan* is declared as an external or internal channel, i.e., $chan \in \mathrm{dom}\, IL$.

When the function $[\![_]\!]^{\epsilon}_{Ref}$ is used to evaluate alphabet expressions, the timed failure parameter *tf* is not needed, see the above definition. For notational convenience, we introduce the following abbreviation

$[\![expr]\!]^{\epsilon}_{\alpha} (\mathcal{M}, IL) = [\![expr]\!]^{\epsilon}_{Ref} ((\langle\rangle, \varnothing), \mathcal{M}, IL)$

where we have arbitrarily chosen the simplest timed failure $(\langle\rangle, \varnothing)$.

The semantic functions $[\![_]\!]^{\epsilon}_{Tr}$, $[\![_]\!]^{\epsilon}_{\mathbb{T}}$, $[\![_]\!]^{\epsilon}_{TInt}$ and $[\![_]\!]^{\epsilon}_{Nat}$ evaluate untimed trace, time, time interval and natural number expressions. Since their definition is straightforward and provides no further insight, we omit it.

To conclude, we have completely formalised the predicate language of the extended timed failures model, in particular its novel features that go beyond the predicate language of timed CSP, e.g., quantification over Z sets and references to Z schemas. This formalisation is in contrast to the use of predicates in timed CSP, where quantification over data sets is used but without any formal underpinning.

9.2.5 Timed Failures/States Model (TFSM)

We now discuss the transition from the extended timed failures model to the timed failures/states model (see Figure 9.4).

The timed failures/states model extends the (extended) timed failures model by taking into account the evolution of the data state of the artifact under consideration, resulting from the termination and execution of operations within the observation interval. A timed state is a finite partial function, associating a data state with particular time instants within the observation interval.

$TimedState == \mathbb{T} \twoheadrightarrow\!\!\!\!\!\!\!\shortmid\;\; Binding$

A timed state is finite, because it is the result of operation terminations and executions, recorded in finite timed traces.

In the timed failures/states model, each specification unit is associated with a set of pairs consisting of a timed failure and a timed state.

Predicate Language

Predicates of the timed failures/states model are used in abstract specification units[7] to define the overall behaviour of the artifact under consideration. Each predicate in the timed failures/states model is interpreted in the context of a model $\mathcal{M}$, reflecting the definitions of the Z part, and a channel signature IL, reflecting the channel declarations of the `INTERFACE` and `LOCAL` sections; and it induces a relation between timed failures $(s, \aleph)$ and timed states tst.

The predicate language is a superset of the predicate language of the extended timed failures model defined in the previous section. The extension consists of the collection of specification macros expressing so-called 'time-variant state properties,' which we introduced in

[7] Therefore, this section is an anticipation of the definition of the semantics of abstract specification units in Section 9.3.

Section 6.1; they are additional atomic predicates, to which all operators defined in the previous section can be applied. These specification macros allow us to specify properties of the data state evolution by relating the timed state component *tst* to the remaining components already present in the extended timed failures model.

The semantic function $[\![-]\!]^{\mathcal{P}}_{TFSM}$ associates each predicate of the timed failures/states model with its characterising set of timed failures $(s, \aleph)$, timed states *tst*, models $\mathcal{M}$ and channel signatures *IL*.

$$[\![-]\!]^{\mathcal{P}}_{TFSM} : \mathit{TFSM_PREDICATE} \rightarrow \mathbb{P}(\mathit{TimedFailure} \times \mathit{TimedState} \times \mathit{Model} \times \mathit{ChanSgn})$$

We need to define the meaning of only the additional atomic predicates, i.e., of the specification macros expressing time-variant state properties. Defining the meaning of the constructs adopted from the extended timed failures model within the current model is analogous to the definition in the ETF model; the only difference is the additional timed state component, which is, however, not constrained by the adopted constructs.

Additional Specification Macros. The specification macro (*Prop* at *t*) is satisfied by a timed state *tst* if, and only if, the last data state that is recorded in the timed state *tst* before time instant *t* (inclusive) has property *Prop*.

$$\begin{array}{l} [\![\mathit{Prop}\ \mathsf{at}\ t]\!]^{\mathcal{P}}_{TFSM} = \{ tf : \mathit{TimedFailure};\ tst : \mathit{TimedState};\ \mathcal{M} : \mathit{Model};\ IL : \mathit{ChanSgn} \mid \\ \quad \mathit{Prop} \in \mathsf{dom}\, \mathcal{M} \wedge \\ \quad \exists_1\, t1 == \sup\{ t2 : \mathsf{dom}\, tst \mid t2 \leqslant_{\mathbb{R}} [\![t]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL) \} \bullet \\ \qquad tst\, t1 \in [\![\mathit{Prop}]\!]^{\epsilon}\, \mathcal{M} \} \end{array}$$

Accordingly, the specification macro (*Prop* before *t*) is satisfied by a timed state *tst* if, and only if, the last data state that is recorded in the timed state *tst* before time instant *t* (exclusive) has property *Prop*.

$$\begin{array}{l} [\![\mathit{Prop}\ \mathsf{before}\ t]\!]^{\mathcal{P}}_{TFSM} = \{ tf : \mathit{TimedFailure};\ tst : \mathit{TimedState};\ \mathcal{M} : \mathit{Model};\ IL : \mathit{ChanSgn} \mid \\ \quad \mathit{Prop} \in \mathsf{dom}\, \mathcal{M} \wedge \\ \quad \exists_1\, t1 == \sup\{ t2 : \mathsf{dom}\, tst \mid t2 <_{\mathbb{R}} [\![t]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL) \} \bullet \\ \qquad tst\, t1 \in [\![\mathit{Prop}]\!]^{\epsilon}\, \mathcal{M} \} \end{array}$$

The specification macro (*Prop* during *TI*) is satisfied by a timed state *tst* if, and only if, all data states of *tst* during interval *TI* have property *Prop*.

$$\begin{array}{l} [\![\mathit{Prop}\ \mathsf{during}\ [t1, t2]]\!]^{\mathcal{P}}_{TFSM} = \{ tf : \mathit{TimedFailure};\ tst : \mathit{TimedState};\ \mathcal{M} : \mathit{Model};\ IL : \mathit{ChanSgn} \mid \\ \quad \mathit{Prop} \in \mathsf{dom}\, \mathcal{M} \wedge \\ \quad \exists_1\, TI == \{ t : \mathsf{dom}\, tst \mid [\![t1]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL) \leqslant_{\mathbb{R}} t \wedge t \leqslant_{\mathbb{R}} [\![t2]\!]^{\epsilon}_{\mathbb{T}} (tf, \mathcal{M}, IL) \} \bullet \\ \qquad \forall\, t : TI \bullet tst\, t \in [\![\mathit{Prop}]\!]^{\epsilon}\, \mathcal{M} \} \end{array}$$

The definition of the macro during for the other possibilities to denote time intervals—$[t1, t2)$, $(t1, t2]$ and $(t1, t2)$—is straightforward and hence omitted.

Finally, (*Prop* invariant) is satisfied by a timed state *tst* if, and only if, each data state of *tst* has property *Prop*, i.e., if *Prop* is an invariant property.

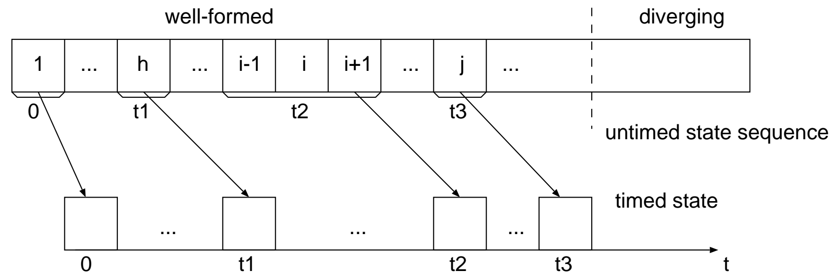


Figure 9.6: Mapping a state sequence to a timed state.

$$\llbracket\, Prop \ \ \mathsf{invariant} \,\rrbracket^{\mathcal{P}}_{TFSM} = \{ tf : TimedFailure;\ tst : TimedState;\ \mathcal{M} : Model;\ IL : ChanSgn \mid$$
$$Prop \in \mathsf{dom}\, \mathcal{M} \land$$
$$\forall\, t : \mathsf{dom}\, tst \bullet\ tst\, t \in\ \llbracket\, Prop \,\rrbracket^{\epsilon}\, \mathcal{M} \}$$

This concludes the formalisation of the predicate language of the timed failures/states model of RT-Z.

Timed Failures/States Semantics of Z Specifications

Consider Figure 9.6. It illustrates the transformation of an untimed state sequence, which is one component of a history, to a timed state. We have to distinguish between two parts of such an untimed state sequence: an initial part corresponding to a sequence of well-formed operation applications (in which operations applied concurrently do not interfere with each other), and the remaining, diverging part (possibly empty) corresponding to a sequence of interfering operation applications. In the example of Figure 9.6, we suppose that the state sequence is well-formed beyond state j. The data states with indices $i-1$, i and $i+1$ are all mapped to the time instant $t2$. That is, all three data states are caused by operations executed or terminated at $t2$. The data state recorded in the timed state at $t2$ is that with the maximal of those indices that are mapped to $t2$, namely $i+1$, because the other data states ($i-1$ and i) are only "intermediate" leading to the total result of applying all three operations to the previous data state.

This process of transforming an (untimed) state sequence into a timed state is formally defined by the relation *timedst_of_stseq*. Note that this relation does not fix a particular mapping of the data states of the untimed sequence to time values. This mapping is possible only in the context of a particular timed trace, recording the time instants of operation terminations and executions.

$$\begin{array}{l} timedst_of_stseq == \{h : History;\ tst : TimedState \mid \\ \quad \exists_1\, states == states_of\ h \bullet \\ \quad 0_{\mathbb{R}} \in \mathrm{dom}\, tst \land tst\, 0_{\mathbb{R}} = states\, 1 \land \\ \quad (\exists\, index_to_time : \mathrm{dom}\, states \setminus \{1\} \twoheadrightarrow \mathrm{dom}\, tst \setminus \{0_{\mathbb{R}}\} \mid \\ \qquad (\forall\, i, j : \mathrm{dom}\, states \mid i < j \bullet index_to_time\, i \leqslant_{\mathbb{R}} index_to_time\, j) \bullet \\ \qquad (\forall\, t : \mathrm{dom}\, tst \bullet tst\, t = states(\max\{i : \mathrm{dom}\, states \mid index_to_time\, i = t\}))) \\ \bullet (h, tst)\} \end{array}$$

The surjective, monotone function *index_to_time* maps the states of the untimed sequence to the time values at which they are caused because of the termination or execution of operations.

The following relation $timed\,states_Z$ specifies the association of histories of the Z part with timed states *tst* that are possible evolutions of the data state provided a particular timed failure *tf* has been observed.

$$\begin{array}{l} timed\,states_Z == \{h : History;\ tf : TimedFailure;\ tst : TimedState \mid \\ \quad \exists_1\, s == tf.1 \bullet \\ \qquad tst \in timedst_of_stseq(\!|\{h\}|\!) \land \\ \qquad \mathrm{dom}\, tst = \{0_{\mathbb{R}}\} \cup \{t : \mathbb{T} \mid \exists\, i : \mathrm{dom}\, s \bullet \\ \qquad\quad chan_of(ev_of(s\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s\, i) = t\} \\ \bullet (h, (tf, tst))\} \end{array}$$

The domain of the timed state is defined to be the set of time values at which operations have terminated or have been executed according to the timed trace *s*.

9.2.6 Semantic Integration: Definition

We have now available all preliminary definitions to deal with the last step of the semantic integration. The semantic function of the model of timed failures/states for concrete specification units, written $timed\,failures\,states_C\, [\![_]\!]$, maps each concrete specification unit to a set of pairs consisting of a timed failure and a timed state. To define this semantic function, we still need two auxiliary relations, $tfs_of_model_{INT}$ and $tfs_of_model_{EXT}$.

The relation $tfs_of_model_{INT}$ formally specifies our informal description in Figure 6.5 of composing the CSP part and the CSP interpretation of the Z part of a concrete specification unit. Let $(s_{CSP}, \aleph_{CSP})$ be a timed failure of the CSP part and $(s_Z, \aleph_Z)$ be a timed failure of the CSP interpretation of the Z part. The conditions defined in $tfs_of_model_{INT}$ to compose these two timed failures to obtain the timed failure $(s, \aleph)$ of the entire specification unit reflect the definition of the interface parallel operator in timed CSP [Schneider, 1999b, p. 353]. Both parts synchronise on the set of operation-related events *ORE*.

$$
\begin{array}{l}
tfs_of_model_{INT} == \{CSU : CSpecUnit;\ \mathcal{M} : Model;\ s : TimedTrace;\ \aleph : TimedRefusal;\\
\qquad\qquad\qquad\qquad\qquad\qquad tst : TimedState \mid\\
\quad \exists_1\, IL == CSU.I \cup CSU.L;\\
\qquad ORE == ORC \times ChannelValue;\ ORE^{\checkmark} == (ORC \times ChannelValue) \cup \{\checkmark\} \bullet\\
\qquad (\forall\, i : \mathrm{dom}\, s \bullet\\
\qquad\quad chan_of(ev_of(s\, i)) \in \mathrm{dom}\, IL \land\\
\qquad\quad val_of(ev_of(s\, i)) \in [\![IL(chan_of(ev_of(s\, i)))]\!]^{\epsilon}\, \mathcal{M}) \land\\
\qquad (\exists\, h : Histories(\!|\{(\mathcal{M}, CSU.OPS)\}|\!);\ s_Z, s_{CSP} : TimedTrace;\ \aleph_Z, \aleph_{CSP} : TimedRefusal \bullet\\
\qquad\quad (s_Z, \aleph_Z) \in timed\,failures_Z(\!|\{(h, Histories(\!|\{(\mathcal{M}, CSU.OPS)\}|\!))\}|\!) \land\\
\qquad\quad ((s_{CSP}, \aleph_{CSP}) \notin timed\,failures_{ETFM}\, [\![CSU.EA]\!]_{(\mathcal{M},IL)}\\
\qquad\quad \lor\ (s_{CSP}, \aleph_{CSP}) \in\\
\qquad\qquad timed\,failures_{ETCSP}\, [\![CSU.Behav\ Behaviour]\!]_{(CSU.Behav,(\mathcal{M},\mathcal{M}),IL)}) \land\\
\qquad\quad ((s_Z, s_{CSP}), ORE)\ \mathrm{synch}\ s \land\\
\qquad\quad \aleph_Z \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{CSP} \setminus (\mathbb{T} \times ORE^{\checkmark}) \land\\
\qquad\quad \aleph = \aleph_Z \cup \aleph_{CSP} \land\\
\qquad\quad ((s_Z, \aleph_Z), tst) \in timed\,states_Z(\!|\{h\}|\!))\\
\bullet\, ((CSU, \mathcal{M}), ((s, \aleph), tst))\}
\end{array}
$$

As depicted in Figure 6.5, the operation-related events are hidden from the environment of the parallel composition, which is formalised by the following relation.

$$
\begin{array}{l}
tfs_of_model_{EXT} == \{CSU : CSpecUnit;\ \mathcal{M} : Model;\ tfail : TimedFailure;\ tst : TimedState \mid\\
\quad \exists_1\, s == tfail.1;\ \aleph == tfail.2;\\
\qquad ORE == ORC \times ChannelValue \bullet\\
\qquad (\exists\, s^* : TimedTrace;\ \aleph^* : TimedRefusal \bullet\\
\qquad\quad ((s^*, \aleph^*), tst) \in tfs_of_model_{INT}(\!|\{(CSU, \mathcal{M})\}|\!) \land\\
\qquad\quad s = s^* \restriction_{ttr} ORE \land\\
\qquad\quad \aleph^* = \aleph \cup (\mathbb{T} \times ORE))\\
\bullet\, ((CSU, \mathcal{M}), (tfail, tst))\}
\end{array}
$$

The conditions defined in $tfs_of_model_{EXT}$ reflect the definition of the hiding operator in timed CSP, see [Schneider, 1999b, p. 354].

Finally, we build the union over all models of the meaning of the Z part that are consistent with the definition of the real numbers within ZF set theory. That is, we take only those models of the Z part into account that map the symbols of the Z section *Reals* to the corresponding entities of the semantic universe, whose existence is supposed on p. 122.

$$
\begin{array}{l}
timed\,failures\,states_C\, [\![_]\!] == \lambda\, CSU : CSpecUnit \bullet\\
\quad \bigcup\{\mathcal{M} : Model \mid \mathcal{M} \in [\![CSU.Zpart]\!]^{\mathcal{Z}}\ OpsPreds \land\\
\qquad\quad \{\mathbb{R}, 0_{\mathbb{R}}, 1_{\mathbb{R}}, +_{\mathbb{R}}, *_{\mathbb{R}}, <_{\mathbb{R}}\} \subseteq \mathrm{dom}\, \mathcal{M} \land\\
\qquad\quad \mathcal{M}\, \mathbb{R} = \mathcal{R} \land \mathcal{M}\, 0_{\mathbb{R}} = 0_{\mathcal{R}} \land \mathcal{M}\, 1_{\mathbb{R}} = 1_{\mathcal{R}} \land\\
\qquad\quad \mathcal{M} +_{\mathbb{R}} = +_{\mathcal{R}} \land \mathcal{M} *_{\mathbb{R}} = *_{\mathcal{R}} \land \mathcal{M}\ <_{\mathbb{R}} = <_{\mathcal{R}}\\
\qquad \bullet\, tfs_of_model_{EXT}(\!|\{(CSU, \mathcal{M})\}|\!)\}
\end{array}
$$

This concludes the description of the semantic model of timed failures/states for concrete specification units.

9.3 Abstract Specification Units (Open System View)

Having established the infrastructure of several semantic models in Section 9.2, the denotational semantics of abstract specification units can be defined on this basis quite succinctly.

Recall the structure of abstract specification units as described in Section 7.2. The semantics of such an abstract unit is the relation between timed failures and timed states as induced by the predicate that is associated with the dedicated identifier *Behaviour* in the `BEHAVIOURAL PROPERTIES` section.[8]

Let *TFSM_PREDICATE* denote the set of predicates of the timed failures/states model as specified by the syntax definition in Appendix D. These predicates constitute the notation of the `BEHAVIOURAL PROPERTIES` section of abstract specification units. The semantic function of the model of timed failures/states for abstract specification units, written $timedfailuresstates_A [\![_]\!]$, associates a set of pairs consisting of a timed failure and a timed state with each abstract specification unit being composed of

- a Z part, consisting of a `TYPES & CONSTANTS`, `I/O RELATIONS` and `STATE PROPERTIES` section,
- an `INTERFACE` section,
- an `ENVIRONMENTAL ASSUMPTIONS` section and
- a `BEHAVIOURAL PROPERTIES` section, which, by referring to the definitions of the other sections, fixes the required relation between timed failures and timed states.

$$\begin{aligned} ASpecUnit == [&Zpart : ZSpec; \\ &I : ChanSgn; \\ &EA : ETFM_PREDICATE; \\ &BehavProp : TFSM_PREDICATE] \end{aligned}$$

A pair $((s, \aleph), tst)$ is part of the denotation of an abstract specification unit *ASU* if

- the timed failure $(s, \aleph)$ violates the environmental assumption or
- there exists a model $\mathcal{M}$ such that $\mathcal{M}$, $(s, \aleph)$ and tst satisfy the predicate *Behaviour* of the `BEHAVIOURAL PROPERTIES` section.

[8] The `BEHAVIOURAL PROPERTIES` section can contain other named predicates, but these are used only to structure the predicate *Behaviour*, which refers—directly or indirectly—to the other predicates. Each set of predicates can thus be normalised to a single predicate *Behaviour*.

$$
\begin{array}{l}
\mathit{timed\,failures\,states}_A\,[\![_]\!] == \lambda\, ASU : ASpecUnit \bullet \\
\quad \bigcup\{\mathcal{M} : Model \mid \mathcal{M} \in [\![ASU.Zpart]\!]^{Z}\ Props \land \\
\qquad \{\mathbb{R}, 0_{\mathbb{R}}, 1_{\mathbb{R}}, +_{\mathbb{R}}, *_{\mathbb{R}}, <_{\mathbb{R}}\} \subseteq \mathrm{dom}\,\mathcal{M} \land \\
\qquad \mathcal{M}\,\mathbb{R} = \mathcal{R} \land \mathcal{M}\,0_{\mathbb{R}} = 0_{\mathcal{R}} \land \mathcal{M}\,1_{\mathbb{R}} = 1_{\mathcal{R}} \land \\
\qquad \mathcal{M}\,+_{\mathbb{R}} = +_{\mathcal{R}} \land \mathcal{M}\,*_{\mathbb{R}} = *_{\mathcal{R}} \land \mathcal{M}\ <_{\mathbb{R}} = <_{\mathcal{R}}\ \bullet \\
\qquad\quad \{s : TimedTrace;\ \aleph : TimedRefusal;\ tst : TimedState \mid \\
\qquad\qquad (\forall\, i : \mathrm{dom}\, s \bullet \\
\qquad\qquad\quad chan_of(ev_of(s\,i)) \in \mathrm{dom}\, ASU.I \land \\
\qquad\qquad\quad val_of(ev_of(s\,i)) \in [\![ASU.I(chan_of(ev_of(s\,i)))]\!]^{\epsilon}\,\mathcal{M}) \land \\
\qquad\qquad ((s, \aleph) \notin \mathit{timed\,failures}_{ETFM}\,[\![ASU.EA]\!]_{(\mathcal{M}, ASU.I)} \\
\qquad\qquad \lor \\
\qquad\qquad ((s, \aleph), tst, \mathcal{M}, ASU.I) \in [\![ASU.BehavProp]\!]^{\mathcal{P}}_{TFSM} \land \\
\qquad\qquad (State \in \mathrm{dom}\,\mathcal{M} \Rightarrow (\forall\, t : \mathrm{dom}\, tst\ \bullet\ tst\, t \in \mathcal{M}\, State))) \\
\qquad\quad \bullet ((s, \aleph), tst)\}\}
\end{array}
$$

This concludes the description of the semantic model of timed failures/states for abstract specification units. Altogether, we have completed the semantic model for the open system view and proceed with the closed system view in the following section.

9.4 Closed System View

The specification of an artifact in the closed system view incorporates only its behaviour at the external interface, i.e., its internal properties are hidden. Based on the definition of the semantic model of RT-Z for the open system view in the previous section, the semantic functions of RT-Z for the closed system view can be specified quite succinctly; we need only to hide the timed state component from the corresponding functions defined in the previous section.

9.4.1 Concrete Specification Units

The semantic function of the timed failures model for concrete specification units, written $\mathit{timed\,failures}_C\,[\![_]\!]$, maps each concrete specification unit to a set of timed failures. This function yields the domain of the relation that is computed by the corresponding semantic function for the open system view. This amounts to an existential quantification over the set of timed states.

$$
\mathit{timed\,failures}_C\,[\![_]\!] == \lambda\, CSU : CSpecUnit\ \bullet\ \mathrm{dom}(\mathit{timed\,failures\,states}_C\,[\![_]\!]\ CSU)
$$

9.4.2 Abstract Specification Units

The semantic function of the timed failures model for abstract specification units, written $\mathit{timed\,failures}_A\,[\![_]\!]$, maps each abstract specification unit to a set of timed failures. Also this function yields the domain of the relation that is computed by the corresponding semantic function for the open system view.

$$timedfailures_A\,[\![_]\!] == \lambda\, ASU : ASpecUnit \bullet \mathsf{dom}(timedfailures\,states_A\,[\![_]\!]\; ASU)$$

This concludes the description of the semantic model of RT-Z.

9.5 Definedness of Recursive Process Equations

Combining process algebras that are based on denotational semantics with state-based techniques generally raises the question if the fixed point theory underlying the denotational semantics of the chosen process algebra can be adopted in the combined language. In denotational approaches to defining the semantics of process algebras, the meaning of a recursive process equation $X = F(X)$ is defined to be the least fixed point of the process term F containing free occurrences of the process variable X. The assumption underlying this definition is the existence of a least fixed point for any valid process term F. To our knowledge, there are two approaches to ensuring the existence of fixed points of process equations.

The first approach is to impose a complete partial order (CPO) structure on the set of processes. The existence of least fixed points is guaranteed in this context, if all process operators are continuous, consult [Davey and Priestley, 1990, Chapter 4] for details.

The second approach is to impose a complete metric space (CMS) structure on the set of processes, based on a particular distance function between processes. The existence and uniqueness of fixed points is guaranteed in this context, if all process operators are contraction mappings. This is established by Banach's fixed point theorem, consult [Sutherland, 1975, p. 130] for details. The fixed point approach of timed CSP is based on a complete metric space structure, as discussed in Section A.2.

When combining a process algebra with a state-based technique, the advantage of the CMS approach over the CPO approach is that the property of a process operator to be a contraction mapping does not depend on the finiteness of process alphabets; this is in contrast to the property of a process operator to be continuous.[9] Consequently, the semantic model of timed CSP, based on the CMS approach, is appropriate for being combined with a state-based technique capable of introducing infinite data domains.

---

[9] Recursion in the failures–divergences model of CSP was defined by imposing a CPO structure on that model. For the failures–divergences model to be *complete*, however, the introduction of unbounded nondeterminism had to be forbidden. This restriction, in turn, ruled out some process operators, e.g., general nondeterministic choice and the hiding of infinite event sets. However, infinite event sets arise naturally in the context of state-based techniques, which are capable of introducing infinite data domains.

To cope with infinite process alphabets and certain forms of unbounded nondeterminism, Roscoe [1992] based the fixed point theory of CSP on an alternative order, which is different from the usual refinement order. Although this order is complete (CPO), the hiding operator fails to be continuous. Moreover, some forms of unbounded nondeterminism still cannot be treated in the context of this alternative order.

For these reasons, Roscoe [1993] invented the infinite traces model, which adds a further component to process denotations: the infinite traces that a process can exhibit. It covers all forms of unbounded nondeterminisms. By proving the congruence of the denotational semantics defined for the infinite traces model with the operational semantics of CSP, Roscoe and Barrett [1989] established the existence of fixed points for arbitrary recursive process equations.

The above discussion should have illustrated the difficulties that were encountered when dealing with unbounded nondeterminism in the context of the CPO underlying the denotational semantics of untimed CSP.

To ensure that recursive process equations in the CSP part of an RT-Z specification unit are well-defined, we must consider two issues that might counteract the fixed point approach of timed CSP:

- the additional "dimension" of nondeterminism introduced by the Z part and
- the dependency of the semantics of process terms in the extended timed failures model from information induced by the Z part.

Concerning the first issue, the behaviour of a specification unit, which is defined by its CSP part, is usually cyclic. The timed CSP process defining such a cyclic behaviour must be defined recursively. Further, the CSP part controls the evolution of the Z data state, which is usually infinite. Thus, the question arises if such recursively defined timed CSP process equations have a unique solution (i.e., if they have a unique fixed point) in spite of the additional "dimension" of unbounded nondeterminism constituted by the evolution of the Z data state.

Fortunately, the unbounded nondeterminism that is potentially introduced by Z operations working on an infinite data state on the one hand and the computation of the fixed point of the timed CSP process equations controlling the execution of these Z operations on the other hand can be completely separated. This ability to separate these two issues is due to the semantic separation of the Z part and the CSP part, which are composed in parallel. The definitions in the CSP part restrict only the order of operation executions and their timing. The corresponding effect on the data state is encapsulated in the parallel process that is the timed CSP interpretation of the Z part.

Note further that the timed failures semantics of the Z part is not defined by means of recursive process equations, but it is a direct mapping from Z specifications to sets of timed failures. The resulting sets of timed failures satisfy the properties of the timed failures model as demonstrated in Appendix B, i.e., they are guaranteed to represent timed CSP processes.

Concerning the second issue, in the extended timed failures model, each timed CSP process term is evaluated in the context of information induced by the corresponding Z part; it is encoded by the parameters of the semantic function $\mathit{timedfailures}_{ETCSP}$. The question arises if this context, in which process terms are evaluated, has any consequences on the existence and uniqueness of fixed points of recursive process equations.

These parameters of the semantic function solely concern the evaluation of value, time and index set expressions within process terms. Furthermore, this context information is identical for the evaluation of all instances of all process terms: whenever a process term is instantiated recursively, it is evaluated in the same context. That is, if a value, time or index set expression evaluates to a particular value, then this value is the result of all evaluations of this expression for all recursive instantiations of the process term. Thus, the context parameters do not influence the computation of fixed points of recursive process equations.

To conclude, the fixed point approach underlying timed CSP can be adopted as the fixed point approach of the integrated formalism RT-Z.

Chapter 10

Refinement

So far, we have introduced abstract and concrete specification units as a notation to define requirements and designs at different levels of abstraction; and we have defined a denotational semantics associating these specification units with a formal meaning.

According to Woodcock and Davies [1996], "writing a formal specification is a worthwhile activity in its own right." Among other things, this is justified by the facts that formal specification languages

- have the ability to precisely, unambiguously express requirements, constraints, architectures etc.,
- are the basis for detecting inconsistencies or incomplete definitions of requirements or design flaws and
- force the specifier to think more rigorously about the considered artifact, which helps gaining a deeper understanding of the artifact.

In addition to documenting the result of a single development step, however, one usually wishes to develop a formal specification towards an implementation in a stepwise manner. This process of development is called *refinement*. Refining a specification means improving it, usually in the sense that it becomes closer to implementation structures. Improving a specification may be achieved by removing nondeterminism. An abstract specification may leave some design decisions open, which are resolved in a refinement step by eliminating nondeterminism.

In the context of RT-Z, we must distinguish between different modes of refinement in several respects. Our starting point is to define refinement relations between single specification units; the refinement of compound specification units is defined on this basis. For both the refinement of single and compound specification units, we must take into account whether the refinement is to be conducted in the context of the *open* or *closed* system view, i.e., if the refinement should encompass the evolution of the (internal) data state. Finally, with regard to the refinement of single specification units, we must distinguish among the refinement between two concrete specification units and the refinement between an abstract and a concrete specification unit.

This chapter is organised according to these distinctions.

10.1 Refining Single Specification Units

In this section, we define the conditions under which a single specification unit is refined by another one in both the closed and the open system view; and we deal with techniques to establish refinement between single specification units.

10.1.1 Retrieve Specification Units

Usually, the refinement of a specification unit is accompanied by a refinement of the underlying data types—concerning the representation of the internal data states as well as the representation of the information communicated at the external interfaces. A retrieve specification unit is used to formally record the relationships between a refined specification unit, say SU_A, and a refining specification unit, say SU_C.

Figure 10.1 depicts the structure of a retrieve specification unit. It incorporates the retrieve relations between corresponding data domains in the refined unit SU_A and the refining unit SU_C, used there in order to define the domains of ports and channels, the data states and the parameters of operations. By convention, each retrieve specification unit imports the Z definitions of the specification units being related.

A retrieve specification unit is structured according to the following three tasks. First, it defines a retrieve relation for each external port and local channel that is subject to a change of representing the communicated information. Second, it records the relationship between the data states of the refining unit and the data states of the refined unit. Third, it defines a retrieve relation for each operation schema, recording a potential change of representing the input and output parameters. The refinement of operation parameters is a concept introduced by Derrick and Boiten [2001] using the term *IO refinement*; IO refinement extends the conventional refinement techniques of Z, which take into account changes of only the data state.

Note the potential redundancy between the retrieve relations in the `LOCAL` section and in the `OPS & PREDS` section. Some of the channels declared in the `LOCAL` section are related to operation applications; the retrieve relation for such a channel defines the change of representing the information communicated via this channel: the input and output parameters of the corresponding operation. Because of this potential redundancy, the retrieve relations in the `LOCAL` section that are associated with operation-related channels are not defined explicitly but are derived from the corresponding retrieve relations in the `OPS & PREDS` section.

Consider the structure of the Z schemas in the `INTERFACE` and the `LOCAL` section, in particular of the schema $RetrChan_{port1}$. These schemas define retrieve relations by declaring two components $\mathcal{A}$ and $\mathcal{C}$, which have a special meaning. $\mathcal{A}$ denotes any value exchanged at the considered port of the refined unit SU_A, while $\mathcal{C}$ denotes a corresponding value exchanged at the considered port of the refining unit SU_C. The predicate part of the schema defines the relationship between the abstract and concrete representation of the information exchanged at that port in terms of a predicate containing $\mathcal{A}$ and $\mathcal{C}$. This predicate, however, cannot be arbitrary. We require that the relation defined by the predicate is total in both directions: each abstract representation must be related to some concrete representation, and vice versa.

The schemas defining the retrieve relations for the data state and the operation parameters, on the other hand, need not introduce the special identifiers $\mathcal{A}$ and $\mathcal{C}$, because the entities

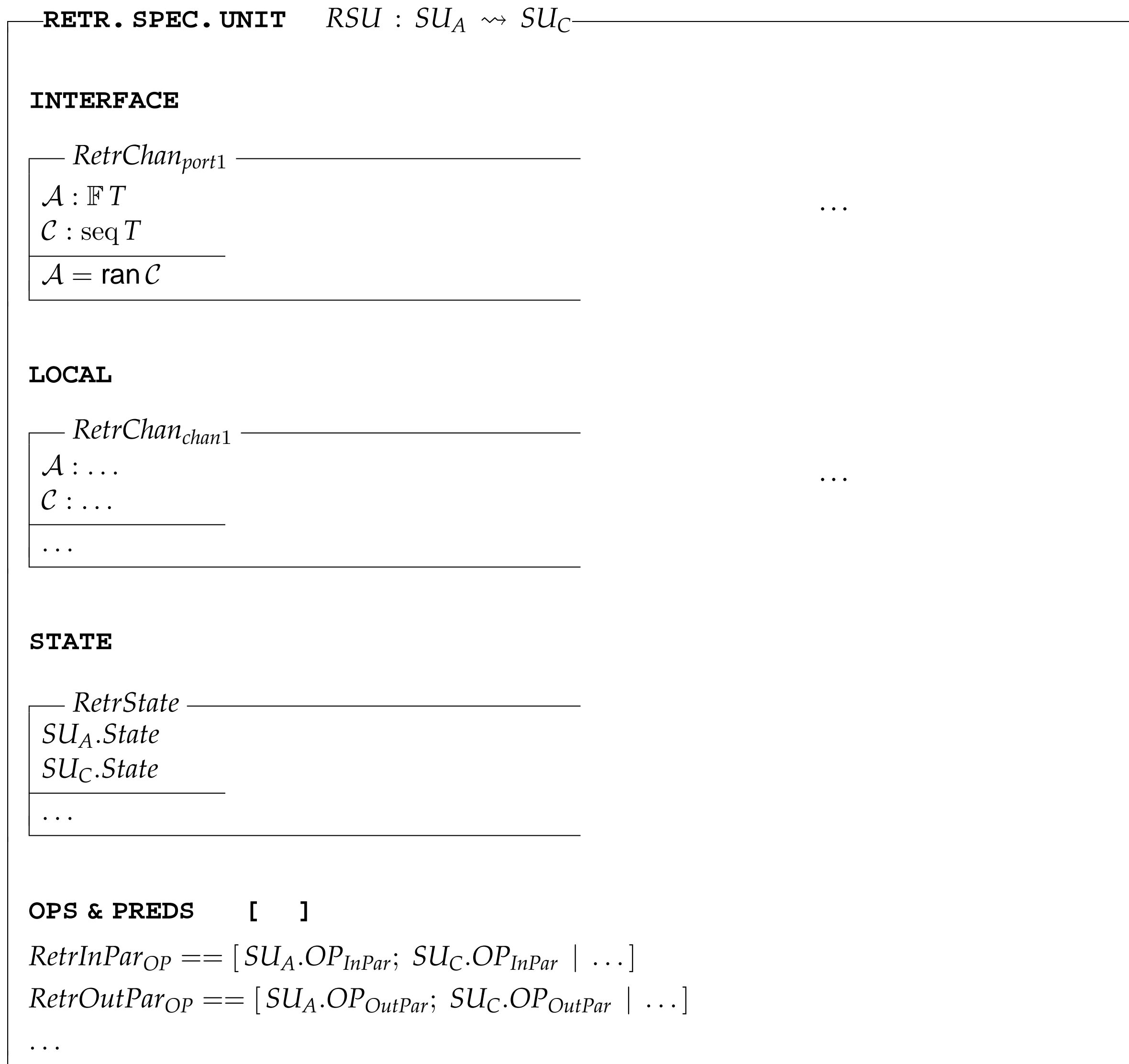


Figure 10.1: Template of retrieve specification units.

to be related are already associated with identifiers in the refined and the refining unit: the respective schemas are simply imported by the schema defining the retrieve relation. This schema import, however, requires that there are no common identifiers used by both the refined and the refining unit.[1]

[1] This can always be achieved by an appropriate renaming.

10.1.2 Relating Timed Observations of a Refined and a Refining Unit

The aim of this section is to lift the retrieve relations defined in a retrieve specification unit to the level of timed observations of a specification unit: retrieve relations between timed failures and between timed states. Accordingly, the subsequent definitions relate a refined and a refining specification unit, which are linked by a retrieve specification unit, with respect to several, increasingly complex concepts: histories, events, traces, failures, timed failures and timed states. These definitions are needed in order to specify the refinement relations in the next section.

Throughout the definition of the following relations, concepts are related that are associated with a refined (subscript a) and a refining specification unit (subscript c). The two specification units are linked by a retrieve specification unit whose Z part is associated with a set of models. $\mathcal{M}_\mathcal{R}$ is assumed to be a member of this set. Let $Retr_{schema}$ denote the function mapping each identifier occurring in the refined and the refining unit to the identifier of the corresponding schema in the retrieve specification unit, which defines the retrieve relation between the concrete instance in the refining unit and the abstract instance in the refined unit.

The set $Retr_{hist}$ defines a relation between two histories h_c and h_a associated with a refined and a refining specification unit.

$$
\begin{array}{l}
Retr_{hist} == \{h_a, h_c : History;\ \mathcal{M}_\mathcal{R} : Model \mid \\
\quad \mathsf{dom}(events_of\ h_c) = \mathsf{dom}(events_of\ h_a) \land \\
\quad (\forall\, i : \mathsf{dom}(events_of\ h_c) \bullet pop_name((events_of\ h_c)i) = pop_name((events_of\ h_a)i) \land \\
\qquad \{\mathcal{A} \mapsto (pop_params((events_of\ h_a)i)), \mathcal{C} \mapsto (pop_params((events_of\ h_c)i))\} \in \\
\qquad\qquad \mathcal{M}_\mathcal{R}(Retr_{schema}(pop_name((events_of\ h_a)i)))) \land \\
\quad \mathsf{dom}(states_of\ h_c) = \mathsf{dom}(states_of\ h_a) \land \\
\quad (\forall\, i : \mathsf{dom}(states_of\ h_c) \bullet ((states_of\ h_c)i \cup (states_of\ h_a)i) \in (\mathcal{M}_\mathcal{R}\ RetrState)) \\
\bullet (\mathcal{M}_\mathcal{R}, (h_a, h_c))\}
\end{array}
$$

The set $Retr_{ev}$ relates two events ev_a and ev_c associated with a refined and a refining specification unit. The events must be transmitted via the same channel, and the values being transmitted with their occurrence must be related by the retrieve schema defined in the retrieve specification unit and belonging to that channel.

$$
\begin{array}{l}
Retr_{ev} == \{ev_a, ev_c : Event;\ \mathcal{M}_\mathcal{R} : Model \mid \\
\quad chan_of\ ev_a = chan_of\ ev_c \land \\
\quad \{\mathcal{A} \mapsto val_of\ ev_a, \mathcal{C} \mapsto val_of\ ev_c\} \in \mathcal{M}_\mathcal{R}(Retr_{schema}(chan_of\ ev_a)) \\
\bullet (\mathcal{M}_\mathcal{R}, (ev_a, ev_c))\}
\end{array}
$$

The sets $Retr_{tr}$, $Retr_{fail}$ and $Retr_{tf}$ define the corresponding relations between traces, failures and timed failures associated with a refined and a refining specification unit. Their definition is based on $Retr_{ev}$.

$$Retr_{tr} == \{tr_a, tr_c : Trace;\ \mathcal{M}_{\mathcal{R}} : Model \mid \\ \quad \mathsf{dom}\, tr_a = \mathsf{dom}\, tr_c \wedge (\forall i : \mathsf{dom}\, tr_a \bullet (tr_a\, i, tr_c\, i) \in Retr_{ev}(\!| \{\mathcal{M}_{\mathcal{R}}\} |\!)) \\ \bullet (\mathcal{M}_{\mathcal{R}}, (tr_a, tr_c))\}$$

$$Retr_{fail} == \{tr_a, tr_c : Trace;\ X_a, X_c : Refusal;\ \mathcal{M}_{\mathcal{R}} : Model \mid \\ \quad (tr_a, tr_c) \in Retr_{tr}(\!| \{\mathcal{M}_{\mathcal{R}}\} |\!) \wedge X_c \subseteq Retr_{ev}(\!| \{\mathcal{M}_{\mathcal{R}}\} |\!)(\!| X_a |\!) \\ \bullet (\mathcal{M}_{\mathcal{R}}, ((tr_a, X_a), (tr_c, X_c)))\}$$

$$Retr_{tf} == \{s_a, s_c : TimedTrace;\ \aleph_a, \aleph_c : TimedRefusal;\ \mathcal{M}_{\mathcal{R}} : Model \mid \\ \quad \forall t : \mathbb{T} \bullet \\ \qquad ((\mathrm{strip}(s_a \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_a\}), (\mathrm{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\})) \\ \qquad \in Retr_{fail}(\!| \{\mathcal{M}_{\mathcal{R}}\} |\!) \\ \bullet (\mathcal{M}_{\mathcal{R}}, ((s_a, \aleph_a), (s_c, \aleph_c)))\}$$

The set $Retr_{tst}$ relates two timed states tst_a and tst_c associated with a refined and a refining specification unit. The timed states must map states to the same time instants, and for each such time instant the states being recorded must be related by the schema *RetrState* defined in the retrieve specification unit.

$$Retr_{tst} == \{tst_a, tst_c : TimedState;\ \mathcal{M}_{\mathcal{R}} : Model \mid \\ \quad \mathsf{dom}\, tst_c = \mathsf{dom}\, tst_a \wedge (\forall t : \mathsf{dom}\, tst_a \bullet tst_c\, t \cup tst_a\, t \in (\mathcal{M}_{\mathcal{R}}\, RetrState)) \\ \bullet (\mathcal{M}_{\mathcal{R}}, (tst_a, tst_c))\}$$

The set $Retr_{Interface}$ defines the retrieve relation between timed failures associated with a refined and a refining unit that is induced by a retrieve specification unit *RSU*.[2]

$$Retr_{Interface} == \{s_a, s_c : TimedTrace;\ \aleph_a, \aleph_c : TimedRefusal;\ RSU : RSpecUnit \mid \\ \quad \exists \mathcal{M}_{\mathcal{R}} : [\![RSU.Zpart]\!]^{\mathcal{Z}}\, Root \bullet ((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\!| \{\mathcal{M}_{\mathcal{R}}\} |\!) \\ \bullet (RSU, ((s_a, \aleph_a), (s_c, \aleph_c)))\}$$

Analogously, the set $Retr_{Interface/State}$ defines the retrieve relation between timed failures and timed states associated with a refined and a refining unit that is induced by a retrieve specification unit *RSU*.

[2] The identifier *Root* occurring in the definition denotes an "artificial" root section, which is part of each retrieve specification unit. This section inherits all other Z sections of a retrieve specification unit; each model associated with it contains hence all variables declared in the entire unit.

$$\begin{array}{l} Retr_{Interface/State} == \{s_a, s_c : TimedTrace;\ \aleph_a, \aleph_c : TimedRefusal;\ tst_a, tst_c : TimedState; \\ \qquad\qquad RSU : RSpecUnit \mid \\ \quad \exists\, \mathcal{M}_{\mathcal{R}} : [\![RSU.Zpart]\!]^{\mathcal{Z}}\ Root \bullet \\ \qquad ((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \wedge (tst_a, tst_c) \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \\ \bullet (RSU, (((s_a, \aleph_a), tst_a), ((s_c, \aleph_c), tst_c)))\} \end{array}$$

The two previous relations are used to define the refinement relation between specification units.

10.1.3 Closed System View

Considering a specification unit in the closed system view, if abstract or concrete, we are interested in its behaviour at the external interface, ignoring its internal behaviour. For a specification unit to refine another one, its associated set of timed failures must be a subset of the set of timed failures associated with the refined unit. However, the timed failures of the two units cannot be compared directly because of a potential change of representing the data values that are recorded in the timed failures. The comparison is thus performed with respect to the retrieve relations defined in the previous section.

Concrete to Concrete

Refining concrete specification units is all about converging models towards implementation structures—with respect to data structures, algorithms as well as architectures.

Definition 10.1
A concrete specification unit CSU_A is refined by a concrete specification unit CSU_C with respect to a retrieve specification unit RSU if, and only if, all external interactions consistent with CSU_C are also consistent with CSU_A—taking into account the potential change of data representation as recorded in the `INTERFACE` *section of RSU.*

$$\begin{array}{l} CSU_A \sqsubseteq_{TF}^{RSU} CSU_C \\ \Leftrightarrow \\ timedfailures_C\ [\![CSU_C]\!] \subseteq Retr_{Interface}(\!|\{RSU\}|\!)(\!|timedfailures_C\ [\![CSU_A]\!]\,|\!) \end{array}$$

Abstract to Concrete

The transition between abstract and concrete specification units is not really a refinement step. Rather, a concrete specification unit is constructed from scratch in order to define a model that meets the requirements defined by an abstract specification unit; demonstrating the satisfaction of these requirements is a verification task. Verification techniques for RT-Z are part of future work, see Chapter 13. We nevertheless define the implementation relation between abstract and concrete specification units in this section.

Definition 10.2
An abstract specification unit ASU is refined by a concrete specification unit CSU if, and only if, all external interactions consistent with CSU are also consistent with ASU—taking into

account the potential change of data representation as recorded in the `INTERFACE` *section of RSU.*

$$ASU \sqsubseteq_{TF}^{RSU} CSU$$
$$\Leftrightarrow$$
$$timed\,failures_C \llbracket CSU \rrbracket \subseteq Retr_{Interface}(\!| \{RSU\} |\!)(\!| timed\,failures_A \llbracket ASU \rrbracket |\!)$$

Note that we have not defined a refinement relation between pairs of abstract specification units. From our point of view, abstract specification units are not subject to refinement, because they fix mere requirements, which should be already known when the part of the development process starts that is supported by RT-Z.

10.1.4 Open System View

Considering a specification unit in the open system view, we are interested in both its interaction at the external interface and the evolution of its internal data state. To check if there is a refinement relationship between two specification units, we must compare their associated sets of timed failures and timed states. Again, the timed failures and timed states of the two units cannot be compared directly because of a potential change of data representation. The comparison is thus made with respect to the retrieve relations defined in Section 10.1.2.

The following definitions are analogous to that of the closed system view.

Concrete to Concrete

Definition 10.3
A concrete specification unit CSU_A is refined by a concrete specification unit CSU_C with respect to a retrieve specification unit RSU if, and only if, all external interactions and data state evolutions consistent with CSU_C are also consistent with CSU_A—taking into account the potential changes in data representation as recorded in the `INTERFACE` *and* `STATE` *sections of RSU.*

$$CSU_A \sqsubseteq_{TFTS}^{RSU} CSU_C$$
$$\Leftrightarrow$$
$$timed\,failures\,states_C \llbracket CSU_C \rrbracket \subseteq Retr_{Interface/State}(\!| \{RSU\} |\!)(\!| timed\,failures\,states_C \llbracket CSU_A \rrbracket |\!)$$

Abstract to Concrete

Definition 10.4
An abstract specification unit ASU is refined by a concrete specification unit CSU with respect to a retrieve specification unit RSU if, and only if, all external interactions and data state evolutions consistent with CSU are also consistent with ASU—taking into account the potential changes in data representation as recorded in the `INTERFACE` *and* `STATE` *sections of RSU.*

$$ASU \sqsubseteq_{TFTS}^{RSU} CSU$$
$$\Leftrightarrow$$
$$\mathit{timed\,failures\,states}_C \,[\![\, CSU \,]\!] \subseteq \mathit{Retr}_{\mathit{Interface/State}}(\!|\{RSU\}|\!)(\!|\mathit{timed\,failures\,states}_A \,[\![\, ASU \,]\!] \,|\!)$$

10.1.5 Techniques for Establishing Refinement

According to the discussion in the previous two sections, the task that remains to be dealt with is the refinement of concrete specification units. In this context, our aim is to make use of the existing refinement techniques of the base formalisms Z and timed CSP. Because of the chosen approach to defining the meaning of concrete specification units—the parallel composition of the Z and CSP part—their parts can be refined independently of each other. This independence is ensured by the monotonicity of the parallel composition operator with respect to refinement in the timed failures model.

Concerning the CSP part of concrete specification units this means that we can simply use the refinement techniques provided by timed CSP.

Concerning the Z part, on the other hand, we must establish a link between the refinement techniques provided by Z and refinement in the timed failures model. In other words, we must establish that using state-based techniques to refine the Z part results in a refinement of the timed CSP interpretation of the Z part according to our semantic definitions in Chapter 9. This approach follows the work of Josephs [1988] who has demonstrated that state-based refinement techniques are consistent with refinement in the failures–divergences model of (untimed) CSP. We cannot, however, simply adopt his results in the context of RT-Z, because

- our approach to defining the timed CSP semantics of a Z specification differs from his approach to defining the CSP semantics of a state-based system and
- two different semantic models are involved: the timed failures model and the failures–divergences model.

We proceed as follows. We first introduce the definition of the notions of forward and backward simulations. In Section 10.3 we prove that forward and backward simulations imply refinement in the timed failures/states model of RT-Z.

Josephs [1988, Rules 2.2 and 2.3 on p. 12] has defined the notions of downward and upward simulations in the context of state-transition systems.[3] We must translate his definitions into the context of Z as used within RT-Z. The most essential aspect of this translation is the identification of the set $next(\sigma)$, denoting the set of outgoing actions from the state σ, and the set of operations whose preconditions are satisfied in the data state corresponding to σ. This identification corresponds to the rôle we have assigned to operation preconditions in RT-Z: an operation is blocked in a data state in which its precondition is not satisfied. The most substantial difference between the following conditions characterising forward and backward simulations in RT-Z and the corresponding conditions given in [Josephs, 1988] is that we cannot (explicitly) quantify over the set of operation identifiers present in a specification unit, whereas Josephs has used universal and existential quantifiers over the set of actions

[3] Instead of the terms upward and downward simulation, we use the terms forward and backward simulation, because they are more intuitive and also used in [Woodcock and Davies, 1996].

of a state-transition system. Another difference between the definitions of Josephs (and also the definitions of forward and backward simulation in [Woodcock and Davies, 1996]) and our definitions is that we have taken into account the refinement of data types associated with the input and output parameters of operations. The conditions have been extended accordingly.

Forward Simulations. Let CSU_A and CSU_C be two concrete specification units. Let *RSU* be a retrieve specification unit defining the various retrieve relations. The schema *RSU.RetrState* is called a *forward simulation* between the data states of CSU_A and CSU_C if the following three conditions hold for each operation schema *OP*. All conditions are implicitly universally quantified over the set of operation identifiers defined in CSU_A and CSU_C. The conditions are a translation of the Conditions 1–3 of Rule 2.2 of [Josephs, 1988].

1. Applicability

$$\forall CSU_A.State;\ CSU_A.OP_{InPar};\ CSU_C.State;\ CSU_C.OP_{InPar} \mid RSU.RetrState \land RSU.InRetr_{OP} \bullet \mathsf{pre}\, CSU_A.OP \Leftrightarrow \mathsf{pre}\, CSU_C.OP$$

   Note that, in contrast to refinement in Z, the precondition of an operation must not be changed during refinement. This is a consequence of the different rôles of the precondition in Z and RT-Z (non-blocking vs. blocking).

2. Correctness

$$\forall CSU_A.State;\ CSU_A.OP_{InPar};\ CSU_C.State;\ CSU_C.OP_{InPar};\ CSU_C.State';\ CSU_C.OP_{OutPar} \mid \mathsf{pre}\, CSU_A.OP \land RSU.RetrState \land RSU.InRetr_{OP} \land CSU_C.OP \bullet \exists CSU_A.State';\ CSU_A.OP_{OutPar} \bullet CSU_A.OP \land RSU.RetrState' \land RSU.OutRetr_{OP}$$

3. Initialisation

$$\forall CSU_C.State \mid CSU_C.Init \bullet \exists CSU_A.State \mid CSU_A.Init \bullet RSU.RetrState$$

Backward Simulations. Let CSU_A and CSU_C be two concrete specification units. Let *RSU* be a retrieve specification unit defining the various retrieve relations. The schema *RSU.RetrState* is called a *backward simulation* between the data states of CSU_A and CSU_C if the following three conditions hold for each operation schema *OP*. Conditions 2 and 3 are direct translations of the Conditions 2 and 3 of Rule 2.3 of [Josephs, 1988]. Condition 1 is slightly strengthened, because it is not possible to directly translate the subset relationship without having the ability to quantify over the set of operation identifiers explicitly.

1. Applicability

$$\forall CSU_A.State;\ CSU_A.OP_{InPar};\ CSU_C.State;\ CSU_C.OP_{InPar} \mid RSU.RetrState \land RSU.InRetr_{OP} \bullet \mathsf{pre}\, CSU_A.OP \Leftrightarrow \mathsf{pre}\, CSU_C.OP$$

2. Correctness

$$\forall CSU_C.State;\ CSU_C.OP_{InPar};\ CSU_A.State';\ CSU_A.OP_{OutPar};\ CSU_C.State';\ CSU_C.OP_{OutPar} \mid CSU_C.OP \land RSU.RetrState' \land RSU.OutRetr_{OP} \bullet \exists CSU_A.State;\ CSU_A.OP_{InPar} \bullet RSU.RetrState \land RSU.InRetr_{OP} \land CSU_A.OP$$

3. Initialisation

$$\forall CSU_C.State;\ CSU_A.State \mid CSU_C.Init \land RSU.RetrState \bullet CSU_A.Init$$

This completes the definition of the notions of forward and backward simulations. Note that identifying and establishing simulations between specification units is a task that has been reduced to proof obligations that are expressed in terms of the Z notation and that can be completely discharged using Z techniques. Such techniques for establishing simulations between abstract data types (ADT) in Z can be found in [Derrick and Boiten, 2001].

In Section 10.3, we prove that the existence of a forward or backward simulation between two concrete specification units, as defined in this section, implies refinement in the semantic model of RT-Z.

Discussion

As already indicated, Derrick and Boiten [2001] have introduced the concept of IO refinement for Z abstract data types, which takes into account changes of representing operation parameters. Their definition of the notions of forward simulation and backward simulation and our definition of these notions in this section, however, differ in some aspects for the following reasons.

First, and most important, the preconditions of operation schemas have different rôles in Z and RT-Z, giving rise to different applicability and correctness conditions.

Moreover, it seemed most natural to us to treat input parameters analogously to pre states and output parameters analogously to post states. This analogy/symmetry is directly reflected in the above conditions. Derrick and Boiten, by contrast, have used the schema piping operator in order to relate the abstract and concrete instances of input and output parameters. This operator allowed them to express the conditions in a more compact way, however, it prevented a symmetric treatment.

As an analogy to our approach, Derrick and Boiten have not claimed that forward and backward simulations together are sufficient to prove all valid IO refinement relationships between abstract data types in Z.

10.2 Refining Compound Specification Units

In this section, we discuss how refinement relations between single specification units are lifted to compound specification units.

10.2.1 Aggregation

Our aim is to establish—under particular assumptions—that the refinement of an aggregated specification unit, while keeping the context unchanged into which it is embedded, results in the refinement of the aggregating specification unit constituting its context. The reasoning in the remainder of this section is carried out with respect to the open system view but holds for the closed system view as well.

Let CSU_A and CSU_C be two concrete specification units and let $Comp_A$ and $Comp_C$ be two compound specification units that aggregate CSU_A and CSU_C, respectively, within the same context.

```
SPEC.UNIT   Comp_A
...
SUBUNITS
subunit agg spec.unit CSU_A
...
```

```
SPEC.UNIT   Comp_C
...
SUBUNITS
subunit agg spec.unit CSU_C
...
```

We need two assumptions in order to justify the satisfaction of the following theorem.

Assumption 10.1
The behaviour definition of $Comp_A$ embeds CSU_A within its context by means of a timed CSP operator that is monotone with respect to refinement in the timed failures model, e.g., parallel composition or interleaving. By assumption, the same holds for $Comp_C$ and CSU_C.

Assumption 10.2
The compound unit $Comp_A$ does not define any global invariants and global initialisation constraints. By assumption, the same holds for $Comp_C$ and CSU_C.

Theorem 10.1
Let RSU be a retrieve specification unit defining retrieve relations for the ports and channels of CSU_A and CSU_C, and let RSU^* extend RSU corresponding to the way CSU_A and CSU_C are embedded within $Comp_A$ and $Comp_C$: there might be additional ports and channels in $Comp_A$ and $Comp_C$ whose associated domains must also be related (in this case identified).

Ideally, the result to be obtained would be:

$CSU_A \sqsubseteq^{RSU}_{TFTS} CSU_C \Rightarrow Comp_A \sqsubseteq^{RSU^*}_{TFTS} Comp_C$.

However, since it has not been established that forward and backward simulations together are sufficient to prove refinement, we must strengthen the premise of the implication by the following conjunct: there exists a forward or backward simulation between the data states of CSU_A and CSU_C.

So, informally speaking, if we are able to prove refinement between CSU_A and CSU_C using a simulation between their data states, we are also able to prove refinement between $Comp_A$ and $Comp_C$.

We do not prove the above theorem formally, but provide a systematic reasoning. This reasoning can be sketched as follows. Concerning the Z part, we construct a forward/backward simulation between the data states of $Comp_A$ and $Comp_C$ based on the existing forward/backward simulation between CSU_A and CSU_C. This ensures that the Z parts of the compound specification units are related by refinement. Concerning the CSP part, we exploit the assumed monotonicity of the used timed CSP operator to establish the necessary refinement relationship between the CSP parts of $Comp_A$ and $Comp_C$.

According to Assumption 10.2 and the reduction process defined in Section 8.1.4, the data state and initialisation schemas of $Comp_A$ and $Comp_C$ are derived as follows.

$Comp_A.State$	$Comp_C.State$
$agg : \texttt{lift_simple_Z}(agg, CSU_A.State)$	$agg : \texttt{lift_simple_Z}(agg, CSU_C.State)$
...	...

$Comp_A.Init$	$Comp_C.Init$
$Comp_A.State$	$Comp_C.State$
$agg \in \texttt{lift_simple_Z}(agg, CSU_A.Init)$	$agg \in \texttt{lift_simple_Z}(agg, CSU_C.Init)$
...	...

By assumption, there is a forward/backward simulation between the data states of CSU_A and CSU_C. This simulation can be easily lifted to a simulation between the data states of $Comp_A$ and $Comp_C$ such that the following two conditions hold.

- The components *agg* of $Comp_A.State$ and $Comp_C.State$ are related by the original simulation.
- The other components of $Comp_A.State$ and $Comp_C.State$, which are identical by assumption, are related by the identity relation.

Choosing this lifted simulation, the definition of the promotion of operations defined in Section 8.1.4 guarantees that each pair of corresponding operations from $Comp_A$ and $Comp_C$ satisfies Conditions 1 and 2 for forward/backward simulations.

- The execution of an operation defined by CSU_A and CSU_C depends on and transforms only the state component *agg*, because we have assumed that there are no global invariants.

- Each operation not defined by CSU_A and CSU_C is defined equally in $Comp_C$ and $Comp_A$ by assumption. Its execution leaves the state component *agg* unchanged.

Finally, Condition 3 for forward/backward simulations is satisfied, because there are no global initialisation constraints and because the state components other than *agg* are related by the identity relation.

The above reasoning implies the existence of a forward/backward simulation between the data states of $Comp_A$ and $Comp_C$, so the Z parts of these two units are related by refinement.

Concerning the CSP part, we have assumed that the behaviour definitions of CSU_A and CSU_C are composed with the context by means of an operator that is monotone with respect to refinement in the timed failures model. As a consequence of this monotonicity, the behaviour definition of $Comp_A$ is refined by the behaviour definition of $Comp_C$.

Altogether, we can conclude that $Comp_A$ is refined by $Comp_C$ because of the monotonicity of the parallel operator composing the Z part and the CSP part. Given the transitivity of the refinement relation, we can reduce the obligation to prove refinement between two compound specification units aggregating an arbitrary number of specification units within the same context to showing that the refinement relation holds between the individual specification units being aggregated.

The reasoning with regard to indexed aggregation is analogous. We omit it since it does not provide further insight.

10.2.2 Extension

In contrast to aggregation, it is not reasonable to lift the refinement relation from the level of single specification units to extension. This is the case because extension is not a means for structuring large specifications like aggregation, but rather a means for achieving reuse.

If we consider a compound specification unit $Comp_A$ that extends the specification unit CSU_A, then the definitions of CSU_A are extended by new entities and constraints. The original entities and constraints and the additional ones constitute a single "unit," and it makes absolutely no sense to retain this extension structure in the compound specification unit refining $Comp_A$.

10.3 Relationship between Forward and Backward Simulations and Refinement in the Timed Failures/States Model

The aim of this section is to prove that two concrete specification units, say CSU_A and CSU_C, which are linked by a retrieve specification unit, say *RSU*, are related by refinement in the timed failures/states model if

1. the CSP part of CSU_C is a refinement of that of CSU_A in the timed failures model and
2. the schema *RetrState* of *RSU* defines a forward or backward simulation between the data states of CSU_A and CSU_C according to the definitions of Section 10.1.3.

In contrast to Smith and Derrick [2001] we are not able to directly adopt the results of Josephs [1988], namely the link between upward and downward simulations that relate state-transition systems and refinement in the failures–divergences model of CSP. The reason is twofold. Firstly, we must prove the refinement relationship in a different, more elaborate semantic model: the timed failures model of timed CSP. Secondly, it is not as obvious as in the approach of Smith and Derrick [2001] that the mapping from Z specifications to the timed failures semantics as defined in Chapter 9 directly corresponds to the mapping from state-transition systems to the failures–divergences model as defined by Josephs.

Let CSU_A and CSU_C be concrete specification units, and let RSU be a retrieve specification unit, which defines retrieve relations between CSU_A and CSU_C. Let further $RSU.RetrState$ define a forward or backward simulation between the data states of CSU_A and CSU_C. We assume that the Z parts of CSU_A and CSU_C do not contain any contradictions (i.e., they are consistent) and that they do not define any constraints on the symbols declared by the Z section *Reals* discussed on p. 122.

Our goal is to prove that

$$timedfailures\,states_C\ [\![\, CSU_C \,]\!] \subseteq Retr_{Interface/State}(\!|\{RSU\}|\!)(\!|timedfailures\,states_C\ [\![\, CSU_A \,]\!]\,|\!)$$

or, informally speaking, that CSU_C refines CSU_A with respect to RSU in the context of the open system view. The above condition is the subject of Theorem 10.9. All other theorems and lemmata are intermediate steps towards this final result.

Within the Theorems 10.2–10.8 (and the lemmata in Appendix C needed to prove them), let $\mathcal{M}_\mathcal{C}$, $\mathcal{M}_\mathcal{A}$ and $\mathcal{M}_\mathcal{R}$ be arbitrary models satisfying

$$\mathcal{M}_\mathcal{C} \in [\![\, CSU_C.Zpart \,]\!]^\mathcal{Z}\ OpsPreds;\ \mathcal{M}_\mathcal{A} \in [\![\, CSU_A.Zpart \,]\!]^\mathcal{Z}\ OpsPreds;$$
$$\mathcal{M}_\mathcal{R} \in [\![\, RSU.Zpart \,]\!]^\mathcal{Z}\ Root.$$

That is, these theorems are implicitly universally quantified.

Moreover, we build our theorems and lemmata in this section on the following definitions by equivalence.

$$HSC == Histories(\!|\{(\mathcal{M}_\mathcal{C}, CSU_C.OPS)\}|\!);\ HSA == Histories(\!|\{(\mathcal{M}_\mathcal{A}, CSU_A.OPS)\}|\!)$$

Theorem 10.2

Considered under the relation $Retr_{hist}$, each history derived from CSU_C is also a history that can be derived from CSU_A. Moreover, for all pairs of histories derived from CSU_C and CSU_A being related by $Retr_{hist}$, the sets of operation-related events constituting possible extensions are equal.

$$\begin{array}{l}
HSC \subseteq Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|HSA|\!) \\
\wedge \\
\forall h_c : HSC;\ h_a : HSA \mid (h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet \\
\quad Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|ev_of_op(\!|\{op : POP_APP \mid \exists h_a^* : HSA \mid (h_a, h_a^*) \in PreHist \bullet \\
\qquad\qquad events_of\ h_a^* = (events_of\ h_a) \frown \langle op \rangle\}|\!)|\!) \\
\quad = ev_of_op(\!|\{op : POP_APP \mid \exists h_c^* : HSC \mid (h_c, h_c^*) \in PreHist \bullet \\
\qquad\qquad events_of\ h_c^* = (events_of\ h_c) \frown \langle op \rangle\}|\!)
\end{array}$$

The conjuncts of the above theorem are immediate consequences of our definition of forward and backward simulations in Section 10.1.3. First, the definitions of forward and backward simulations ensure that the abstract description can "simulate" each behaviour of the concrete description. This is expressed by the first conjunct. Second, Condition 1 of these definitions imply that the sets of operations whose preconditions are satisfied with respect to a corresponding pair of concrete and abstract data states are equal. This is expressed by the second conjunct.

Theorem 10.3
Considered under the relation $Retr_{tr}$*, the trace derived from any history of* CSU_C *is related to the trace derived from any history of* CSU_A*, if these histories are related by* $Retr_{hist}$*.*

$$\forall h_c : HSC;\ h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet traces_Z\, h_c \in Retr_{tr}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{traces_Z\, h_a\}|\!)$$

This is an immediate consequence of the definition of $traces_Z$ *in Section 9.2.4.*

Theorem 10.4
Considered under the relation $Retr_{fail}$*, the set of failures derived from any history of* CSU_C *is a subset of the set of failures derived from any history of* CSU_A*, if these histories are related by* $Retr_{hist}$*.*

$$\forall h_c : HSC;\ h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$failures_Z(\!|\{(h_c, HSC)\}|\!) \subseteq Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|failures_Z(\!|\{(h_a, HSA)\}|\!)|\!)$$

The theorem is proved in Appendix C.

Theorem 10.5
For each history h_c *derived from* CSU_C *there is a corresponding history* h_a *derived from* CSU_A *such that the timed failures associated with* h_c *are a subset of the timed failures associated with* h_a*—when being considered under the relation* $Retr_{hist}$*.*

$$\forall h_c : HSC \bullet \exists h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$timedfailures_Z(\!|\{(h_c, HSC)\}|\!) \subseteq Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|timedfailures_Z(\!|\{(h_a, HSA)\}|\!)|\!)$$

The theorem is proved in Appendix C.

Having treated the timed failure component of the meaning associated with concrete specification units, we turn our attention to the timed state component.

Within Theorem 10.6 (and in the lemmata in Appendix C needed to prove it), let $h_c \in HSC$ and $h_a \in HSA$ be arbitrary such that $(h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)$. Furthermore, we build our theorems and lemmata in the remainder of this section on the following definitions by equivalence.

$$TFA == timedfailures_Z(\!|\{(h_a, HSA)\}|\!);\ TFC == timedfailures_Z(\!|\{(h_c, HSC)\}|\!)$$

Theorem 10.6
Considered under the relation $Retr_{tst}$*, the set of timed states that can be derived from any timed failure of* CSU_C *is a subset of the set of timed states that can be derived from any timed failure of* CSU_A*, if these timed failures are related by* $Retr_{tf}$*.*

$$\forall s_a, s_c : TimedTrace;\ \aleph_a, \aleph_c : TimedRefusal \mid (s_a, \aleph_a) \in TFA \wedge (s_c, \aleph_c) \in TFC \wedge ((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet timed\ states_Z(\!|\{h_c\}|\!)(\!|\{(s_c, \aleph_c)\}|\!) \subseteq Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|timed\ states_Z(\!|\{h_a\}|\!)(\!|\{(s_a, \aleph_a)\}|\!)|\!)$$

The theorem is proved in Appendix C.

Theorem 10.7
Considered under the relation $Retr_{Interface/State}$, the set of timed failures and timed states associated with CSU_C by $tfs_of_model_{INT}$, which is the auxiliary semantic relation defined on p. 144, is a subset of the set of timed failures and timed states associated with CSU_A.

$$tfs_of_model_{INT}(\!|\{(CSU_C, \mathcal{M}_C)\}|\!) \subseteq Retr_{Interface/State}(\!|\{RSU\}|\!)(\!|tfs_of_model_{INT}(\!|\{(CSU_A, \mathcal{M}_A)\}|\!)|\!)$$

The following two assumptions, which are needed to prove Theorem 10.7, concern the CSP parts of CSU_A and CSU_C. They formalise the first condition expressed informally on p. 164.

Assumption 10.3
Firstly, we assume that the behaviour definition in CSU_C is a timed failures refinement of that in CSU_A.

$$timed\,failures_{ETCSP}\ [\![\ CSU_C.Behav\ Behaviour\ ]\!]_{(CSU_C.Behav,(\mathcal{M}_C,\mathcal{M}_C),CSU_C.I \cup CSU_C.L)} \subseteq Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|timed\,failures_{ETCSP}\ [\![\ CSU_A.Behav\ Behaviour\ ]\!]_{(CSU_A.Behav,(\mathcal{M}_A,\mathcal{M}_A),CSU_A.I \cup CSU_A.L)}|\!)$$

Assumption 10.4
Secondly, we assume that the environmental assumption of CSU_C is weaker than that of CSU_A.

$$Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|timed\,failures_{ETFM}\ [\![\ CSU_A.EA\ ]\!]_{(\mathcal{M}_A,CSU_A.I \cup CSU_A.L)}|\!) \subseteq timed\,failures_{ETFM}\ [\![\ CSU_C.EA\ ]\!]_{(\mathcal{M}_C,CSU_C.I \cup CSU_C.L)}$$

Theorem 10.7 is proved in Appendix C.

Theorem 10.8
Considered under the relation $Retr_{Interface/State}$, the set of timed failures and timed states associated with CSU_C by $tfs_of_model_{EXT}$, which is the auxiliary semantic relation defined on p. 145, is a subset of the set of timed failures and timed states associated with CSU_A.

$$tfs_of_model_{EXT}(\!|\{(CSU_C, \mathcal{M}_C)\}|\!) \subseteq Retr_{Interface/State}(\!|\{RSU\}|\!)(\!|tfs_of_model_{EXT}(\!|\{(CSU_A, \mathcal{M}_A)\}|\!)|\!)$$

The theorem is proved in Appendix C.

Theorem 10.9
The final theorem contains the refinement condition given in Definition 10.3.

$$\mathit{timed\,failures\,states}_C \llbracket CSU_C \rrbracket \subseteq \mathit{Retr}_{\mathit{Interface/State}} (\!| \{RSU\} |\!) (\!| \mathit{timed\,failures\,states}_C \llbracket CSU_A \rrbracket |\!)$$

Let *Reals* denote the conjuncts in the definition of $\mathit{timed\,failures\,states}_C \llbracket _ \rrbracket$ on p. 145 related to the mapping of the symbols representing the real numbers to the corresponding entities of the semantic universe.

$((s_c, \aleph_c), tst_c) \in \mathit{timed\,failures\,states}_C \llbracket CSU_C \rrbracket$

$\vdash$ [def. $\mathit{timed\,failures\,states}_C \llbracket _ \rrbracket$]

$\exists \mathcal{M}_C : \llbracket CSU_C.Zpart \rrbracket^{\mathcal{Z}} \mathit{OpsPreds} \mid \mathit{Reals}(\mathcal{M}_C) \bullet$
$\quad ((s_c, \aleph_c), tst_c) \in \mathit{tfs_of_model}_{EXT} (\!| \{(CSU_C, \mathcal{M}_C)\} |\!)$

$\vdash$ [Theorem 10.8]

$\forall \mathcal{M}_{\mathcal{A}} : \llbracket CSU_A.Zpart \rrbracket^{\mathcal{Z}} \mathit{OpsPreds} \bullet$
$\quad ((s_c, \aleph_c), tst_c) \in \mathit{Retr}_{\mathit{Interface/State}} (\!| \{RSU\} |\!) (\!| \mathit{tfs_of_model}_{EXT} (\!| \{(CSU_A, \mathcal{M}_{\mathcal{A}})\} |\!) |\!)$

$\vdash$ [consistency of Z part of CSU_A; symbols related to reals are not restricted by Z part]

$\exists \mathcal{M}_{\mathcal{A}} : \llbracket CSU_A.Zpart \rrbracket^{\mathcal{Z}} \mathit{OpsPreds} \mid \mathit{Reals}(\mathcal{M}_{\mathcal{A}}) \bullet$
$\quad ((s_c, \aleph_c), tst_c) \in \mathit{Retr}_{\mathit{Interface/State}} (\!| \{RSU\} |\!) (\!| \mathit{tfs_of_model}_{EXT} (\!| \{(CSU_A, \mathcal{M}_{\mathcal{A}})\} |\!) |\!)$

$\vdash$ [def. $\mathit{timed\,failures\,states}_C \llbracket _ \rrbracket$]

$((s_c, \aleph_c), tst_c) \in \mathit{Retr}_{\mathit{Interface/State}} (\!| \{RSU\} |\!) (\!| \mathit{timed\,failures\,states}_C \llbracket CSU_A \rrbracket |\!)$

□

To conclude, we have demonstrated that any refinement of the Z part of a concrete specification unit—characterised by a forward or backward simulation—results in a refinement of the overall specification unit in the timed failures/states model of RT-Z. This allows us to refine concrete specification units in the context of the open system view.

In the closed system view, by contrast, only the timed failures at the external interface of a concrete specification unit are taken into consideration—ignoring the evolution of the data state. Given our above reasoning in the context of the open system view, the corresponding proof in the context of the closed system view is trivial.

Part IV

Discussion

Chapter 11

Comparison with Related Formalisms

In this chapter, we contrast RT-Z with those approaches introduced in Chapter 4 that are related most closely to RT-Z and that have strongly influenced its design.

11.1 CSP-OZ

In this section, we contrast RT-Z and CSP-OZ and discuss their benefits and drawbacks. The most relevant differences between CSP-OZ and RT-Z can be traced back to the following two aspects: the choice of similar but different base formalisms and the membership in different classes of integrated formalisms.

The most evident difference between CSP-OZ and RT-Z is the integration of similar but distinct formalisms: Object-Z instead of Z and (untimed) CSP instead of timed CSP. This entails two immediate consequences.

- Since (untimed) CSP is not built on a model of real-time, CSP-OZ is not directly able to deal with real-time constraints.
- CSP-OZ is able to make use of Object-Z's structuring mechanisms: classes and their objects, inheritance and object instantiation.

As already indicated in Chapter 4, CSP-OZ is a monolithic integration according to our proposed classification scheme. This classification is primarily due to the CSP part of CSP-OZ classes, which is constituted by the intermediate language CSP_Z. In CSP_Z, the Z and CSP notations are integrated very closely. The feature of the CSP_Z notation that particularly represents the monolithic nature of CSP-OZ is the schema prefixing operator [Fischer, 2000, p. 52]. This operator is similar to state guards of TCOZ: it allows an arbitrary schema expression to play the role of an event.

Thus, all advantages and disadvantages that are discussed in general with respect to the relationship between monolithic and conserving integrations in Chapter 5 apply in particular to CSP-OZ and RT-Z.

In the remainder of this section, we discuss specific differences between the two integrated formalisms.

Supported Phases. CSP-OZ and RT-Z focus on different phases of the development process.

CSP-OZ, on the one hand, is well-suited for the later development phases, where CSP-OZ supports the design phase and where the translation from CSP-OZ to Java [Fischer, 2000, Part 3] supports parts of the implementation phase. However, CSP-OZ does not offer abstract language features for the early requirements phases: there is no counterpart in CSP-OZ for RT-Z's abstract specification units.

RT-Z, on the other hand, primarily focuses on the earlier phases, where abstract specification units support the requirements phases and where concrete specification units support the design phases. The derivation of code, however, is not covered.

System Model. CSP-OZ and RT-Z have different underlying system models. More specifically, the two formalisms have different pictures of how systems are decomposed into components and how components interact with each other.

In CSP-OZ, channels of class interfaces are related to operations, i.e., the occurrence of an event on a channel is associated with the application of an operation in the corresponding objects. The identification of external channels and operations results in a *strong coupling* of interacting objects: two objects connected by a channel must contain a pair of equally named operations, and the parameters of these operations must also be equally named. This means that the interface between objects is broad, i.e., many details concerning internal properties of objects are globally visible. As Mahony and Dong [1998a] pointed out, this leads to the problem that the configuration of objects within networks is largely fixed by internal aspects of the objects (operation and parameter names) rather than by structural aspects.

Environmental Assumptions. CSP-OZ does not provide any notation to explicitly[1] specify assumptions about the behaviour of the environment of a class with respect to the channels of its interface.

Dynamic Creation of Objects. CSP-OZ allows the dynamic creation of objects during the evolution of a system. While this undeniably increases the flexibility of specifying a dynamic system, the fact that the collection of active objects and their configuration can change dynamically is an enormous complication with respect to verification (in the cases in which dynamic creation is used).

Semantic Definition. Comparing the semantic definitions of CSP-OZ and RT-Z, we must conclude that the semantics of CSP-OZ is defined on a higher level of abstraction and in a less complex way. Both the higher level of abstraction and the lower complexity rest on the notational power of the intermediate language CSP_Z of CSP-OZ, which is used to express the semantics of CSP-OZ classes. In particular, they rest on the ability of CSP_Z to closely integrate process and Z expressions and the ability to parametrise CSP_Z process expressions

[1] Of course, environmental assumptions can always be expressed by incorporating the description of the environment into the system specification and by using appropriate naming conventions. However, this approach is not backed up semantically.

by arbitrary Z expressions. We argue in the following that these powerful notational abilities are not backed up semantically in a sound manner and should thus, strictly speaking, not have been used in the semantic definition of CSP-OZ.

We first deal with the parametrisation of process definitions. In CSP_Z, process definitions can be parametrised by arbitrary Z expressions, e.g.,

$$P(x : S) \stackrel{c}{=} a!x \rightarrow Stop.$$

Fischer fixed the meaning of parametrised process definitions by giving syntactic translation rules into Z. The above definition, e.g., is translated into the axiomatic definition

$$\begin{array}{|l} P : S \rightarrow PROCESS \\ \hline \forall x : S \bullet P(x) = a!x \rightarrow Stop \end{array}$$

Consider the right hand side of the above equation: syntactically, it is a Z expression containing the terminals of the CSP process syntax $\rightarrow$ and *Stop*. To fix the semantics of the above axiomatic definition, the application of the semantic function for Z expressions to the right hand side of the equation

$$[\![a!x \rightarrow Stop]\!]^{\epsilon} M = ?$$

needed to be defined. However, as elaborated in the following, the semantic function $[\![_]\!]^{\epsilon}$ is not defined for process expressions.

Let us address the close integration of process and Z expressions in CSP_Z. In contrast to the approach pursued by Fischer [2000, Section 3.1], we think that Z expressions and process expressions must be treated as separate syntactic units in the grammar of CSP_Z. Otherwise, the semantic function $[\![_]\!]^{\epsilon}$ of the Z Standard for evaluating expressions had to be extended for process expressions. This would amount to define $[\![_]\!]^{\epsilon}$ for all productions of the extended grammar involving process expressions, e.g.,

$$[\![P \Box Q]\!]^{\epsilon} M = \ldots$$

However, this extension of $[\![_]\!]^{\epsilon}$ to process expressions is not defined in [Fischer, 2000]. It would thus be necessary in CSP-OZ to separate process expressions and usual Z expressions. But this would mean that the syntax of CSP_Z is essentially more restricted than suggested by Fischer.[2] As a consequence, the conciseness and convenience of the semantic definition of CSP-OZ [Fischer, 2000, Chapter 4] is based on features of the CSP_Z notation that are not backed up semantically.

Z vs. Object-Z. Object-Z provides powerful constructs such as inheritance and object instantiation to structure large and complex specifications; this is a contrast to (plain) Z, which does not support any elaborate structuring mechanisms. Since the integrated formalism we aimed at must necessarily provide means for structuring specifications, it was tempting to choose Object-Z rather than Z as the basis for our integration, and indeed this is the approach

[2] For instance, the term $\forall x : T \bullet P(x)$ with P a process term is not defined semantically.

taken in the design of CSP-OZ (and also TCOZ). Let us explain our reasons in the design of RT-Z for choosing Z nevertheless.

In the following, we argue that the semantic definition of Object-Z's structuring mechanisms [Smith, 2000] cannot be reused for the definition of the meaning of the corresponding structuring mechanisms of CSP-OZ. This particularly applies to object instantiation, so we concentrate our discussion on that construct.

In Object-Z, object instantiation is a means for decomposing a complex object into a number of smaller objects. The component objects are declared in the state schema of the compound object. This allows the compound object (among other things) to define constraints on the relationships between state components of several component objects (global state invariants) and to define operations on the overall state space by referring to the operations of the component objects. On the semantic level, the meaning of such a compound object is defined as a function of the meaning values of its component objects; but this semantic function inherently depends on the semantic model of Object-Z. If, however, it is decided to adopt classes (and their objects) as the basic structuring units of an integrated formalism (as is the case in CSP-OZ and TCOZ), then the notation of Object-Z is mixed with the notations of the other base formalisms within a class. The meaning (denotation) associated with a class, then, must necessarily be different from the meaning of an Object-Z class because of the additional information covered by the other base formalisms. But this means that object instantiation in an integrated formalism (considered as a semantic function) operates on new units: the semantic function of Object-Z cannot be used. From a semantical point of view, object instantiation must be defined from scratch.

As a consequence, we decided to choose Z and to build the needed structuring mechanisms on top of the integration of Z and timed CSP. To emphasise it again, the decision to use Z and to build structuring mechanisms on top of the integration of Z and timed CSP does not cause any additional work compared with a decision in favour of Object-Z: the structuring mechanisms must be backed up semantically anyway.

Object Instantiation. According to the above discussion, CSP-OZ needed to redefine object instantiation. Due to the difficulties of defining its meaning, CSP-OZ distinguishes between two different kinds of classes: CSP-OZ classes and Object-Z classes. Object-Z classes encapsulate only Object-Z notation, and they are able to apply object instantiation in the full generality of Object-Z. CSP-OZ classes, on the other hand, integrating Object-Z and CSP notation, use a restricted form of object instantiation: CSP-OZ classes cannot instantiate objects of other CSP-OZ classes in their state schema. In CSP-OZ classes, object instantiation takes place in the process expressions of the CSP part. But in this context, instantiating an object means to aggregate a fully encapsulated entity, which can be accessed only via its external interface. In particular, it is not possible in CSP-OZ classes to specify global invariants on the data states of different instantiated objects which is possible in Object-Z and also in RT-Z.

The basic idea of defining the semantics of CSP-OZ classes was to use a layering approach. The first layer is constituted by the language CSP_Z, a language extending the Z syntax with the language of CSP process expressions. CSP_Z specifications, like Z specifications, have a

flat structure; they are the constituents of classes of CSP-OZ specifications, which add object-oriented mechanisms such as inheritance and object instantiation to CSP_Z. CSP-OZ forms the second layer. The meaning of a CSP-OZ class is defined in terms of transition rules mapping the syntax of CSP-OZ to CSP_Z syntax.

At first glance, the definition of the restricted form of object instantiation for CSP-OZ classes is very elegant: it involves only two additional inference rules for CSP_Z. In RT-Z, by contrast, the effort to define the meaning of simple and indexed aggregation was enormous, see Sections 8.1.4 and 8.1.5. However, we argue in the following that the approach chosen by Fischer to define object instantiation is not sound.

To define the meaning of the restricted form of object instantiation for CSP-OZ classes, which is expressed in the CSP part of a CSP-OZ class in terms of CSP_Z syntax, Fischer [2000] defined the following inference rule on p. 146 (Rule 4.6)

$$\frac{}{(c, M) \xrightarrow{\tau} (proc_{M(c)}, M_0 \oplus \{self \mapsto c\})}$$

which is part of the definition of the hybrid semantics $[\![_]\!]^{\mathcal{O}}$ of CSP_Z. Note that this inference rule refers to the semantic function of CSP-OZ classes *proc*. This entails several consequences.

Firstly, the rule is problematic from a conceptual point of view. The circularity—the semantic function of CSP_Z depends on the semantic function of CSP-OZ and vice versa—counteracts the layering approach. The semantic function *proc* and the concept of object instantiation are not known on the level of single CSP_Z specifications. Moreover, the semantic function *proc* is defined in terms of syntactic transformation rules—on a completely different level of formality than the definition of the hybrid semantics of CSP_Z. In other words, the semantic function of CSP_Z is based on a semantic function with an extremely lower level of formality which is an arguable approach.

Secondly, and even more important, the above inference rule is incorrect, and there appears to be no simple way to repair this incorrectness for the following reason.

The semantic function for CSP_Z process terms has the following signature.

$$[\![_]\!]^{\mathcal{O}} : ProcExpr \rightarrow Model \rightarrow LTS$$

That is, the meaning of a CSP_Z process expression is defined only in the context of a particular model M. Note that each CSP_Z process expression is embedded within an entire CSP_Z specification, which also contains usual Z definitions needed to interpret the process expression. Thus, although Fischer did not define it explicitly, the model M used to interpret the process expression must be a member of the meaning of the entire CSP_Z specification.

Applied to the context of a particular CSP-OZ class, it follows that the meaning of its CSP part—a CSP_Z process expression—must be evaluated in the context of a model M that is a member of the meaning of its Z part.[3] Consider now Rule 4.6: the model M_0 used to interpret the process term $proc_{M(c)}$, which represents the meaning of the class $M(c)$ of which the instantiated object is an instance, is a model of the Z part of the class of the *instantiating* object. However, a correct model to interpret the CSP_Z process term must be related to the Z part of the class of the *instantiated* object. But there is no way to refer to the Z part of the class

[3] which defines, e.g., types that are used in the process expressions of the CSP part

$M(c)$ of the instantiated object, because one can deal only with a single CSP_Z specification on the CSP_Z layer, in which the considered inference rule is defined.

In conclusion, the inference rule for object instantiation is not only problematic from a conceptual point of view but not even sound. The consequences are (alternatively)

- That object instantiation is not supported at all in CSP-OZ classes (even in the restricted form).
- That object instantiation must be dealt with at the CSP-OZ level. That would mean that object instantiation had to be syntactically separated from CSP_Z terms, because otherwise the CSP_Z semantic function would not be applicable; then, transformation rules had to be defined in the style of the inheritance rules.

11.2 TCOZ

In this section, we contrast TCOZ and RT-Z. There are two main differences between TCOZ and RT-Z that entail several relative merits and drawbacks.

- The choice of the language specifying the data-related characteristics: Object-Z versus Z.
- The design decisions underlying the two integrations, leading to the classification of TCOZ as a monolithic integration and of RT-Z as a conserving integration according to the classification scheme discussed in Chapter 5.

These differences and their consequences are discussed in the sequel.

Object-Z [Smith, 2000] is an extension of Z with object-oriented structuring operators, allowing a system specification to be structured into a collection of class specifications. On the one hand, this increases the ability to cope with complex system specifications. TCOZ inherits these structuring operators and consequently this ability. On the other hand, the current version of the semantic model of TCOZ [Mahony and Dong, 1997, 1999a] abstracts from these structuring operators of Object-Z, so the application of inheritance and object instantiation is not backed up semantically. For RT-Z, in contrast, whose base formalism Z does not provide any top-level structuring operators, we needed to define these structuring operators as a part of the integration itself.

In TCOZ, as already indicated, the base formalisms Object-Z and timed CSP are closely integrated, leading to a monolithic formalism rather than to a relatively independent coexistence of two complementary base formalisms (as is the case in RT-Z). The interdependence between the elements of timed CSP and Object-Z is strong: the timed CSP syntax is extensively changed by the introduction of new operators that allow the definition of the dynamic behaviour to refer to the object state. This strong interdependence between the two notations makes it more difficult to reuse the infrastructure of the base formalisms, e.g., tool support, proof support, the semantic models, etc.

On the other hand, the close integration of the Object-Z and timed CSP notations in TCOZ leads to a high notational flexibility and convenience compared with conserving integrations

such as RT-Z. For instance, the possibility to directly refer to state variables within value expressions in timed CSP processes allows in many circumstances a more succinct specification than is possible in RT-Z. As another example, the ability to associate operation parameters with external channels avoids the need to explicitly specify aspects of parameter communication in the process definition, in cases where the information of the exact order and time instants of parameter communications is not relevant. As mentioned, from a mere notational point of view the structuring operators provided by TCOZ are very elaborated, in any case superior to the concepts of simple and indexed aggregation offered by RT-Z.

Semantic Models

We contrast the approaches to defining the semantic models of TCOZ and RT-Z. In the following we emphasise the main differences between these approaches and the resulting merits and drawbacks.

Invocation/Termination Events vs. Sequences of Update Events. TCOZ is a monolithic integration. Because of the chosen syntactic integration—each operation schema can be referred to as a process—operations and processes must be identified semantically. The immediate consequence was to identify each operation schema with a set of sequences of so-called update events, where an update event represents the update of a single state variable. We contrast the TCOZ approach of associating each operation application with a sequence of update events with the RT-Z approach of associating each operation application with an invocation and termination event.

The abstraction of interpreting an operation application as a sequence of updates of state variables is, from our point of view, on a fairly concrete level, not adequate for the needs of the requirements and early design phases. There, one is in general not interested in making statements or reasoning about the changes of individual state variables. Certainly, a real implementation is not able to perform a complete state change instantaneously; therefore, interpreting an operation application as a sequence of state variable updates is adequate for the description of an implementation on a very concrete level. But even on a concrete level it is not really adequate to consider updates of single state variables as atomic events: if one considers a sequence of elements as a single state variable, it is certainly not possible to update all elements of the sequence instantaneously; therefore, the advantage that the TCOZ semantic model is closer to real implementations is only of a relative nature.

Moreover, not considering state changes as atomic events means that the concept of state invariants, which is central to Z, cannot be adopted in TCOZ, because a potential state invariant would be valid only between different operation applications. Furthermore, the identification of operations with sequences of update events appears not to be consistent with the notion of Z data refinement. An abstract state with n state components can be refined in Z by a concrete state with m ($\neq n$) state components; but in the TCOZ model, an operation application in the abstract system is represented by a sequence of n events, whereas in the concrete system it is represented by a sequence of m events, between which the CSP refinement notion cannot hold.

Parallel Composition. The TCOZ semantics allows parallel processes (within a single object) to apply operations in parallel.[4] The resulting effect on the object state is that the parallel processes have to synchronise on the individual updates of the state variables that are in the intersection of the delta lists of their operation schemas. In other words, only those value changes of state variables in the delta lists of concurrently applied operations are allowed by the whole process that are individually allowed by all operations. If the concurrently applied operations specify inconsistent constraints for at least one common variable, the whole parallel process deadlocks. The resulting constraint on the state change is hence in effect a conjunction of the constraints of the operations that are composed in parallel.

At first glance, the ability to arrange different operations of a single object in parallel processes is a powerful feature of TCOZ, not provided by RT-Z. However, since operations in parallel operations have to synchronise on all update events, the specified behaviour is not really parallel.[5]

$$(P1;\ Op1;\ Q1) \parallel (P2;\ Op2;\ Q2) = (P1 \parallel P2);\ (Op1 \wedge Op2);\ (Q1 \parallel Q2)$$

Let us illustrate the shortcomings caused by this semantic definition by means of three examples.

The parallel composition of the operations $Op1a$ and $Op1b$ in the first example seems to be unproblematic from the semantic point of view, because their delta lists are disjoint, so their state changes do not interfere.

$$State == [\, x, y : \mathbb{N} \,]$$
$$Op1a == [\, \Delta(x) \mid x' \neq x \,]; \qquad Op1b == [\, \Delta(y) \mid y' = x \,]$$
$$MAIN \mathrel{\widehat=} Op1a;\ e \rightarrow Skip \parallel e \rightarrow Op1b$$

According to the TCOZ semantics, the state changes caused by the operations refer to the pre state that is present when the whole parallel process starts. Thus, the value of y after the termination of $Op1b$ equals the value of x before the process starts. This seems to contradict intuition, because $Op1a$ must have completed its state change before $Op1b$ is invoked (because of synchronisation e). It would thus seem more intuitive if $Op1b$ would assign y the value of x that $Op1a$ has computed.

The operations in the second example have a common element in their delta list, namely x.

[4] This is a contrast to RT-Z, where parallel operations must be encapsulated in different specification units.

[5] For the TCOZ definition of parallel composition to work, it would in fact be necessary that each operation schema is mapped semantically to the *deterministic* choice of the order and values of the update events allowed by the operation schema. In contrast, the TCOZ semantics actually specifies a *nondeterministic* choice. This means that a parallel composition of operations can arbitrarily deadlock which does not seem to be the intended interpretation of the parallel application of operations. To define a deterministic choice with respect to the order and the values of state variable updates, however, would make the semantic definition of TCOZ extremely more difficult.

$State == [\, x : \mathbb{N} \,]$
$Op2a == [\, \Delta(x) \mid x' > x \,]; \qquad Op2b == [\, \Delta(x) \mid x' < x \,]$
$MAIN \mathrel{\widehat=} Op2a;\; e \rightarrow Skip \parallel e \rightarrow Op2b$

The operations define an inconsistent state change for this variable so a deadlock results. Intuition would suggest that *Op*2*a* is performed before *Op*2*b* such that two changes of x take place.

In the last example, the state change that is specified by the whole parallel composition is that the value 4 is assigned to x. The parallel composition degenerates to a conjunction of the two operations.

$State == [\, x : \mathbb{N} \,]$
$Op3a == [\, \Delta(x) \mid x' > 3 \,]; \qquad Op3b == [\, \Delta(x) \mid x' < 5 \,]$
$MAIN \mathrel{\widehat=} Op3a \mid\mid\mid Op3b$

Also this is counterintuitive because two seemingly independently applied operations have to agree on the common state change. The realisation of such a specification is absolutely unclear, because in order to agree on a common state change the parallel subprocesses must somehow communicate (via a channel not recorded in the list of synchronisation channels).

To conclude, while it would be useful to permit the parallel application of operations within a single object, the particular semantic definition of TCOZ does not achieve this goal.

Structuring Mechanisms. Object-Z extends Z with different concepts and operators beyond the pure class construct, e.g., inheritance and object instantiation. The semantic definition of TCOZ does not refer to the semantics of Object-Z at all, but it refers only to the Z semantics of schemas and expressions. So, it is not clear how these additional constructs, which are the very reason to prefer Object-Z over Z, are defined. According to the TCOZ language description and case studies, these additional operators of Object-Z are part of TCOZ and are the only means for decomposing a TCOZ specification.

Embedding of Z Semantics. When comparing the semantic definitions of TCOZ and RT-Z, the embedding of the Z semantics into the overall semantic model appears to be more seamless in TCOZ, because the semantics of Z is referred to at only two points: when referring to the meaning of an operation and when referring to the meaning of a state guard. The process of deriving the timed failures semantics of the Z part of an RT-Z specification unit is undeniably more complex; however, the TCOZ definition does not take into account that the meaning of an operation schema cannot be derived in isolation, because it usually depends on the state schema and other global constants. So, the meaning of an operation schema can be defined only in the context of a complete Z specification, and if this was taken into account in TCOZ, the embedding of the Z semantics would have a comparable complexity.

Finally, there are some features that have played an important rôle in the design of RT-Z, but have no counterpart in TCOZ.

- TCOZ classes can be compared with concrete specification units of RT-Z. There are, however, no counterparts of abstract specification units in TCOZ. This makes it difficult to use TCOZ in the early phases of a development process.
- In TCOZ, it is not possible to specify assumptions about the behaviour of the environment at the external interface (cf. `ENVIRONMENTAL ASSUMPTIONS` section in RT-Z).
- In TCOZ, channels are type-heterogeneous, i.e., not associated with a type. This is a restriction of the interface specification whose reason is not clear to us. We think that the type of values transmitted via a certain channel is a very relevant piece of information.

11.3 Object-Z / CSP

In this section, we contrast RT-Z and the integration of Object-Z and CSP proposed by Smith and Derrick [2001] and discuss their benefits and drawbacks.

As already indicated in Chapter 4, although both integrated formalisms are conserving integrations, the underlying principles of integrating the base notations are different. While Object-Z/CSP separates its base notations in different layers of an overall system specification, RT-Z separates the base notations in different parts of a single component specification. This entails the following consequences.

The Object-Z/CSP integration is a conserving integration in the true sense of the word, because the base notations are *strictly* separated from each other. This means that the benefits attributed to the class of conserving integrations in Chapter 5 apply maximally.

The drawback of the layering approach, however, is that CSP is used to specify only architectural aspects, i.e., the configuration of system components and their interaction. In other words, the CSP notation is not used on the (fine-grained) component level, e.g., to fix the temporal ordering of operation applications.

11.4 LOTOS

LOTOS, as introduced in Section 4.4, has several strengths that have strongly influenced the design of RT-Z.

A major strength of LOTOS are its modularity concepts:

- LOTOS allows the hierarchical decomposition of a system specification into component descriptions (process definitions).
- Process definitions are parametrised and can thus be instantiated differently in distinct contexts, which supports the reuse of process definitions.
- The interface of a process is explicitly specified in terms of a list of formal gates.

```
process P1 (pre : State) :=
  In1 ?input:T1 ; ...
  choice post : State, output : T2 []
    [Op1(pre, post, input, output)] --> ... ;
        Out1 !output ; P2(post)
  []
  In2 ?input:T1 ; ...
  choice post : State, output : T2 []
    [Op2(pre, post, input, output)] --> ... ;
        Out2 !output ; P3(post)
  [] ...
endproc

process P2 (pre : State) := ...
process P3 (pre : State) := ...

sorts: State, T1, T2
opns: Op1, Op2, Op3 : State, State, T1, T2 --> Boolean
eqns: ...
```

Figure 11.1: Modelling a component as an EFSM.

These modularity concepts have influenced the design of the structuring operators of RT-Z and also the design of the internal structure of specification units.

A very important contribution of LOTOS is the integration of the concepts "synchronisation" and "value transmission." This is achieved by associating actions with data types and by associating synchronisations with the agreement on concrete values. This approach has been adopted in RT-Z.

LOTOS has, on the other hand, several shortcomings that make it, from our point of view, less adequate for the application domain of real-time embedded systems than RT-Z.

One serious disadvantage of LOTOS is that it does not provide any concept for explicitly modelling the *data state* of a component. In general, the behaviour of a component depends on the data values that have been communicated with the environment. Therefore, it must be possible in LOTOS to reflect this value dimension of the past interaction of a component. Providing no concepts to directly model the data state of a component, LOTOS must simulate this data state by means of the model of extended finite state machines (EFSM). The specification of a component as an EFSM is illustrated in Figure 11.1.

An EFSM comprises a finite set of control states, transitions between these control states and a data state associated with each control state. Each control state of the EFSM is modelled by a separate LOTOS process (`P1`, `P2`, `P3`), and the data state is represented by introducing additional data parameters (`pre:State`) for all these processes. Each transition in the EFSM coincides with a transformation of the underlying data state; it is guarded by an external in-

teraction and can depend on a certain condition on the data state. A transition in the EFSM is modelled by instantiating the process representing the destination control state with the transformed data state by the process representing the source control state. The transformation of the data state that coincides with a transition in the EFSM is described by a relation between the pre state, post state, inputs affecting the transformation and outputs resulting from the transformation. In the algebraic language of LOTOS, ACT ONE, such a relation is modelled by a boolean-typed function (`Op1, Op2, Op3`). This modelling is achieved in the data type part of the LOTOS specification by means of a set of conditional equations stating the properties the relation must satisfy.[6]

The outlined approach to simulating a data state by additional parameters and mutually recursive process instantiations might be adequate in relatively simple cases in which the dynamic behaviour is not complex. A complex dynamic behaviour of a data-state–dependent component, however, results in a complex EFSM with a variety of control states and transitions and therefore in a large amount of LOTOS processes. This would it make hard to understand the specification; the process algebra LOTOS was not designed to construct EFSM models (like for example the ISO language ESTELLE [ISO, 1997]).

The meaning of basic and full LOTOS is defined by an operational semantics identifying each process with a labelled transition system (LTS). As a consequence of the use of an operational semantics, properties of processes can itself be expressed only in terms of processes, where a refinement relationship must be proved between the process denoting a specification (model) and the process denoting a property. Compared with a process algebra whose meaning is defined by means of a denotational semantics, such as CSP, this way of specifying properties of processes is on a rather concrete level. Specifying properties in terms of predicates like in CSP is more abstract and more comprehensible.

LOTOS does not aim to specify real-time behaviour and thus does not provide any constructs to do so. There are several proposed enhancements of LOTOS with real-time concepts [Bolognesi et al., 1994, Bryans et al., 1995], but none of them has established as *the* real-time enhancement of LOTOS.

The language used by LOTOS for specifying data types, ACT ONE, is an algebraic specification language. Its undeniable advantage is the high level of abstraction in specifying abstract data types; only properties are specified without fixing design decisions. This is particularly useful in the early phase of requirements specification. On the other hand, ACT ONE does not provide language constructs that are concrete enough to fix design decisions in the later design phase.

[6] We are not completely able to assess the adequacy of specifying a state transition by means of a set of conditional equations. We assume, however, that this is less adequate than the operators provided by a model-oriented approach such as Z.

Chapter 12

Case Studies

We present two case studies in order to illustrate RT-Z, as defined in the previous two parts, more intuitively. Another aim of the case studies is to demonstrate the appropriateness of RT-Z for the domain of real-time embedded systems.

The first case study focuses on the ability of RT-Z to cope with real-time and complexity, and the second case study deals with a safety-critical application.

12.1 Multi-lift System

The first case study illustrating RT-Z is the formal specification of a multi-lift system,[1] which we have chosen for the following reasons. Firstly, a lift system is something people are familiar with, so we are able to concentrate on discussing the features of RT-Z. Secondly, the multi-lift system has the appropriate level of complexity. On the one hand, it is complex enough to demonstrate the decomposition concepts of RT-Z, it is an inherently concurrent system, and there are several real-time requirements to be met. On the other hand, it allows us to apply an appropriate abstraction that is not overwhelming. Last but not least, the multi-lift system has served other formalisms as a case study, e.g., TCOZ introduced by Mahony and Dong [1998a], Object-Z and CSP. The RT-Z case study has been strongly influenced by the TCOZ case study. In Section 12.1.4, we contrast our case study with the TCOZ case study.

12.1.1 Problem Description and System Architecture

We first explain the architecture of the multi-lift system as far as it is needed to define the interface between the part of the system that is to be implemented by software, depicted in Figure 12.1 by the rectangle named *SoftwareController*, and the other system components. This system architecture is the result of the system design and taken as the input to our development process.

[1] Strictly speaking, we do not specify an entire multi-lift system but a distributed software controller of such a system.

The considered multi-lift system consists of several *lifts* moving between the *floors* of a building. Each floor is equipped with a panel of buttons that allow users to make a request to move upward or downward by pressing the corresponding request button.[2] Analogously, each lift is equipped with a panel of request buttons, one for each floor of the building. There is a central external request handler that manages external requests that are currently pending. External requests are caused by pressing a button at a floor panel; they are to be served in a 'first-in-first-out' manner. As soon as a lift becomes idle, i.e., it has no pending internal request, the external request handler delivers the next external request to this lift. Internal requests are caused by pressing a button at a lift panel. According to the above statements, internal requests have priority over external requests: when internal requests are pending, a lift serves the closest internal request in the current move direction.

When a lift enters a floor in order to serve a request, the door of the lift is first opened, then kept open for a constant time period and is closed afterwards. Closing the door might be blocked by an object. The state of the lift door and potential objects blocking a closing door are detected by a door sensor. A door motor serves to execute the commands given by the lift controller. The movement of a lift is controlled by its shaft component. It is not the aim of this case study to formalise aspects of this component, e.g., how the component detects that the next floor is reached, etc. Therefore, this component is considered to be part of the software environment, and we model only the interface to it as well as important assumptions about its real-time behaviour.

The following data type definitions are global with respect to the whole specification and serve, among other things, to specify the kind of information that is communicated with the environment of the software controller and between its components.

$MoveDir ::= Up \mid Down$
$DoorMsg ::= Opened \mid Closed \mid Blocked$
$DoorCmd ::= OpenCmd \mid CloseCmd$

The global constants *floor_num* and *lift_num* are parameters of the specification, denoting the number of floors and lifts in the considered building,

$$
\begin{array}{|l}
floor_num, lift_num : \mathbb{N}_1 \\
\hline
floor_num > 1
\end{array}
$$

and the subsequent sets are needed in order to identify the lifts and floors in the building.

$LID == 1 \mathinner{..} lift_num$
$FID == 0 \mathinner{..} floor_num$
$MFloors == 1 \mathinner{..} (floor_num - 1)$

The following time constants are described in Table 12.1. They reflect characteristics of the real-time behaviour of the lifts and their doors.

$$
| \; t_{close}, t_{open}, t_{wait}, t_{move}, t_{acc/brk}, t_{priority} : \mathbb{T}
$$

[2] Of course, the ground and the top floor are equipped with only one button.

time constant	meaning
t_{close}	time needed to close lift doors provided process of closing is not blocked
t_{open}	time needed to open lift doors
t_{wait}	length of interval in which lift doors must be kept open before they can be closed
t_{move}	time needed by a lift to move from a floor to the next in either direction
$t_{acc/brk}$	additional time needed to accelerate in the start floor and to break in the destination floor, respectively
$t_{priority}$	length of time interval in which internal requests have priority over external requests

Table 12.1: Time constants.

The derived time constant

$$t_{max_int_schedule} == 2 * floor_num * (t_{open} + t_{wait} + t_{close} + t_{move} + 2 * t_{acc/brk})$$

defines the maximal amount of time needed by a lift to go from the ground floor to the top floor and to return back to the ground floor when all intermediate floors are to be served in both directions, provided the doors of the lift are not blocked.

12.1.2 Requirements Specification

Let us first express the requirements to the software controller informally. The following requirements should be understood as a selection from a complete list of requirements.

*Req*1: Internal requests have priority over external requests.

*Req*2: Each internal request is served after at least $t_{max_int_schedule}$ time units provided the doors of the corresponding lift are not blocked.

*Req*3: When there are no pending internal request for at least one lift, the oldest external request is served (FIFO).

*Req*4: After the doors of a lift are opened, they remain open for at least t_{wait} time units.

The interface between the distributed software controller and its environment is defined in a separate specification unit because it is needed in the requirements and the design specification. The specification units representing both the requirements and the design specification are defined as extensions of the following specification unit.

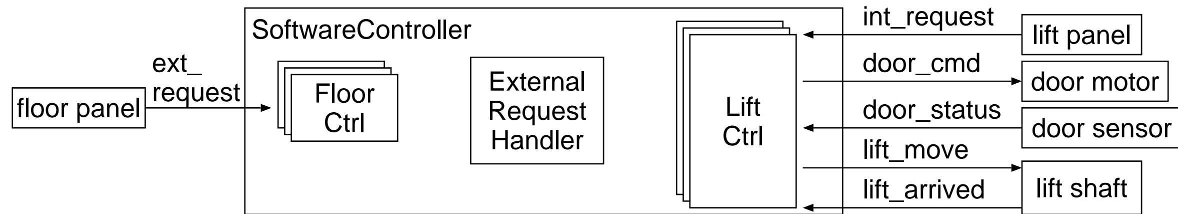


Figure 12.1: Software controller interface.

SPEC. UNIT *SoftwareControllerInterface*

INTERFACE

Figure 12.1 shows the interface between the software controller and its environment. External requests from a floor are transmitted along the channel *ext_request*, where each request incorporates the floor from which it originates and the desired direction of movement. Pushing a floor button causes it to be activated until the request is served. Pushing an already activated button has no consequence. Internal requests from a lift are transmitted along the channel *int_request*, where each request contains the lift from which it originates and the target floor. The door of each lift is linked with its sensor via the channel *door_status*, and it gives commands to its actuator along the channel *door_cmd*. Further, each lift gives its shaft component the command to move to a particular floor via the the channel *lift_move*, and it awaits the completion of such a command on the channel *lift_arrived*.

`port` *ext_request* `domain` $FID \times MoveDir$;
`port` *int_request* `domain` $LID \times FID$;
`port` *door_status* `domain` $LID \times DoorMsg$;
`port` *door_cmd* `domain` $LID \times DoorCmd$;
`port` *lift_move*, *lift_arrived* `domain` $LID \times FID$

The *abstract* specification unit *SoftwareControllerReqs* constitutes the requirements specification of the distributed software controller of the multi-lift system.

SPEC. UNIT *SoftwareControllerReqs* **EXTENDS** *SoftwareControllerInterface*

ENVIRONMENTAL ASSUMPTIONS

Since the door sensors, the door motors and the shaft components of the lifts are part of the environment, we can express only *assumptions* about their behaviour at the software interface. If these assumptions are violated by these components, the software controller is free to behave arbitrarily. These assumptions are formalised by the following predicates.

$EA \mathrel{\widehat{=}} EA1 \land EA2 \land EA3 \land EA4$

First, the door motor of each lift is always supposed to accept each possible command on the channel *door_cmd*.

$EA1 \mathrel{\widehat{=}} \forall\, t : \mathbb{T};\ cmd : DoorCmd;\ lift : LID \bullet door_cmd.(lift, cmd)\ \mathsf{open}\ t$

Second, the shaft component of a lift is assumed to accept the command to move to a floor when not executing a prior moving command.

$$
\begin{array}{l}
EA2 \mathrel{\widehat{=}} \forall\, t : \mathbb{T};\ lift : LID;\ floor : FID \bullet \\
\quad (s \restriction_{ttr} (\{f : FID \bullet lift_arrived.(lift,f)\} \cup \{f : FID \bullet lift_move.(lift,f)\}) \Vert\!\restriction_{ttr} t = \langle\rangle \Rightarrow \\
\qquad lift_move.(lift,floor)\ \mathsf{open}\ t) \\
\quad \land \\
\quad (\mathrm{last}(s \restriction_{ttr} (\{f : FID \bullet lift_arrived.(lift,f)\} \cup \{f : FID \bullet lift_move.(lift,f)\}) \Vert\!\restriction_{ttr} t) \in \\
\qquad \{f : FID \bullet lift_arrived.(lift,f)\} \\
\quad \Rightarrow \\
\qquad lift_move.(lift,floor)\ \mathsf{open}\ t) \\
\quad \land \\
\quad 0 \leq \#(s \restriction_{ttr} \{f : FID \bullet lift_move.(lift,f)\} \restriction_{ttr} t) - \\
\qquad \#(s \restriction_{ttr} \{f : FID \bullet lift_arrived.(lift,f)\} \restriction_{ttr} t) \leq 1
\end{array}
$$

There is a further assumption that is related to the real-time behaviour of the shaft component. Whenever a lift controller transmits a command via the channel *lift_move* to move to a particular floor, the shaft component is supposed to accomplish this command within a particular time interval that depends on the number of floors to be moved.

$$
\begin{array}{l}
EA3 \mathrel{\widehat{=}} \forall\, t : \mathbb{T};\ lift : LID;\ floor : FID \bullet \\
\quad \exists_1\, last : FID \mid \\
\qquad (\exists\, t1 : [0,t) \bullet lift_arrived.(lift, last)\ \mathsf{at}\ t1 \land \\
\qquad\quad (\forall\, t2 : (t1,t);\ f : FID \bullet \neg\, lift_arrived.(lift,f)\ \mathsf{at}\ t2)) \\
\qquad \lor \\
\qquad \neg\, (\exists\, t1 : [0,t);\ f : FID \bullet lift_arrived.(lift,f)\ \mathsf{at}\ t1) \land last = 0 \bullet \\
\qquad lift_move.(lift,floor)\ \mathsf{at}\ t \\
\qquad \Rightarrow \\
\qquad lift_arrived.(lift,floor)\ \mathsf{open\ from}\ (t +_{\mathbb{R}} abs(floor - last) *_{\mathbb{R}} t_{move} +_{\mathbb{R}} 2 *_{\mathbb{R}} t_{acc/brk})
\end{array}
$$

Finally, the process of closing the door of a lift is supposed to be completed after t_{close} time units, provided the door is not blocked by an object.

$EA4 \mathrel{\widehat{=}} \forall\, lift : LID;\ t1 : \mathbb{T} \bullet$
$door_cmd.(lift, CloseCmd)$ at $t1$
$\wedge$
$\neg\ (\exists\, t2 : [t1, t1 + t_{close}] \bullet door_status.(lift, Blocked)$ open $t2)$
$\Rightarrow$
$door_status.(lift, Closed)$ open from $(t1 +_{\mathbb{R}} t_{close})$

BEHAVIOURAL PROPERTIES

The following requirements $Req1$ to $Req4$ correspond to the four requirements described informally.

$Behaviour$ **sat** $Req1 \wedge Req2 \wedge Req3 \wedge Req4$

We first introduce two parametrised process definitions in order to make the definition of the requirements more intelligible. The predicate *ExtPending* expresses that an external request (f, dir) made at time $t1$ is still pending at $t2$, i.e., not served by any lift between $t1$ and $t2$. Similarly, the predicate *IntPending* states that an internal request f made in lift l at $t1$ is still pending at $t2$, i.e., not served between $t1$ and $t2$.

$ExtPending(f, dir, t1, t2) \mathrel{\widehat{=}}$
$ext_request.(f, dir)$ at $t1$
$\wedge \neg\ (\exists\, t : (t1, t2);\ l : LID \bullet lift_move.(l,f)$ at $t)$
$IntPending(l,f, t1, t2) \mathrel{\widehat{=}}$
$int_request.(l,f)$ at $t1$
$\wedge \neg\ (\exists\, t : (t1, t2) \bullet lift_move.(l,f)$ at $t)$

Internal requests have priority over external requests. In other words, when the controller of a particular lift gives the command to go to a particular floor that is related to an external request,[a] there must not be any pending internal request.

$Req1 \mathrel{\widehat{=}} \forall\, t : \mathbb{T};\ l : LID;\ f : FID \bullet$
$lift_move.(l,f)$ at t
$\wedge\ (\exists\, t1 : [0, t);\ dir : MoveDir \bullet ExtPending(f, dir, t1, t))$
$\wedge \neg\ (\exists\, t1 : [0, t) \bullet IntPending(l,f, t1, t))$
$\Rightarrow$
$\neg\ (\exists f2 : FID;\ t1 : [0, t) \bullet IntPending(l,f2, t1, t))$

Each internal request must be served after at least $t_{max_int_schedule}$ time units provided the doors of the corresponding lift are not blocked.

$Req2 \mathrel{\widehat{=}} \forall\, l : LID;\ t : \mathbb{T};\ f : FID \bullet$
$(\forall\, t : \mathbb{T} \bullet \neg\ door_status.(l, Blocked)$ open $t)$
$\wedge\ int_request.(l,f)$ at t
$\Rightarrow$
$(\exists\, t2 : [t, t +_{\mathbb{R}} t_{max_int_schedule}] \bullet lift_move.(l,f)$ live from $t2)$

[a] because all prior internal requests for this floor are already served

When there are no pending internal request for at least one lift, the oldest external request must be served.

$Req3 \mathrel{\widehat=} \forall t, t1 : \mathbb{T};\ f : FID;\ dir : MoveDir \bullet$
$(\exists l : LID \bullet \forall t1 : [0, t);\ f : FID \bullet \neg IntPending(l, f, t1, t))$
$\land\ t1 \leqslant_{\mathbb{R}} t \land ExtPending(f, dir, t1, t)$
$\land \neg (\exists f2 : FID;\ dir2 : MoveDir;\ t2 : \mathbb{T} \bullet t2 <_{\mathbb{R}} t1 \land ExtPending(f2, dir2, t2, t))$
$\Rightarrow$
$(\exists l : LID \bullet lift_move.(l, f)$ live from $t)$

After the doors of a lift are opened, they must remain open for at least t_{wait} time units.

$Req4 \mathrel{\widehat=} \forall t : \mathbb{T};\ l : LID \bullet$
$door_status.(l, Opened)$ at t
$\Rightarrow$
$\neg (\exists t1 : [t, t + t_{wait}] \bullet door_cmd.(l, CloseCmd)$ live $t1)$

This completes the requirements specification. By distributing predicates to the `ENVIRONMENTAL ASSUMPTIONS` and `BEHAVIOURAL PROPERTIES` sections, we have unambiguously stated which requirements are related to the software controller under consideration and which requirements are related to its environment. The predicate language of RT-Z allows us to express all requirements on an abstract level without fixing any implementation aspects.

12.1.3 Design Specification

We now present the architectural design of the software controller of the multi-lift system, see Figure 12.2. We discuss the specification units that constitute the design specification in a top–down order for the high abstraction levels (top-level units) combined with a bottom–up order for the lower abstraction levels. We think that a system is preferably explained by first describing the coarse structures before delving into the details.

SPEC. UNIT *SoftwareControllerDesign* **EXTENDS** *SoftwareControllerInterface*

SUBUNITS

The components of the software controller are the controllers of the floor panels and of the lifts and a component that handles the pending external requests.

`subunit` *floors* `spec. unit` *FloorCtrls*;
`subunit` *lifts* `spec. unit` *LiftCtrls*;
`subunit` *erh* `spec. unit` *ExternalRequestHandler*

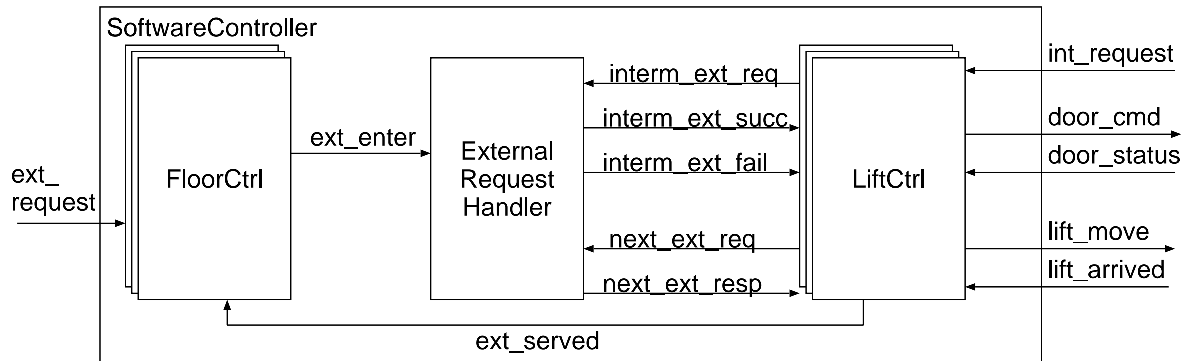


Figure 12.2: Software controller architecture.

LOCAL

The channels that serve to connect the components of the software controller without being part of the external interface are declared in the `LOCAL` section. An external request is passed on by the involved floor controller to the external request handler along the channel *ext_enter*. A lift that has completed an external request reports this to the respective floor controller along the channel *ext_served*, which causes the deactivation of the respective button. Whenever a lift has no pending internal requests, it instructs the external request handler to provide the next external request in its queue via the channel *next_ext_req*, which in turn is delivered by the external request handler via the channel *next_ext_resp*. Whenever a lift schedules a new internal request, it sends a request along the channel *interm_ext_req* whether the external request handler has an external request in its queue between the current and the target floor in the current move direction. This request is replied via the channels *interm_ext_succ* and *interm_ext_fail*, respectively.

`channel` *ext_enter* `domain` $FID \times MoveDir$;
`channel` *ext_served* `domain` $FID \times MoveDir$;
`channel` *interm_ext_req* `domain` $LID \times FID \times FID$;
`channel` *interm_ext_succ* `domain` $LID \times FID$;
`channel` *interm_ext_fail*, *next_ext_req* `domain` LID;
`channel` *next_ext_resp* `domain` $LID \times FID \times MoveDir$

ENVIRONMENTAL ASSUMPTIONS

The environmental assumptions made by the design are identical with that of the requirements specification. They are hence omitted. In general, the design specification is able to weaken the environmental assumptions of the requirements specification.

BEHAVIOUR

The following abbreviation definition introduces *ChannelSet* to denote the set of events that can be communicated along particular channels. It is referred to in the following behaviour definition.

$ChannelSet \mathrel{\widehat=} \{\!|\ next_ext_resp\ |\!\} \cup \{\!|\ next_ext_req\ |\!\} \cup \{\!|\ interm_ext_req\ |\!\}$
$\quad \cup \{\!|\ interm_ext_succ\ |\!\} \cup \{\!|\ interm_ext_fail\ |\!\}$

The process term *Behaviour* defines the configuration of the components of the software controller including their interaction along particular channels. Basically, the components are composed in parallel, synchronising on the given set of internal channels.

$Behaviour \mathrel{\widehat=}$
$\quad floors.Behaviour$
$|[\ \{\!|\ ext_enter\ |\!\} \cup \{\!|\ ext_served\ |\!\}\ ]|$
$\quad (erh.Behaviour\ |[\ ChannelSet\ ]|\ lifts.Behaviour)$

The previous specification unit defines the top-level structure of the software controller. We now switch to the bottom–up order. That is, we first define the basic specification units of which the components of the software controller are composed.

SPEC. UNIT *Button*

The current specification unit defines the behaviour of 'raw' buttons as they can be found on the floors. They are raw in that their interface is not ready to be put into the floor context. Lifting the button interface into this context is the aim of the following specification units.

INTERFACE

A button can be turned on and off represented by events occurring on the channels *turn_on* and *turn_off*. Moreover its status can be requested via the channel *status*, and the current status is reported either via the channel *button_on* or *button_off*.

`port` $status, turn_on, turn_off, button_on, button_off$ `domain` $SYNC$

BEHAVIOUR

$Behaviour \mathrel{\widehat=} Off$

$Off \mathrel{\widehat=} turn_on \rightarrow On$
$\quad \Box\ turn_off \rightarrow Off$
$\quad \Box\ status \rightarrow button_off \rightarrow Off$

$On \mathrel{\widehat=} turn_off \rightarrow Off$
$\quad \Box\ turn_on \rightarrow On$
$\quad \Box\ status \rightarrow button_on \rightarrow On$

SPEC. UNIT *FloorButton*[*dir*]

The previous unit serves to model the behaviour of a button independently of its particular context. The current, parametrised unit serves to lift the *Button* unit into the context of a floor panel, where it has to handle either upward or downward requests, depending on the instantiation of the parameter *dir*.

TYPES & CONSTANTS

The formal parameter *dir*, recorded in the head of the specification unit, is declared formally in the TYPES & CONSTANTS section.

| $dir : MoveDir$

SUBUNITS

subunit *button* spec. unit *Button*

INTERFACE

The current unit lifts a raw button to a button accepting downward/upward requests at a particular floor. Pressing the request button is modelled by an event on the channel *ext_request*. If it is deactivated when being pressed, then it sends an event to the external request handler along the channel *ext_enter*. The occurrence of an event on the channel *ext_served* indicates that the request is served.

port *ext_request*, *ext_enter*, *ext_served* domain *MoveDir*

LOCAL

The interface of a raw button, which is encapsulated by the current unit, is introduced as a list of internal channels.

channel *status*, *turn_on*, *turn_off*, *button_on*, *button_off* domain *SYNC*

BEHAVIOUR

External events occurring at the external interface are passed on to the interface of the raw button, and the resulting reaction of the raw button is passed on to the external interface.

Downward as well as upward requests are transmitted along the channel *ext_request*, where the current unit reacts only to *Down* or *Up* requests, depending on the instantiation of the parameter *dir*. The behaviour of the aggregated raw button is composed in parallel with a process that is responsible for mapping the external to the corresponding internal events.

$Behaviour \mathrel{\widehat=}$
$\quad FB \,|[\, \{status, turn_on, turn_off, button_on, button_off\} \,]|\, button.Behaviour$

$FB \mathrel{\widehat=} (Request \,\Box\, Serviced);\ FB$

$Request \mathrel{\widehat=} ext_request.dir \rightarrow status \rightarrow$
$\quad (button_off \rightarrow turn_on \rightarrow ext_enter.dir \rightarrow Skip$
$\quad \Box\ button_on \rightarrow Skip)$

$Serviced \mathrel{\widehat=} ext_served.dir \rightarrow status \rightarrow$
$\quad (button_on \rightarrow turn_off \rightarrow Skip$
$\quad \Box\ button_off \rightarrow Skip)$

SPEC. UNIT *MiddleFloorCtrl*

The current unit models the controller of a floor between the bottom and top floor. These floors are equipped with a panel consisting of two buttons, handling downward and upward requests, respectively. Accordingly, two button units are aggregated.

SUBUNITS

subunit *up* spec. unit *FloorButton*
  instantiate *dir* with *Up*;
subunit *down* spec. unit *FloorButton*
  instantiate *dir* with *Down*

INTERFACE

The interface of a middle floor controller is identical with the interface of an upward and downward request button, where the two kinds of requests are accepted at the same time.

port *ext_request*, *ext_enter*, *ext_served* domain *MoveDir*

BEHAVIOUR

The behaviour of a middle floor controller is simply the interleaving of the downward and upward request buttons.

$Behaviour \mathrel{\widehat=} up.Behaviour \,|||\, down.Behaviour$

The specification units *TopFloorCtrl* and *BottomFloorCtrl*, modelling the top and bottom floor controllers, are defined in the obvious way and are hence omitted.

SPEC. UNIT *FloorCtrls*

The controllers of the middle floors, the bottom floor and the top floor of the considered building are aggregated by the current unit.

SUBUNITS

The controllers of the middle floors are composed by indexed aggregation, one for each member of the set *MFloors*.

`subunit` *mfloors*(*MFloors*) `spec. unit` *MiddleFloorCtrl*;
`subunit` *tfloor*({*floor_num*}) `spec. unit` *TopFloorCtrl*;
`subunit` *bfloor*({0}) `spec. unit` *BottomFloorCtrl*

Also the controllers of the top and the bottom floor are composed by means of indexed (rather than simple) aggregation. At first glance, this might appear suspicious, because there is only a single instance to be aggregated for each. However, indexed aggregation with singleton sets is used instead of simple aggregation in order to lift the interfaces of the top and bottom floor units automatically into the context.

The declaration introduces a collection of instances of the specification unit *MiddleFloorCtrl*, one for each member of *MFloors*, a single instance of the unit *TopFloor* with the identifier *floor_num* and a single instance of the unit *BottomFloor* with the identifier 0. The interfaces of all instances are obtained by enriching the type of each external channel, say T, with the used index set, yielding $FID \times T$. That is, each individual instance is addressed by using the identifier of the respective instance as the first component of the value transmitted along the considered external channel.

INTERFACE

In fact, the interface of the floor aggregation is identical with that of a single floor unit. However, the domains of the channels are extended by the identifier of the floor that is the source or destination.

`port` *ext_request*, *ext_enter*, *ext_served* `domain` $FID \times MoveDir$

BEHAVIOUR

The behaviour of the overall floor aggregation is simply the interleaving of the controllers of the middle floors, the bottom and the top floor, because they evolve independently of each other.

$$
\begin{aligned}
Behaviour \mathrel{\widehat{=}} & \left(\mathop{|||}_{id \in MFloors} mfloors.Behaviour(id) \right) \\
& ||| \ tfloor.Behaviour(floor_num) \ ||| \ bfloor.Behaviour(0)
\end{aligned}
$$

SPEC. UNIT *ExternalRequestHandler*

The current unit is responsible for managing the external floor requests in a first-in-first-out manner.

INTERFACE

Incoming external requests are transmitted by the floor panels along the channel *ext_enter*. There are two possibilities to assign an external request to a lift. First, when a lift becomes idle, it instructs the controller to provide a new external request via the channel *next_ext_req*, and the controller assigns the first pending external request in its queue to the lift via the channel *next_ext_resp*. Second, whenever a lift schedules a new internal request, it asks the external request handler for an appropriate external request between the current floor and the target floor in the current move direction. This external request needs not be the first in the queue, and it is transmitted along the channel *interm_ext_succ* in the positive case.

port ext_enter domain $FID \times MoveDir$;
port $next_ext_req$ domain LID;
port $next_ext_resp$ domain $LID \times FID \times MoveDir$;
port $interm_ext_req$ domain $LID \times FID \times FID$;
port $interm_ext_succ$ domain $LID \times FID$;
port $interm_ext_fail$ domain LID

The above partitioning of the interface into ports carrying requests from other units and ports carrying responses to these requests to the corresponding units is a general pattern. Since the channels carrying operation-related events are internal in RT-Z, we cannot model a 'request–computation–response' sequence by a single event; we must split it into at least three events.

STATE

The data state of the external request handler consists of the queue of external requests *queue* that are currently pending. The remaining state components are derived ones.

State

$queue : \mathrm{seq}(FID \times MoveDir)$
$up_reqs, down_reqs : FID \times FID \to \mathbb{F}\, FID$

$\forall f1, f2 : FID \mid f1 \leq f2 \bullet$
$\quad up_reqs(f1, f2) = \{f : FID \mid f \mapsto Up \in \mathsf{ran}\, queue \land f1 < f < f2\}$
$\quad down_reqs(f1, f2) = \{f : FID \mid f \mapsto Down \in \mathsf{ran}\, queue \land f1 < f < f2\}$

Initially, the queue is empty.

$Init == [\, State \mid queue = \langle\rangle \,]$

OPS & PREDS **[** $Join, FindSucc, FindFail, Dispatch, Empty, NonEmpty$ **]**

The operation *Join* adds a new external request to the current queue.

$$Join == [\,\Delta State;\ req? : FID \times MoveDir \mid queue' = queue \frown \langle req? \rangle\,]$$

The aim of the next two operations is to determine a potential pending external request in the queue that is situated between the current and the target floor in the current move direction of a particular lift. They represent the successful case of finding an appropriate external request and the corresponding failure case. Note that the preconditions of the two operations partition the data state space.

FindSucc

$$\Delta State$$
$$curr_fl?, curr_dest?, new_dest! : FID$$

$$curr_fl? \neq curr_dest?$$
$$(\exists_1\ md == \mathbf{if}\ curr_fl? \leq curr_dest?\ \mathbf{then}\ Up\ \mathbf{else}\ Down \bullet$$
$$(md = Up \land up_reqs(curr_fl?, curr_dest?) \neq \varnothing \land$$
$$new_dest! = \min up_reqs(curr_fl?, curr_dest?)$$
$$\lor$$
$$md = Down \land down_reqs(curr_dest?, curr_fl?) \neq \varnothing \land$$
$$new_dest! = \max down_reqs(curr_dest?, curr_fl?)) \land$$
$$queue' = squash(queue \rhd \{(new_dest!, md)\}))$$

FindFail

$$\Xi State$$
$$curr_fl?, curr_dest? : FID$$

$$(\exists_1\ md == \mathbf{if}\ curr_fl? \leq curr_dest?\ \mathbf{then}\ Up\ \mathbf{else}\ Down \bullet$$
$$md = Up \land up_reqs(curr_fl?, curr_dest?) = \varnothing$$
$$\lor$$
$$md = Down \land down_reqs(curr_dest?, curr_fl?) = \varnothing)$$

The operation *Dispatch* removes the first external request from the queue.

Dispatch

$$\Delta State$$
$$req! : FID \times MoveDir$$

$$queue \neq \langle\rangle$$
$$queue' = tail\ queue \land req! = head\ queue$$

The schemas *Empty* and *NonEmpty* define predicates on the data state to which the specification of the dynamic behaviour refers.

$$Empty == [\,\Xi State \mid queue = \langle\rangle\,]$$
$$NonEmpty == [\,\Xi State \mid queue \neq \langle\rangle\,]$$

BEHAVIOUR

The external request handler has to accomplish three main tasks. It has to insert new external requests into its queue (process *Join*), it has to provide the first pending request to an idle lift (process *Dispatch*) and it has to check whether there are appropriate pending external requests on the itinerary of a lift (process *Check*).

The stimulus–response behaviour of the external request handler depends on its current data state. When its queue is empty, it is able only to accept new external requests. Otherwise it provides each of its three tasks to the environment. This dependence on the state is modelled by the external choice whose branches are guarded by the state predicates *Empty* and *NonEmpty*.

$$Behaviour \mathrel{\widehat=} ERH$$

$$\begin{aligned} ERH \mathrel{\widehat=}\ & Empty_X \rightarrow Join;\ ERH \\ & \Box\ NonEmpty_X \rightarrow (Join \Box Dispatch \Box Check);\ ERH \end{aligned}$$

$$Join \mathrel{\widehat=} ext_enter?req : FID \times MoveDir \rightarrow Join_X!\langle\!|\ req? == req\ |\!\rangle \rightarrow Skip$$

$$\begin{aligned} Dispatch \mathrel{\widehat=}\ & next_ext_req?lift : LID \rightarrow Dispatch_X?par : [\,req! : FID \times MoveDir\,] \rightarrow \\ & next_ext_resp!(lift, par.req!) \rightarrow Skip \end{aligned}$$

$$\begin{aligned} Check \mathrel{\widehat=}\ & interm_ext_req?req : LID \times FID \times FID \rightarrow \\ & (FindSucc_X?par : [\,curr_fl?, curr_dest?, new_dest! : FID\ | \\ & \qquad curr_fl? = req.1 \land curr_dest? = req.2\,] \rightarrow \\ & \qquad interm_ext_succ!(req.0, par.new_dest!) \rightarrow Skip \\ & \Box\ FindFail_X!\langle\!|\ curr_fl? == req.1, curr_dest? == req.2\ |\!\rangle \rightarrow \\ & \qquad interm_ext_fail!(req.0) \rightarrow Skip) \end{aligned}$$

Let us illustrate the relationships between the Z operation schemas and the CSP events by means of the process *Check*. The process first awaits a request from a lift on the channel *interm_ext_req*. After such a request has been entered, the handler has to check whether it has an appropriate external request in its queue. This is the task of the operations *FindSucc* and *FindFail* whose execution events (with the postfix $_X$) guard both branches of the external choice operator $\Box$. In each data state of the handler, the precondition of exactly one operation is satisfied, depending on whether an appropriate request is available. The corresponding operation is executed which resolves the external choice.

Consider now the values that are communicated with the execution events of the operations.

On the one hand, the operation *FindFail* has only input parameters whose values are solely fixed by the CSP process. Therefore, the value denotation of the CSP execution event is a single binding that defines the mapping of input parameters to the values that have been transmitted with the external event before.

On the other hand, the operation *FindSucc* has input and output parameters. The values of the input parameters are to be fixed by the invoking CSP process, whereas the values of the output parameters are to be fixed by the invoked Z operation. Accordingly, a set of values can be communicated with the execution event. The event denotation represents a set of bindings mapping values to the input and output parameters in a way that the values of the input parameters are uniquely determined and that the values of the output parameters are arbitrary (constrained only by the operation schema). The set of bindings is denoted by a schema expression.

SPEC. UNIT *LiftCtrls*

The controllers of the lifts of the considered building are aggregated by the current specification unit.

INTERFACE

port $int_request$ domain $LID \times FID$;
port ext_served domain $FID \times MoveDir$;
port $next_ext_req$ domain LID;
port $next_ext_resp$ domain $LID \times FID \times MoveDir$;
port $interm_ext_req$ domain $LID \times FID \times FID$;
port $interm_ext_succ$ domain $LID \times FID$;
port $interm_ext_fail$ domain LID;
port $door_status$ domain $LID \times DoorMsg$;
port $door_cmd$ domain $LID \times DoorCmd$;
port $lift_move, lift_arrived$ domain $LID \times FID$

SUBUNITS

subunit $lifts(LID)$ spec. unit *LiftCtrl*

TYPES & CONSTANTS

The set *REN* defines a mapping of the values transmitted along the channel *ext_served*. It is referred to in the BEHAVIOUR section.

$REN == \{l : LID;\ f : FID;\ md : MoveDir \bullet ext_served.(l, f, md) \mapsto ext_served.(f, md)\}$

BEHAVIOUR

The lift controllers are composed by means of indexed aggregation using the interleaving operator, which allows the different lift controllers to evolve independently of each other.

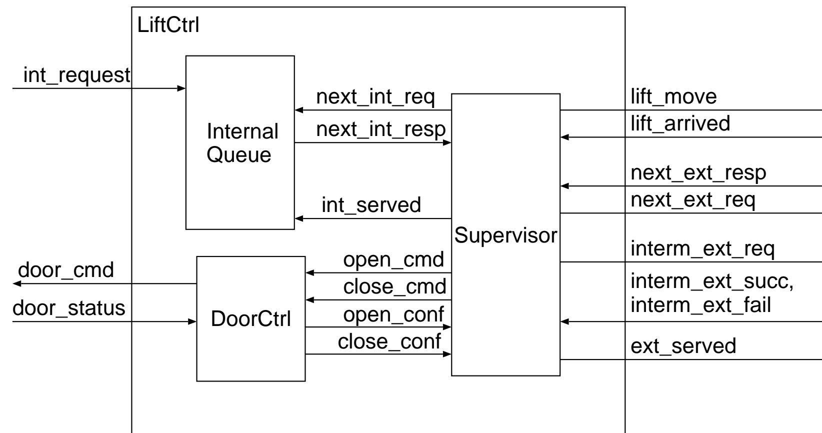


Figure 12.3: Lift controller architecture.

Technically speaking, the indexed aggregation has the consequence that the interaction of individual lift controllers along particular channels is extended by the respective lift identifier. That is, all lift controllers communicate via the identical channels, but synchronise on disjoint sets of transmitted values.

$$Behaviour \mathrel{\widehat{=}} REN\left(\mathop{|||}_{id \in LID} lifts.Behaviour(id)\right)$$

However, this extension of the domain is not appropriate for the external channel *ext_served*; because the floor controller that has made the request, whose completion is to be reported, is not interested in the particular lift that has accomplished the request. Therefore the renaming function *REN* is applied, which omits the lift identifier from each communication on the channel *ext_served*.

The architecture of a lift controller is illustrated in Figure 12.3. Each lift controller consists of a component *DoorCtrl* controlling the opening and closing process of the lift door, a component *InternalQueue* responsible for managing the set of current internal requests and a component *Supervisor* responsible for supervising the other components. Since the ports of the unit *LiftCtrl* have already been described, we outline only the internal channels.

The supervisor instructs the internal queue to provide the next internal request via the channel *next_int_req*, and the internal queue sends the respective internal request, if any, via the channel *next_int_resp*. The supervisor informs the internal queue about the successful completion of an internal request via the channel *int_served*.

The door controller is instructed to open and close via the channels *open_cmd* and *close_cmd*, and it reports the successful completion of the opening and closing process via the channels *open_conf* and *close_conf*, respectively.

SPEC. UNIT *LiftCtrl*

SUBUNITS

A lift controller is the aggregation of a supervisor component, a door controller and an internal queue of the activated panel buttons.

subunit *queue* spec. unit *InternalQueue*;
subunit *supervisor* spec. unit *Supervisor*;
subunit *door* spec. unit *DoorCtrl*

The ports and internal channels have already been discussed.

INTERFACE

The domains of the ports are obtained by omitting the set *LID* from the domains of the corresponding ports of the specification unit *LiftCtrls*, which composes the current unit by indexed aggregation.

port *int_request* domain *FID*;
port *ext_served* domain $FID \times MoveDir$;
port *next_ext_req* domain *SYNC*;
port *next_ext_resp* domain $FID \times MoveDir$;
port *interm_ext_req* domain $FID \times FID$;
port *interm_ext_succ* domain *FID*;
port *interm_ext_fail* domain *SYNC*;
port *door_status* domain *DoorMsg*;
port *door_cmd* domain *DoorCmd*;
port *lift_move*, *lift_arrived* domain *FID*

LOCAL

channel *open_cmd*, *close_cmd*, *open_conf*, *close_conf* domain *SYNC*;
channel *next_int_req*, *next_int_resp* domain $FID \times MoveDir$;
channel *int_served* domain *FID*

BEHAVIOUR

The following abbreviation definition introduces *ChannelSet* to denote the set of events that can be communicated along particular channels. It is referred to in the following behaviour definition.

$$ChannelSet \mathrel{\widehat{=}} \{open_cmd, close_cmd, open_conf, close_conf\} \cup \{\!|\ next_int_req\ |\!\} \cup \{\!|\ next_int_resp\ |\!\} \cup \{\!|\ int_served\ |\!\}$$

The components of a lift controller are composed in parallel. The door controller and the internal queue evolve completely independently, and they synchronise with the supervisor through the events in *ChannelSet*.

$Behaviour \mathrel{\widehat=}$
$\quad supervisor.Behaviour$
$|[\, ChannelSet \,]|$
$\quad (door.Behaviour \mid\mid\mid queue.Behaviour)$

SPEC. UNIT *DoorCtrl*

INTERFACE

The door controller is instructed by the supervisor component via the channels *open_cmd* and *close_cmd*, it confirms the completion of these commands via the channels *open_conf* and *close_conf*, it instructs the door motor via the channel *door_cmd* and it asks the sensor for the current state of the physical door via the channel *door_status*.

`port` *open_cmd*, *close_cmd*, *open_conf*, *close_conf* `domain` *SYNC*;
`port` *door_status* `domain` *DoorMsg*;
`port` *door_cmd* `domain` *DoorCmd*

BEHAVIOUR

The behaviour of a lift door (and consequently of its controller) is simple. Whenever a lift enters a target floor, the door goes through a cycle. The door is first opened, then remains open for at least t_{wait} time units and is subsequently closed. The process of closing the door may be interrupted, which is indicated by the door sensor. In this case, the door must be re-opened before the next attempt to close the door can take place.

$Behaviour \mathrel{\widehat=} OPEN$

$OPEN \mathrel{\widehat=} Wait\ t_{wait};\ close_cmd \rightarrow CLOSING$

$CLOSING \mathrel{\widehat=} door_cmd.CloseCmd \rightarrow$
$\quad (door_status.Closed \rightarrow close_conf \rightarrow CLOSED$
$\quad \triangle\ door_status.Blocked \rightarrow BLOCKED)$

$BLOCKED \mathrel{\widehat=} door_cmd.OpenCmd \rightarrow door_status.Opened \rightarrow Wait\ t_{wait};\ CLOSING$

$CLOSED \mathrel{\widehat=} open_cmd \rightarrow OPENING$

$OPENING \mathrel{\widehat=} door_cmd.OpenCmd \rightarrow door_status.Opened \rightarrow open_conf \rightarrow OPEN$

SPEC. UNIT *InternalQueue*

The internal queue of a lift is responsible for managing the pending internal requests. There is no order on these internal requests.

INTERFACE

New internal requests for a lift enter via the channel *int_request*. The supervisor component instructs the internal queue component via the channel *next_int_req* to provide the next internal request on the lift's itinerary, and this next internal request is delivered on the channel *next_int_resp*. When an internal request is completed, it is reported on the channel *int_served*.

port $next_int_req, next_int_resp$ domain $FID \times MoveDir$;
port $int_request, int_served$ domain FID

STATE

The pending (unordered) internal requests are modelled by the set *internal_reqs*. The remaining state components are derived ones used to specify the following operations more conveniently. Initially, there is no pending internal request.

State

$internal_reqs : \mathbb{F}\, FID$
$up_reqs, down_reqs : FID \rightarrow \mathbb{F}\, FID$

$up_reqs = \lambda fl : FID \bullet \{n : internal_reqs \mid n \geq fl\}$
$down_reqs = \lambda fl : FID \bullet \{n : internal_reqs \mid n \leq fl\}$

Init

$State$

$internal_reqs = \varnothing$

OPS & PREDS [$Join, Dispatch, Next, Empty, NonEmpty$]

$Join == [\, \Delta State;\ req? : FID \mid internal_reqs' = internal_reqs \cup \{req?\}\,]$
$Dispatch == [\, \Delta State;\ req? : FID \mid internal_reqs' = internal_reqs \setminus \{req?\}\,]$

The operation *Next* determines, for a given current floor and a current direction of movement, the next pending internal request in the itinerary. The operation is divided into two cases for each of the two move directions. The chosen requested floor is the closest floor in the current move direction, where the move direction changes at the top and bottom floor.

NextUp

$\Xi State$
$fl?, dest! : FID$
$md?, md! : MoveDir$

$internal_reqs \neq \varnothing$
$md? = Up$
$(up_reqs(fl?) \neq \varnothing \wedge$
$dest! = \min up_reqs(fl?) \wedge md! = Up)$
$\vee$
$(up_reqs(fl?) = \varnothing \wedge down_reqs(fl?) \neq \varnothing \wedge$
$dest! = \max down_reqs(fl?) \wedge md! = Down)$

NextDown

$\Xi State$
$fl?, dest! : FID$
$md?, md! : MoveDir$

$internal_reqs \neq \varnothing$
$md? = Down$
$(down_reqs(fl?) \neq \varnothing \wedge$
$dest! = \max down_reqs(fl?) \wedge md! = Down)$
$\vee$
$(down_reqs(fl?) = \varnothing \wedge up_reqs(fl?) \neq \varnothing \wedge$
$dest! = \min up_reqs(fl?) \wedge md! = Up)$

$Next == NextUp \lor NextDown$

The following predicates are needed to express the dependence of the dynamic behaviour on the current data state.

$NonEmpty == [\,\Xi State \mid internal_reqs \neq \varnothing\,]$
$Empty == [\,\Xi State \mid internal_reqs = \varnothing\,]$

BEHAVIOUR

The behaviour of the queue controller depends on the current data state. If there is no pending internal request, the internal queue is not able to respond to a request to deliver the next floor.

$Behaviour \mathrel{\widehat=} IQ$

$IQ \mathrel{\widehat=} (NonEmpty_X \rightarrow (Request \,\Box\, Serviced \,\Box\, Select);\; IQ$
$\quad \Box\; Empty_X \rightarrow (Request \,\Box\, Serviced);\; IQ)$

The queue controller must accomplish three main tasks. First, it has to react to new internal requests by activating the corresponding button (process *Request*). Second, it has to react to the completion of an internal request by deactivating the corresponding button (process *Serviced*). Finally, it has to provide the lift controller with the next pending internal request when being asked (process *Select*).

$Request \mathrel{\widehat=} int_request?fl : FID \rightarrow Join_X!\langle\!|\; req? == fl \;|\!\rangle \rightarrow Skip$

$Serviced \mathrel{\widehat=} int_served?fl : FID \rightarrow Dispatch_X!\langle\!|\; req? == fl \;|\!\rangle \rightarrow Skip$

$Select \mathrel{\widehat=} next_int_req?req : FID \times MoveDir \rightarrow$
$\quad Next_X?par : [fl?, dest! : FID;\; md?, md! : MoveDir \mid fl? = req.0 \land md? = req.1\,] \rightarrow$
$\quad next_int_resp!(par.dest!, par.md!) \rightarrow Skip$

SPEC. UNIT *Supervisor*

The current unit supervises the other components of the lift controller.

INTERFACE

port *lift_move*, *lift_arrived* domain *FID*;
port *open_cmd*, *close_cmd*, *open_conf*, *close_conf* domain *SYNC*;
port *next_int_req*, *next_int_resp* domain $FID \times MoveDir$;
port *int_served* domain *FID*;
port *next_ext_req* domain *SYNC*;
port *next_ext_resp*, *ext_served* domain $FID \times MoveDir$;
port *interm_ext_req* domain $FID \times FID$;
port *interm_ext_succ* domain *FID*;
port *interm_ext_fail* domain *SYNC*

STATE

The data state consists of the current floor and move direction.

$State == [floor : FID;\ mdir : MoveDir]$
$Init == [State \mid floor = 0 \land mdir = Up]$

BEHAVIOUR

The behaviour of the supervisor component is cyclic, expressed by a recursive equation. In each cycle, it serves an internal or external request, defined by the processes *InternalReq* and *ExternalReq*, respectively. Internal requests have priority over external requests. This is implemented with the help of the timeout operator ($\triangleright$): during the first $t_{priority}$ time units after the last request has been accomplished, only internal requests are accepted; afterwards internal and external requests are accepted equally, represented by the external choice.

$$
\begin{aligned}
& Behaviour \mathrel{\widehat=} SV \\
& SV \mathrel{\widehat=} \sigma(\\
& \quad InternalReq(floor, mdir) \\
& \quad \triangleright\{t_{priority}\} \\
& \qquad (InternalReq(floor, mdir) \mathrel{\Box} ExternalReq));\ SV
\end{aligned}
$$

Note the use of the σ operator, which we have introduced in Section 7.1 as an abbreviation. To obtain the next internal request on the itinerary, the supervisor needs to access its current data state, incorporating the current floor (*floor*) and the current move direction (*mdir*). As explained in Section 7.1, a dedicated operation *Lookup* is defined for each specification unit, which provides the access to the current data state. The σ-notation is a shorthand for the execution of this operation, represented by the event

$$Lookup_X?st : [floor! : FID;\ mdir! : MoveDir]$$

prefixing the process that is obtained from the process contained in the brackets of the σ-operator by substituting $st.floor!$ for *floor* and $st.mdir!$ for *mdir*.

Whenever the lift controller schedules a new internal request it first checks whether there is an external request between the current and the scheduled floor in the respective move direction. If this is the case, the external request is first served; otherwise the internal request is served.

$$
\begin{aligned}
& InternalReq(fl, md) \mathrel{\widehat=} next_int_req!(fl, md) \rightarrow next_int_resp?dest : FID \times MoveDir \rightarrow \\
& \quad interm_ext_req!(fl, dest.0) \rightarrow \\
& \qquad (interm_ext_fail \rightarrow HandleInternal(dest.0, dest.1) \\
& \qquad \Box\ interm_ext_succ?newdest : FID \rightarrow HandleExternal(newdest, md))
\end{aligned}
$$

$$
\begin{aligned}
& ExternalReq \mathrel{\widehat=} next_ext_req \rightarrow next_ext_resp?dest : FID \times MoveDir \rightarrow \\
& \quad HandleExternal(dest.0, dest.1)
\end{aligned}
$$

$$HandleInternal(fl, md) \mathrel{\widehat=} Move(fl, md);\ int_served!fl \rightarrow Skip$$

$$HandleExternal(fl, md) \mathrel{\widehat=} Move(fl, md);\ ext_served!(fl, md) \rightarrow Skip$$

$Move(fl, md) \mathrel{\widehat=} close_cmd \rightarrow close_conf \rightarrow lift_move!fl \rightarrow lift_arrived!fl \rightarrow$
$\quad open_cmd \rightarrow open_conf \rightarrow Update_{floor_mdir}(fl, md);\ Skip$

The notation $Update_{floor_md}(fl, md)$ is the counterpart of the σ-operator for updating the current data state, also discussed in Section 7.1. It is a shorthand for the process that executes the dedicated operation $Update_{floor_mdir}$ and associates the operation parameters with the provided values.

12.1.4 Discussion

The case study presented in this section demonstrates the ability of RT-Z to formally specify a moderately complex, inherently concurrent and distributed system.

RT-Z has succeeded in modelling all relevant properties of the multi-lift controller, including real-time constraints. It has coped with the complexity of the controller by applying its decomposition concepts, i.e., simple and indexed aggregation and extension. The broad spectrum of properties of the multi-lift controller has been addressed by the combined expressive power of Z and timed CSP as provided by RT-Z. Last but not least, we have covered the (software) requirements specification and design phases, which involve different levels of abstraction, by using abstract and concrete specification units.

However, we have not demonstrated that the design specification is a refinement of the requirements specification, i.e., that it correctly implements the specified requirements. As discussed in Chapters 10 and 13, the verification techniques needed to bridge the gap between abstract and concrete specification units are part of future work.

A more compact version of the multi-lift specification can also be found in [Sühl, 2002].

Let us now discuss the differences between our case study and the TCOZ case study of the multi-lift controller presented by Mahony and Dong [1998a].

The case study has revealed that the concept of object instantiation adopted in TCOZ is more flexible than the concept of aggregation provided by RT-Z. Class objects in TCOZ can be composed by the instantiating object by using arbitrary Z structures, e.g., sets and sequences. In comparison, indexed aggregation is more restricted. However, the syntactical power of object instantiation in TCOZ is impaired by its semantical undefinedness, as discussed in detail in Chapter 11.

Disparate integration principles underlie RT-Z and TCOZ. In some cases, the RT-Z model, which has no direct counterparts of state guards and of direct references from process expressions to state variables, gives rise to a larger and more complex specification than the TCOZ model. An example of this is the need in the RT-Z specification to split the original channel *select* of the TCOZ specification into the channels *next_ext_req* and *next_ext_resp* for the discussed modelling reasons. On the one hand, this channel duplication makes the specification less succinct, always involving a pair of events rather than a single event. On the other hand, modelling this communication relationship between the lift controllers and the external request handler by means of two channels is closer to reality. No implementation of the multi-lift specification is able to perform the state computation of the external request

handler instantaneously; therefore, the events of demanding and providing an appropriate external request have to be separated in time, i.e., modelled by two different events. To conclude, RT-Z forces the specifier to adopt a more concrete modelling style that is—concerning the intervals of performing operations—closer to the real-time behaviour of the resulting implementation.

A large part of the RT-Z specification is covered by the declaration of ports and channels. In particular, the top-level specification units, which aggregate a number of other specification units, have rather long declaration lists. The `INTERFACE` and `LOCAL` sections of compound specification units are redundant, because they can be uniquely determined given the external ports and internal channels of the units being composed. In spite of this redundancy, we have decided to make the interface of compound specification units explicit. This is a contrast to TCOZ specifications, in which the interface of a compound object is not defined explicitly. While this makes the TCOZ specification of the multi-lift controller more compact, it is very hard to retain the overview: looking at the definition of a compound class, we have no idea about its interface.

RT-Z's ability to specify environmental assumptions enabled us to make a clearer separation between the system to be implemented and its environment than in the TCOZ specification. In the latter, the shaft component of a lift is part of the modelled system, because it is the only possibility to express assumptions about its real-time behaviour. The RT-Z specification, on the other hand, stipulated that the shaft component is outside the system to be implemented by not incorporating a shaft specification unit but only recording the ports in the `INTERFACE` section that link the multi-lift controller to the shaft component. The `ENVIRONMENTAL ASSUMPTIONS` section allowed us to express the assumptions about the real-time behaviour formally that must be met for the multi-lift controller to operate reasonably.

Last, but not least, another difference between the RT-Z and the TCOZ specification, which is not really caused by the different underlying models, is the order of presenting the specification units and classes, respectively. The TCOZ specification follows a strict bottom–up order which is consistent with the 'definition-before-use' paradigm. In contrast, we have deviated from this paradigm in the RT-Z specification, because in our opinion a system is explained preferably by providing the coarse system structures (i.e., the top-level units) first before giving the details (i.e., the bottom-level units).

12.2 Gas Burner

To demonstrate the applicability of RT-Z to safety-critical systems, we discuss a case study involving a gas burner, whose malfunctioning may cause accidents. The considered system in terms of which safety constraints are to be formulated is a plant within which a gas burner is to be operated.

Our case study was inspired by the case study presented by Ravn et al. [1993], which served the authors to illustrate the duration calculus.

12.2.1 System Requirements

There are two safety constraints on the operation of the gas burner in the plant context.

1. The volume of unburned gas within the plant escaped from the gas burner must never exceed a volume of *max_gas_volume* units.
2. Depending on specific objects in the surroundings of the gas burner within the plant, the heat produced by the (activated or deactivated) gas burner must not exceed a particular temperature.

It is important to stress that the system that we specify by the following specification unit is *not* the artifact we want to implement (gas burner). Rather, it is the system whose safety is affected by the operation of the artifact to be implemented. That is, the general approach taken is to take into account parts of the environment of the artifact to be implemented.

The following data types and constants are global with respect to the whole system specification. The type definitions embody several abstractions we have carried out in modelling the plant system. Temperature values, distances between objects and volumes are modelled by natural numbers rather than by real numbers. Further, by introducing *Object* as a given set, we express that we are not interested in the characteristics of the objects in the surroundings of the gas burner.

$[Object]$
$Temp == \mathbb{N};\ Dist == \mathbb{N};\ Vol == \mathbb{N}$

The following global variables represent static properties of the plant system, i.e., they are parameters of the system specification.

$$\begin{array}{|l}
max_gas_volume, min_gas_escape_plant : Vol \\
max_burner_temp : Temp \\
max_temp_dist : Object \rightarrow\!\!\!\!\!\!+ (Dist \rightarrow Temp) \\
\Delta : \mathbb{T} \\
\hline
\forall o : Object;\ d1, d2 : Dist \mid o \in \mathrm{dom}\, max_temp_dist \wedge d1 \leq d2 \bullet \\
\qquad max_temp_dist\ o\ d1 \leq max_temp_dist\ o\ d2
\end{array}$$

In each time interval of length Δ, *min_gas_escape_plant* volume units of gas are supposed to escape from the plant. The function *max_temp_dist* assigns to each object a function mapping

each distance of that object to the gas burner to the temperature that it can maximally sustain. The constant *max_burner_temp* denotes the maximal temperature the gas burner can produce.

The first specification unit encapsulates the description of the interface between the plant system and its environment. The encapsulation allows us to refer to these definitions several times.

SPEC. UNIT *PlantInterface*

INTERFACE

Channels in a system specification do not necessarily carry the transmission of information units, but they may carry the "flow of material units." The escape of a particular volume of gas from the plant to the environment is modelled by events on the channel *gas_escape_plant*. Note that the model underlying RT-Z does not allow us to model a continuous escape of gas. The gas that escapes from the plant within an interval has to be accumulated in a single event on the channel *gas_escape_plant*. Events occurring on the channel *object_enter* represent the entry of new objects into the plant, with a specific distance to the gas burner.

port *gas_escape_plant* domain *Vol*;
port *object_enter* domain $Object \times Dist$

The specification unit *Plant* is defined to extend the previous one.

SPEC. UNIT *Plant* **EXTENDS** *PlantInterface*

ENVIRONMENTAL ASSUMPTIONS

Analogously to environmental assumptions of software components, which concern properties of information units, environmental assumptions of system components express properties required of the material flows between system components.

$EA \mathrel{\widehat{=}} EA1 \land EA2$

Regarding the plant system we assume that at least *min_gas_escape_plant* volume units of gas may escape from the plant in each interval of Δ time units.

$$\begin{aligned} EA1 \mathrel{\widehat{=}} & \forall t : \mathbb{T} \mid t \geqslant_{\mathbb{R}} \Delta \bullet \\ & \quad \Sigma(\mathrm{strip}(s \uparrow_{ttr} [t -_{\mathbb{R}} \Delta, t) \upharpoonright_{ttr} \{\!|\, gas_escape_plant \,|\!\})) \geq min_gas_escape_plant \\ & \quad \lor (\exists\, vol : Vol \mid \Sigma(\mathrm{strip}(s \uparrow_{ttr} [t -_{\mathbb{R}} \Delta, t) \upharpoonright_{ttr} \{\!|\, gas_escape_plant \,|\!\})) + vol \\ & \qquad\qquad \geq min_gas_escape_plant \bullet \\ & \qquad gas_escape_plant.vol \ \mathsf{open}\ t + \Delta) \end{aligned}$$

Moreover, objects entering the plant must not be as close to the gas burner as to violate the second safety constraint immediately.

$EA2 \mathrel{\widehat=} \forall\, t : \mathbb{T};\ o : Object;\ d : Dist \bullet$
$\quad object_enter.(o, d)\ \mathsf{open}\ t \Rightarrow (max_temp_dist\ o\ d) \geq max_burner_temp$

STATE PROPERTIES

The following state schema models the real state of the plant system. That is, each component of the state schema represents a characteristic of the plant system that is really present (observable).

State

$gas_volume : Vol$
$burner_temp : Temp$
$object_dist : Object \rightarrowtail\!\!\!\!\!\!\!\rightarrow Dist$

$gas_volume \leq max_gas_volume$
$burner_temp \leq max_burner_temp$
$\mathsf{dom}\, object_dist \subseteq \mathsf{dom}\, max_temp_dist$
$\forall\, o : \mathsf{dom}\, object_dist \bullet burner_temp \leq max_temp_dist\ o\ (object_dist\ o)$

The predicate of the state schema mainly expresses the safety constraints on the operation of the gas burner (first and last conjunct). The state schema is the natural place to formulate safety constraints, because—by definition—safety is a property referring to the state of a system, for a definition of safety consult [Leveson, 1995, pp. 181–183].

In conclusion, the above specification of the system requirements has indicated that RT-Z is flexible enough to deal with general system components in addition to software components. This is mainly due to the capabilities of the base language Z whose underlying concepts (set theory and predicate logic) are general enough to enable the formulation of system and of software properties.

We have also seen, however, that RT-Z is not able to model continuous behaviour. If the correctness of a system can be expressed only in terms of continuous functions and differential equations, i.e., if the abstraction carried out with respect to the gas escape is not adequate, RT-Z is not the appropriate choice. In this case, formalisms such as the duration calculus [Chaochen et al., 1991] must be used.

12.2.2 System Design

To guarantee the safety constraints fixed in the system requirements phase, the plant system is decomposed into the components depicted in Figure 12.4, which are associated with specific responsibilities. The gas burner is equipped with a controller that monitors a thermometer (sensor) and instructs a gas valve and an ignition source (actuators). The whole gas burner is supervised by a human operator. The gas burner controller is responsible for ensuring the first safety constraint, and the human operator is responsible for the second safety constraint.

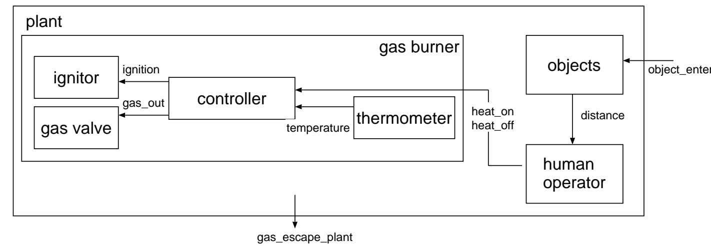


Figure 12.4: Plant design.

The volume of unburned gas in the plant escaped from the gas burner must not exceed *max_gas_volume* units. According to the environmental assumptions, at least *min_gas_escape_plant* volume units are guaranteed to escape from the plant in each time interval of length Δ. From the technical parameters of the gas burner one can derive how long the gas burner may maximally leak unburned gas and how long the plant needs to reduce the maximal amount of unburned gas to zero. Suppose that depending on these factors unburned gas is allowed to escape at most Δ_2 time units within any time interval of length Δ_1. It is decided in the system design that it is the task of the gas burner controller to guarantee that constraint and that this controller is to be implemented by software.

The second safety constraint concerns the burner temperature that is maximally allowed, which depends on the distance of objects to the gas burner. It is decided that it is primarily the responsibility of the human operator to deactivate the gas burner early enough such that the maximally admissible temperature is not exceeded for each approaching object. This, however, requires that a command of the human operator to deactivate the gas burner is executed by the controller timely. Two assumptions underlie the implementation of the second safety constraint. First, it is supposed that the temperature in the surroundings of the burner decreases with increasing distance according to a particular function. Second, it is assumed that once the gas burner is switched off the flame will disappear and the temperature produced by the gas burner will decrease to the environmental temperature according to a particular function.

The following specification unit aims to formally define the architecture of the plant system as devised in the system design process discussed above. We fix only the structuring into particular components and their interfaces here. We do not specify the requirements associated with these system components; this is the subject of the following software requirements phase.

SPEC. UNIT *PlantDesign* **EXTENDS** *PlantInterface*

SUBUNITS

subunit *gas_burner* spec. unit *GasBurner*;
subunit *human_operator* spec. unit ...
subunit *objects* spec. unit ...

LOCAL

channel $heat_on, heat_off$ domain $SYNC$;
channel $distance$ domain ...

BEHAVIOUR

$Behaviour \mathrel{\widehat=} (gas_burner.Behaviour \,|[\, \{heat_on, heat_off\} \,]|\, human_operator.Behaviour)$
$\quad |[\, \{\!| \; distance \; |\!\} \,]|\, objects.Behaviour$

In the remainder of this case study, we deal only with the gas burner, omitting the other system components.

SPEC. UNIT *GasBurner*

INTERFACE

port $heat_on, heat_off$ domain $SYNC$

SUBUNITS

subunit *controller* spec. unit *GasBurnerControl*;
subunit *gas_valve* spec. unit ...
subunit *ignitor* spec. unit ...
subunit *thermometer* spec. unit ...

LOCAL

channel $temperature$ domain $Temp$;
channel $gas_out, ignition$ domain $SYNC$

BEHAVIOUR

$Behaviour \mathrel{\widehat=} controller.Behaviour$
$\quad |[\, \{gas_out, ignition\} \cup \{\!| \; temperature \; |\!\} \,]|$
$\quad (gas_valve.Behaviour \; ||| \; ignitor.Behaviour \; ||| \; thermometer.Behaviour)$

The sensor and actuator components of the gas burner would be designed in hardware development processes. In the remainder of this case study, however, we address only the development of the software controller.

12.2.3 Software Requirements

The subject of the software requirements specification is the software controller identified in the previous system design. More specifically, we are dealing with the requirements the software controller must meet.

We first encapsulate the parameters and the interface of the software controller in separate specification units, which allows us to refer to them several times.

SPEC. UNIT *GBCConstants*

TYPES & CONSTANTS

The time constants Δ_1 and Δ_2 have already been introduced informally in the previous section, and Δ_3 models the length of the control cycles of the software component. The existence of a control cycle is already fixed in the software requirements specification, because it has consequences on the expression of the software requirements. The constant *limit* denotes the temperature above which the presence of a flame is assumed.

$\Delta_1, \Delta_2, \Delta_3 : \mathbb{T}$
$limit : Temp$

$0_{\mathbb{R}} <_{\mathbb{R}} \Delta_3 <_{\mathbb{R}} \Delta_2 <_{\mathbb{R}} \Delta_1$

SPEC. UNIT *GBCInterface* **EXTENDS** *GBCConstants*

INTERFACE

port *temperature* domain *Temp*;
port *heat_on*, *heat_off* domain *SYNC*;
port *ignition*, *gas_out* domain *SYNC*

ENVIRONMENTAL ASSUMPTIONS

We assume that the software controller has the full and immediate control of the ignition and the gas actuators. Moreover, the thermometer is supposed to deliver a unique temperature at any time instant.

$EA \mathrel{\widehat{=}} EA1 \wedge EA2$

$EA1 \mathrel{\widehat{=}} ignition \text{ active} \wedge gas_out \text{ active}$

$EA2 \mathrel{\widehat{=}} \forall t : \mathbb{T} \bullet \exists_1 meas : Temp \bullet temperature.meas \text{ open } t$

The responsibility to ensure the safety constraints associated with the gas burner controller

during the system design is expressed by the following specification unit.

SPEC. UNIT *SafeGasBurnerControl* **EXTENDS** *GBCInterface*

I/O RELATIONS

The following two Z schemas define the property of a temperature measurement to characterise either the absence or the presence of a flame. They are referred to in the BEHAVIOURAL PROPERTIES section.

$NotBurning == [\, meas : Temp \mid meas < limit \,];\ Burning == [\, meas : Temp \mid meas \geq limit \,]$

The subsequent specification of the safety constraints demonstrates one benefit of RT-Z, namely the coherent integration of the property languages of Z and timed CSP.

BEHAVIOURAL PROPERTIES

The following auxiliary predicates characterise the states in which gas has escaped in the current control cycle and where the last temperature measurement indicates the ignition of gas or no ignition, respectively. The two predicates are used in the following predicates defining the safety constraints.

$$UnignitedGas(t : \mathbb{T}) \mathrel{\widehat=} NotBurning(meas \mathrel{\widehat=} \mathrm{last}((s \restriction_{ttr} t) \restriction_{ttr} \{\!| \, temperature \, |\!\})) \land \\ \quad \exists\, t' : [t -_{\mathbb{R}} \Delta_3, t) \bullet gas_out \ \mathsf{at}\ t'$$

$$IgnitedGas(t : \mathbb{T}) \mathrel{\widehat=} Burning(meas \mathrel{\widehat=} \mathrm{last}((s \restriction_{ttr} t) \restriction_{ttr} \{\!| \, temperature \, |\!\})) \land \\ \quad \exists\, t' : [t -_{\mathbb{R}} \Delta_3, t) \bullet gas_out \ \mathsf{at}\ t'$$

$Behaviour$ **sat** $SC1 \land SC2 \land SC3$

Firstly, there must not be any time interval longer than Δ_2 time units in which gas constantly escapes and in which no flame is detected, i.e., in which gas constantly escapes unignited.

$$SC1 \mathrel{\widehat=} \forall\, t1, t2 : \mathbb{T} \mid t1 \leqslant_{\mathbb{R}} t2 \bullet \\ \quad (\forall\, t3 : [t1, t2) \bullet UnignitedGas(t3)) \Rightarrow t2 -_{\mathbb{R}} t1 \leqslant_{\mathbb{R}} \Delta_2$$

Secondly, two different attempts to ignite gas (two time instants at which gas escapes unignited) between which there is a time instant at which a flame is detected have to be separated in time by at least Δ_1 time units.

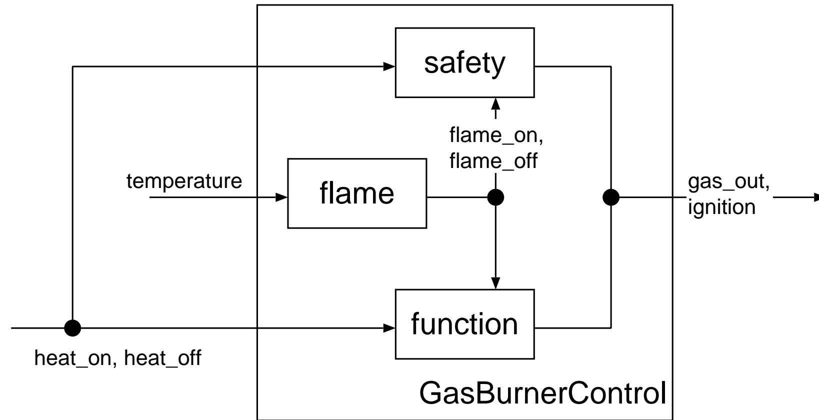


Figure 12.5: Gas burner controller architecture.

$$SC2 \mathrel{\widehat=} \forall\, t1, t2 : \mathbb{T} \mid t1 \leqslant_{\mathbb{R}} t2 \bullet$$
$$(UnignitedGas(t1) \land UnignitedGas(t2) \land \exists\, t3 : (t1, t2) \bullet IgnitedGas(t3))$$
$$\Rightarrow t2 -_{\mathbb{R}} t1 >_{\mathbb{R}} \Delta_1$$

Finally, gas may escape and the ignition source may be activated only when the last command was a *heat_on* command.

$$SC3 \mathrel{\widehat=} \forall\, t : \mathbb{T} \bullet$$
$$\mathrm{last}(s \restriction_{ttr} t \restriction_{ttr} \{heat_on, heat_off\}) = heat_off \Rightarrow (\neg\, gas_out \text{ live } t \land \neg\, ignition \text{ live } t)$$

12.2.4 Software Design

In general, RT-Z is designed to formally specify real-time embedded systems and is thus not specifically oriented towards safety-critical systems. However, it provides important means for coping with safety-critical applications. In the following software design specification, one means for enhancing the safety of the gas burner is the strict separation of functional and safety aspects in different components (defined by specification units). The parallel composition operator of timed CSP is adequate to compose the functional and safety components, because each communication with the environment (in particular commands to the actuators, which are subject to safety considerations) is possible only when the two components agree on that communication.

SPEC. UNIT *GasBurnerControl* **EXTENDS** *GBCInterface*

SUBUNITS

The current specification unit defines the decomposition of the software controller into three components (see also Figure 12.5): one for ensuring the safety constraints, one for accomplishing the function and one for detecting the transitions of the flame's state.

subunit *function* spec. unit *FuncComp*;
subunit *safety* spec. unit *SafetyComp*;
subunit *flame* spec. unit *FlameComp*

LOCAL

channel $flame_on, flame_off$ domain *SYNC*

BEHAVIOUR

The configuration of the three components is defined by the process term *Behaviour*. Most important, the safety component and the functional component are composed in parallel; and they synchronise on the commands to the actuators, which ensures that each command is given only if the components agree on that command.

$$
\begin{array}{l}
Behaviour \mathrel{\widehat=} \\
\quad flame.Behaviour \\
|[\,\{flame_on, flame_off\}\,]| \\
\qquad (function.Behaviour \\
\quad\; |[\,\{gas_out, ignition, flame_on, flame_off, heat_on, heat_off\}\,]| \\
\qquad safety.Behaviour)
\end{array}
$$

The safety component is responsible for guaranteeing that the outputs to the actuators are given according to the safety constraints. In other words, it blocks the output of commands that would violate these safety constraints. However, it does not take functional considerations into account, i.e., it does not define when it is necessary to send particular commands.

SPEC. UNIT *SafetyComp* **EXTENDS** *GBCConstants*

INTERFACE

port $flame_on, flame_off, heat_on, heat_off$ domain *SYNC*;
port $gas_out, ignition$ domain *SYNC*

LOCAL

channel $flm_on, flm_off, first_gas_out$ domain *SYNC*;
channel $enable_1, disable_1, enable_2, disable_2$ domain *SYNC*

BEHAVIOUR

The following process terms defining the behaviour of the safety component are best understood by considering the state machine depicted in Figure 12.6, which is a direct translation of these process terms.

$$\begin{aligned} &\mathit{Behaviour} \mathrel{\widehat{=}} \mathit{BufferOff} \mid[\, \{\mathit{flm_on}, \mathit{flm_off}\} \,]\mid \\ &\quad (\mathit{GasIgnition} \mid[\, \{\mathit{enable}_1, \mathit{enable}_2, \mathit{disable}_1, \mathit{disable}_2\} \,]\mid (\mathit{SC}1\&2 \mid\mid\mid \mathit{SC}3)) \end{aligned}$$

The first two safety constraints are modelled by the process term $SC1\&2$, and the last safety constraint is modelled by $SC3$. The process terms express the corresponding safety constraints in terms of the events $enable_1$, $enable_2$, $disable_1$ and $disable_2$. The two enable events indicate that the corresponding safety constraint allows the escape of gas and the ignition. The disable events have the analogous meaning.

The process term *GasIgn* models the safety component's acceptance of the escape of gas and of the ignition in terms of the enable and disable events. It is composed in parallel with the two process terms modelling the safety constraints, where all three processes synchronise on the enable and disable events. In brief, the escape of gas and the ignition is accepted if, and only if, the parallel processes have lastly sent an enable event.

The task of the process term *BufferOff*, which in turn is composed in parallel with the three process terms described above, is to buffer the external events *flame_on* and *flame_off*, which are passed on by means of the internal events *flm_on* and *flm_off*. This decoupling is necessary, because the safety component is always required to accept the two external events.

$$\begin{aligned} &\mathit{BufferOff} \mathrel{\widehat{=}} \mathit{flame_on} \rightarrow \mathit{BufferOn} \;\Box\; \mathit{flm_off} \rightarrow \mathit{BufferOff} \\ &\mathit{BufferOn} \mathrel{\widehat{=}} \mathit{flame_off} \rightarrow \mathit{BufferOff} \;\Box\; \mathit{flm_on} \rightarrow \mathit{BufferOn} \end{aligned}$$

$$\begin{aligned} &\mathit{GasIgnition} \mathrel{\widehat{=}} \mathit{GI}_{\mathit{Disabled}} \\ &\mathit{GI}_{\mathit{Disabled}} \mathrel{\widehat{=}} \mathit{enable}_1 \rightarrow \mathit{GI}_{\mathit{Disabled2}} \;\Box\; \mathit{enable}_2 \rightarrow \mathit{GI}_{\mathit{Disabled1}} \\ &\mathit{GI}_{\mathit{Disabled1}} \mathrel{\widehat{=}} \mathit{enable}_1 \rightarrow \mathit{GI}_{\mathit{Enabled1}} \;\Box\; \mathit{disable}_2 \rightarrow \mathit{GI}_{\mathit{Disabled}} \;\Box\; \mathit{ignition} \rightarrow \mathit{GI}_{\mathit{Disabled1}} \\ &\mathit{GI}_{\mathit{Disabled2}} \mathrel{\widehat{=}} \mathit{enable}_2 \rightarrow \mathit{GI}_{\mathit{Enabled1}} \;\Box\; \mathit{disable}_1 \rightarrow \mathit{GI}_{\mathit{Disabled}} \\ &\mathit{GI}_{\mathit{Enabled1}} \mathrel{\widehat{=}} \mathit{ignition} \rightarrow \mathit{GI}_{\mathit{Enabled1}} \;\Box\; \mathit{gas_out} \rightarrow \mathit{first_gas_out} \rightarrow \mathit{GI}_{\mathit{Enabled2}} \\ &\quad \Box\; \mathit{disable}_1 \rightarrow \mathit{GI}_{\mathit{Disabled1}} \;\Box\; \mathit{disable}_2 \rightarrow \mathit{GI}_{\mathit{Disabled2}} \\ &\mathit{GI}_{\mathit{Enabled2}} \mathrel{\widehat{=}} \mathit{ignition} \rightarrow \mathit{GI}_{\mathit{Enabled2}} \;\Box\; \mathit{gas_out} \rightarrow \mathit{GI}_{\mathit{Enabled2}} \\ &\quad \Box\; \mathit{disable}_1 \rightarrow \mathit{GI}_{\mathit{Disabled1}} \;\Box\; \mathit{disable}_2 \rightarrow \mathit{GI}_{\mathit{Disabled2}} \end{aligned}$$

$$\begin{aligned} &\mathit{SC}1\&2 \mathrel{\widehat{=}} \mathit{enable}_1 \rightarrow \mathit{SC}1\&2_{\mathit{ENABLED}} \\ &\mathit{SC}1\&2_{\mathit{ENABLED}} \mathrel{\widehat{=}} \mathit{first_gas_out} \rightarrow \mathit{SC}1\&2_{\mathit{IGNITION}} \\ &\mathit{SC}1\&2_{\mathit{IGNITION}} \mathrel{\widehat{=}} \mathit{flm_on} \rightarrow \mathit{SC}1\&2_{\mathit{BURNING}} \\ &\quad \triangleright\{\Delta_2\}\; \mathit{disable}_1 \rightarrow \mathit{SC}1\&2_{\mathit{DISABLED}} \\ &\mathit{SC}1\&2_{\mathit{BURNING}} \mathrel{\widehat{=}} (\mathit{flm_off} \rightarrow \mathit{disable}_1 \rightarrow \mathit{Skip} \mid\mid\mid \mathit{Wait}\;\Delta_1); \\ &\quad \mathit{enable}_1 \rightarrow \mathit{SC}1\&2_{\mathit{ENABLED}} \\ &\mathit{SC}1\&2_{\mathit{DISABLED}} \mathrel{\widehat{=}} \mathit{Wait}\;\Delta_1;\; \mathit{enable}_1 \rightarrow \mathit{SC}1\&2_{\mathit{ENABLED}} \end{aligned}$$

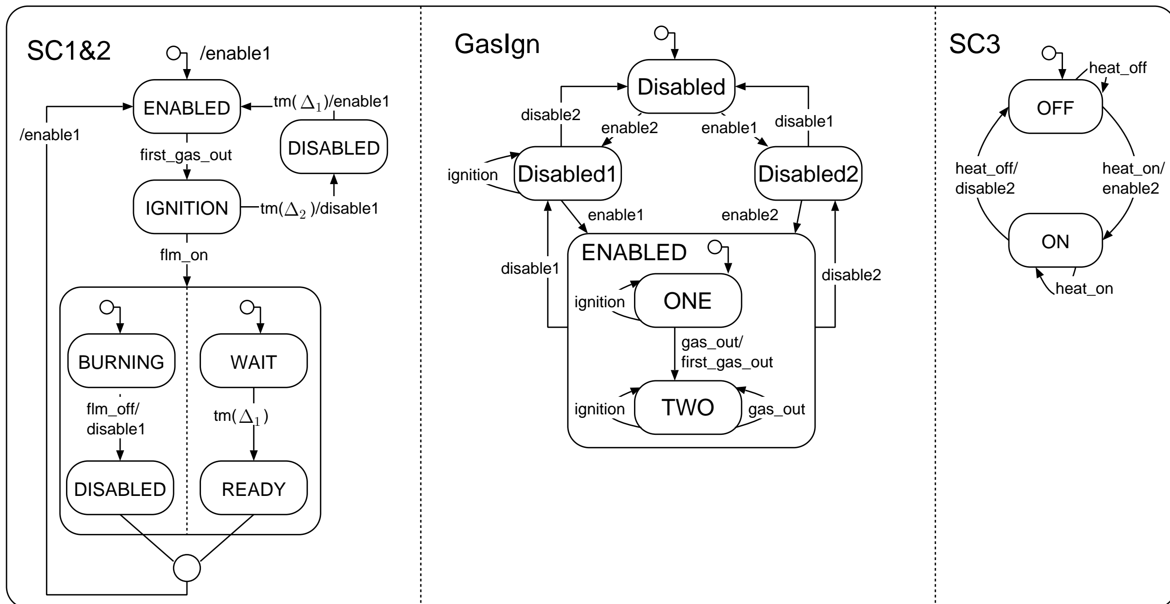


Figure 12.6: State machine illustrating the behaviour of the safety component.

$SC3 \mathrel{\widehat{=}} SC3_{OFF}$
$SC3_{OFF} \mathrel{\widehat{=}} heat_on \rightarrow enable_2 \rightarrow SC3_{ON} \;\square\; heat_off \rightarrow SC3_{OFF}$
$SC3_{ON} \mathrel{\widehat{=}} heat_off \rightarrow disable_2 \rightarrow SC3_{OFF} \;\square\; heat_on \rightarrow SC3_{ON}$

The above process terms are a relatively direct translation of the safety constraints. They are considerably simpler than a process description of the entire software controller, because they do not take into account functionality aspects.
Note that the safety component is always ready to engage in the events *flame_on*, *flame_off*, *heat_on* and *heat_off*. This is necessary, because the flame component and the human operator, respectively, must have the full control of these events. The safety component only "consumes" these events. The same is true of the functionality component.

The functionality component is responsible for "pushing" the function of the software controller. That is, it defines in a positive manner when it is necessary to send particular actuator commands; it does so independently of safety concerns.

SPEC. UNIT *FuncComp* **EXTENDS** *GBCConstants*

INTERFACE

port *heat_on*, *heat_off*, *flame_on*, *flame_off* domain *SYNC*;
port *ignition*, *gas_out* domain *SYNC*

LOCAL

`channel` *flm_on*, *flm_off* `domain` *SYNC*

BEHAVIOUR

The following process terms defining the behaviour of the functional component are best understood by considering the state machine depicted in Figure 12.7, which is a direct translation of these process terms.

$Behaviour \mathrel{\widehat{=}} Cycle \,|[\, \{flm_on, flm_off\} \,]|\, BufferOff$

$Cycle \mathrel{\widehat{=}} Idle \bigtriangleup (heat_off \rightarrow Cycle)$

The process terms *Idle*, *Ignition* and *Burning* correspond to the states of the state machine.

Again, the task of the process term *BufferOff* is to buffer the external events *flame_on* and *flame_off*, which are passed on by means of the internal events *flm_on* and *flm_off*.

$BufferOff \mathrel{\widehat{=}} flame_on \rightarrow BufferOn \,\Box\, flm_off \rightarrow BufferOff$
$BufferOn \mathrel{\widehat{=}} flame_off \rightarrow BufferOff \,\Box\, flm_on \rightarrow BufferOn$

The following three process terms define the recurring activities that are associated with particular states of the state machine.

$HeatOn \mathrel{\widehat{=}} heat_on \rightarrow HeatOn$

$GasIgn \mathrel{\widehat{=}} Wait\, \Delta_3;\; gas_out \rightarrow ignition \rightarrow GasIgn$

$Gas \mathrel{\widehat{=}} Wait\, \Delta_3;\; gas_out \rightarrow Gas$

$Idle \mathrel{\widehat{=}} heat_on \rightarrow Ignition$

$$Ignition \mathrel{\widehat{=}} (GasIgn \,|||\, HeatOn) \\ \bigtriangleup (flm_on \rightarrow Burning \rhd \{\Delta_2\}\; Idle)$$

$$Burning \mathrel{\widehat{=}} (Gas \,|||\, HeatOn) \\ \bigtriangleup (flm_off \rightarrow Idle)$$

The last component translates the measurements of the thermometer into events that indicate that a flame appeared and disappeared, respectively. The other components have been defined in terms of these produced events. The purpose of the flame component is thus to define an abstraction for the safety and function component.

SPEC. UNIT *FlameComp* **EXTENDS** *GBCConstants*

INTERFACE

`port` *temperature* `domain` *Temp*;
`port` *flame_on*, *flame_off* `domain` *SYNC*

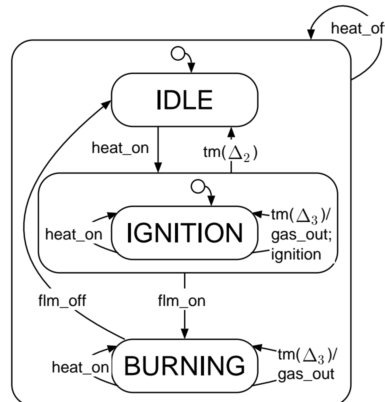


Figure 12.7: State machine illustrating the behaviour of the functional component.

TYPES & CONSTANTS

$Bool ::= Yes \mid No$

STATE

The data state records the state of the flame at the last measurement. It is needed in order to detect changes of the state of the flame. Initially, the flame is assumed to be off.

$State == [flame : Bool];$ $\quad Init == [\,State \mid flame = No\,]$

OPS & PREDS **[** $NoChange, Appeared, Disappeared$ **]**

The following operation schemas define all possible transitions of the flame's state (including the case that no transition takes place).

$$NoChange == [\,\Xi State;\ meas? : Temp \mid flame = No \wedge meas? < limit \vee flame = Yes \wedge meas? \geq limit\,]$$
$$Appeared == [\,\Delta State;\ meas? : Temp \mid flame = No \wedge meas? \geq limit \wedge flame' = Yes\,]$$
$$Disappeared == [\,\Delta State;\ meas? : Temp \mid flame = Yes \wedge meas? < limit \wedge flame' = No\,]$$

BEHAVIOUR

The behaviour of the flame component is cyclic. In a distance of Δ_3 time units, a measurement of the thermometer is requested. The subsequent behaviour depends on the relationship between the current data state and the received measurement *val*, determining the unique operation whose precondition is satisfied and thereby resolving the choice. Accordingly, *flame_on*, *flame_off* or no event is performed.

$Behaviour \mathrel{\widehat=} Cycle$

$Cycle \mathrel{\widehat=} Wait\ \Delta_3;\ temperature?val : Temp \rightarrow$
$\quad (NoChange_X.\llparenthesis\ meas? == val\ \rrparenthesis \rightarrow Cycle$
$\quad \Box\ Appeared_X.\llparenthesis\ meas? == val\ \rrparenthesis \rightarrow flame_on \rightarrow Cycle$
$\quad \Box\ Disappeared_X.\llparenthesis\ meas? == val\ \rrparenthesis \rightarrow flame_off \rightarrow Cycle)$

12.2.5 Discussion

To summarise, the gas burner case study has demonstrated the ability of RT-Z

- to cover different levels of abstraction, i.e., requirements and designs,
- to cover general system and software components and
- to cope with safety-critical applications.

The applicability of RT-Z to safety-critical systems is also demonstrated in [Sühl, 2000], where we have discussed the formal specification of a railway network.

We have, however, not established that our system and software designs really meet the constraints expressed in the system and software requirements specifications, respectively. As we discuss in the next chapter, the provision of verification techniques for RT-Z that would allow us to prove that a given concrete specification unit refines an abstract one is part of future work.

We have used a Statechart-like notation in order to discuss the behaviour of the safety and the functional components, specified by timed CSP process terms. One might therefore be tempted to conclude that Statecharts are a better candidate for being combined with Z. Indeed this is the approach taken by the ESPRESS project, in which μSZ [Büssow et al., 1997]—a combination of Z and Statecharts—was developed. Statecharts are undoubtedly more intuitive than CSP, because they are based on a graphical notation. However, they have important drawbacks compared to CSP.

Firstly, the Statechart notation lacks a precise, complete and commonly accepted semantics. There are several tools implementing Statecharts, and all have interpreted the notation in a slightly different manner. Harel and Naamad [1996], e.g., have dealt with the meaning of the Statechart notation as implemented in the STATEMATE tool. However, their semantic definition is on a rather low level of formality: it is basically an informal description of the step algorithm implemented by STATEMATE in the context of illustrating examples. Further, the described semantics does not have the useful characteristics of the denotational semantics of CSP, e.g., compositionality. More recently, formal definitions of the Statechart semantics have been proposed. Helke and Kammüller [2001], e.g., have formalised Statecharts based on hierarchical automata using the HOL instance of the generic theorem prover Isabelle. So far, however, only a subset of the language has been covered.

Secondly, in contrast to CSP, Statecharts are based on a model of asynchronous communication. From a theoretical point of view, asynchronous communication is less appropriate

than synchronous communication when reasoning about the behaviour of compound systems. Moreover, (basic) Statecharts do not have any interface/encapsulation mechanism. This makes it very difficult to model complex behaviour patterns, because a component broadcasting a signal can give rise to reactions in all other components. The incorporation of Statecharts into UML has alleviated this deficiency, because object-orientation provides the necessary encapsulation mechanisms. However, the formal meaning of UML Statecharts is even more a subject of current research.

In conclusion, while useful in illustrating the structural aspects of the behaviour of the safety and functional components, Statecharts are not as appropriate as CSP to define this behaviour formally (and to reason about it).

Chapter 13

Conclusions

We finally discuss the contributions of this thesis and some open issues remaining as subjects for future work.

13.1 Contributions

The following points concern the contributions of RT-Z to the research field dealing with the integration of formal specification languages.

Coherent Integration of Z and timed CSP. In this thesis, we have presented an integration of Z and timed CSP that allows us to specify all relevant aspects of real-time embedded systems in a coherent manner.

By developing the classification of approaches to integrating existing formalisms in Chapter 5, we have given an overview of the different (and frequently mutually contradictory) design rationales underlying different integrated formalisms. The classification has established a basis for comparing RT-Z with the related work.

According to our classification, we have conceived RT-Z as a conserving integration. The general benefits and drawbacks attributed to this class hence apply to the instance RT-Z. Most important, by having encapsulated the notations of Z and timed CSP within two separate parts, we have pursued a separation of concerns. This has helped us in defining the denotational semantics of RT-Z by reusing the semantic functions of the base formalisms and in developing refinement techniques for RT-Z by reusing the refinement techniques of the base formalisms. Moreover, the separation between the base formalisms enhances the intelligibility of RT-Z specifications for those being familiar with Z and timed CSP and makes it possible to use existing tools.

Coverage of Several Abstraction Levels. A distinguishing feature of RT-Z is that it covers several abstraction levels. All other approaches integrating (timed/untimed) CSP with Z/Object-Z have integrated only the concrete language constructs of the respective base

formalisms, resulting in integrated constructs that basically correspond to the *concrete* specification units of RT-Z. That is, they do not provide any counterparts of abstract specification units.

To define the formal meaning of abstract specification units, we have formalised the use of abstract data types in the predicate language of timed CSP. This use of abstract data types within predicates is common in the literature on timed CSP [Schneider, 1999b, Chapter 12], e.g., for quantifying over the time domain or over other data domains. However, this use is not underpinned by the denotational (and operational) semantics of timed CSP. So, our definitions can be regarded as a contribution to the formalisation of an aspect of the actual use of timed CSP.

Further, the formalisation of timed CSP's predicate language allowed us to include environmental assumptions in abstract and in concrete specification units. The ability to specify environmental assumptions is essential; it assists in clearly separating concerns, because the requirements to the system under consideration and the requirements to its environment are specified in separate sections of a specification unit. Environmental assumptions are a concept not provided by the other integrated formalisms explicitly.

Formal Foundation. We have formally defined the meaning of abstract and concrete specification units by means of a denotational semantics. Further, the meaning of the structuring operators of RT-Z has been defined by adopting a transformational approach. This stepwise semantic definition has been necessary, because a monolithic approach would have made it impossible to reuse the semantic definitions of the base formalisms.

We have spent much effort in proving that the definition of the semantics of concrete specification units is consistent with the axioms of the timed failures model of timed CSP (Appendix B). The proof has been carried out in a mathematical way.[1] The consistency ensures that the meaning that is associated with the Z part of a specification unit represents some timed CSP process. Only this fact has entitled us to interpret a specification unit as the parallel composition of its two parts.

The formal definition of the semantics is the foundation for formal specification. Formal specification is an activity that is useful in its own rights. This claim is supported by the following points.

- Perhaps the most important advantage of a formal specification is its ability to clearly, unambiguously express requirements, constraints and properties of a system. When accompanied by appropriate informal explanations, a formal specification facilitates the discussion between different stakeholders of the system development process.

- Formal specifications are in general, if a suitable abstraction has been identified, more concise than corresponding informal descriptions.

[1] This certainly restricts the level of confidence as compared to a formal proof (e.g., by using a theorem prover). However, the granularity of the proof is fine enough to convince us that the principles of the semantic definitions are sound.

- Using a formal language, one is forced to think more rigorously about the considered system, which helps gaining a deeper understanding of the system and its requirements. In fact, the early phases of the development process are extended due to a more thorough analysis; but this investment is returned in the later phases.
- A formal requirements or design specification is a sound basis for the still indispensable task of black-box testing; on the basis of a formal specification one is able to derive test cases to later exercise the implemented system as well as to decide on the correctness of the system's reactions.

However, the full potential of the mathematical basis of a formalism can only be exploited by formal proof techniques that allow one to establish in a rigorous way that specifications exhibit certain properties. As already stated, we have not yet defined a calculus for RT-Z that would allow us to formally derive properties of RT-Z specification units. However, the foundation for such a calculus has already been established. Its fundamental prerequisite is the formal semantic definition, which we have accomplished. Moreover, we have designed the integration of Z and timed CSP in such a way that proof obligations to a specification unit can be reduced to separate proof obligations to its parts. These, in turn, can be discharged by means of the proof techniques provided by the two base formalisms. This separation of concerns, by the way, is the very essence of conserving integrations.

Formal Definition of Structuring Operators. We have chosen plain Z in favour of Object-Z, fully aware of the fact that plain Z does not provide as elaborate structuring operators as Object-Z. The reason for this choice has been our intention to use the combined expressive power of Z and timed CSP for the specification at both the component and the system level. This is in contrast to the approach of Smith and Derrick [2001] who used Object-Z to specify the behaviour of single components and CSP to specify the interactions between components.

Having chosen plain Z, we needed to define our own structuring operators on top of the integration. The structuring operators of RT-Z have been backed up by means of a transformational approach, see Chapter 8. This separate definition is in contrast to TCOZ, for which the authors claim that they can use for free the structuring operators of Object-Z as containers for the integrated notation. However, as we have argued in detail in Chapter 11, the semantics of these Object-Z operators is not defined for the integrated language.

Global Invariants. As discussed in Section 9.2.1, a major achievement of the semantic model underlying RT-Z and at the same time a major source of complexity is its ability to cope with the concurrent execution of operations that work on different subspaces of the overall data state but which might nevertheless interact. This interaction is expressed in terms of global invariants.

To the best of our knowledge, this concept is novel in the realm of integrated formalisms. It is vital for our approach in the system-related phases in the context of safety-critical applications, because the safety of a system must be expressed in terms of its states: since the overall state of a system may be distributed to several components, safety constraints may include relationships between the states of different components. Expressing these relationships is exactly the purpose of global invariants.

Refinement. We have formally defined the notion of refinement between specification units, and we have shown how to use the refinement techniques provided by Z and timed CSP in order to establish the refinement relationship between specification units in RT-Z. This connection has been underpinned by our proof in Appendix C that the existence of a forward or backward simulation between the Z parts of two specification units implies the refinement relationship between the corresponding timed CSP interpretations in the timed failures model.

We have also demonstrated that refinement relationships between compound specification units can be reduced to refinement relationships between pairs of corresponding component specification units. This compositionality of the refinement relationship is a necessary precondition for establishing refinement relationships between compound specification units in a convenient manner.

13.2 Satisfaction of Objectives

Recall the objectives that we have formulated in Chapter 3. RT-Z, as presented in this thesis, satisfies these objectives for the most part.

First, RT-Z is a formally defined specification language allowing one to clearly and precisely express requirements, constraints and designs. It also allows the specifier to freely choose the appropriate level of abstraction. Regarding rigour, we have already stated that we have not yet provided verification techniques that would allow us to reason about RT-Z specifications rigorously, but that we have established the foundation for such verification techniques, see the discussion in the next section.

Second, the base formalisms of RT-Z have been intentionally chosen in order to cover the relevant dimensions of the behaviour of real-time embedded systems. On the other hand, we have deliberately dispensed with taking into account hybrid and stochastic behaviour, which makes RT-Z tremendously simpler (however, with the drawback that some types of embedded systems cannot be modelled adequately). So, to conclude, we think that we have made a reasonable compromise between the expressiveness and the complexity of the model underlying RT-Z.

Third, as demonstrated by the case studies in Chapter 12, the language constructs provided by RT-Z allow us to cover the phases of the development process referred to in the objectives.

Last, but not least, the structuring operators of RT-Z, in particular aggregation and extension, ensure the scalability of RT-Z. This is witnessed by the multi-lift case study in Section 12.1, which concerns a moderately complex system.

13.3 Future Work

The following points concern the features that—in spite of being worthwhile—are not accomplished so far. Accomplishing these points would lift RT-Z from the level of formal specification languages to the level of formal methods.

Calculus. A prominent part of future work is the development of a calculus for RT-Z. It will consist of a collection of axioms and inference rules allowing us to derive properties from an RT-Z specification that are related to both behavioural and state-oriented aspects.

As stated in Chapter 6, an essential design rationale behind our integration of Z and timed CSP has been the possibility to reuse the infrastructure of the base formalisms as far as possible. This applies particularly to the existing proof techniques of Z and timed CSP. The reuse is facilitated by the separation of the base formalisms, as generally motivated in Chapter 5 for the class of conserving integrations.

In RT-Z, the notation used to express properties is the notation integrated in abstract specification units. So, to be more precise, the purpose of the calculus we are striving for is to derive properties, as expressed by abstract specification units, from models, as expressed by concrete specification units. A particular purpose of these rules will be the proof of refinement relationships between abstract and concrete specification units.

The rules of the calculus we are aiming at will be related to two different tasks: first deriving properties from a single specification unit; and second deriving properties from a compound specification unit, given the properties of its components.

Considering the first task, the aim is to derive those properties of a concrete specification unit that depend on both the Z part and the CSP part. These can be either properties that can be *expressed* only by referring to the two parts, or it can be properties that can be *proved* only by employing properties of the two parts. The axioms and inference rules will express the mutual relationships between the base formalisms in a concrete specification unit as defined by the semantic model in Chapter 9; they will reduce properties depending on the two parts to properties depending on only one part, which can be discharged by applying the proof techniques of the respective base formalisms. The rules will reflect the relationships (among others)

- between a termination event and the post state resulting from executing the operation,
- between an invocation event and the pre state to which the operation is applied,
- between the value transmitted with a termination event (binding that associates each output parameter with a value) and the output parameters of the corresponding operation, and
- between the value transmitted with an invocation event and the input parameters of the corresponding operation

as defined by the semantics.

The second task deals with deriving properties of compound specification units from properties of their component specification units. The corresponding inference rules will reflect the reduction processes defined for aggregation and extension in Chapter 8.

For both Z and timed CSP, there are techniques for formally deriving properties from specifications. The sound and complete calculus of timed CSP is outlined in Section A.2.6. Z, by contrast, is not equipped with such a complete calculus. However, since the Z notation is based on predicate logic and set theory, the well-established mathematical techniques developed for these concepts can be used. Of particular interest are the proof tools developed

for Z, e.g., HOL-Z [Kolyang et al., 1996] and Z/EVES [Saaltink, 1997], automating various kinds of proof tasks.

Finally, the axioms and inference rules we are aiming at must be proved sound with respect to the denotational semantics of RT-Z defined in Chapter 9. Completeness of these rules—although being desirable—is not realistic.

Examples. In the following, we attempt to give an idea of the nature of the inference rules we are striving for by means of some examples. All examples deal with deriving properties from single specification units.

The first inference rule reflects the relationship between the precondition of an operation schema, a data state in which the corresponding operation is invoked and the input parameters transmitted with a corresponding invocation event. The two antecedents of the inference rule are related only to the Z part of the concrete specification unit under analysis. They can be discharged using the proof techniques of Z. The conclusion is related to the CSP part: it makes a statement about the liveness of an invocation event.

Consider an operation schema Op with input parameters $in1?, \ldots, inN?$. Suppose the data state present at time t fulfils property $Prop$. Suppose further that $Prop$ implies the precondition of Op, with the input parameters fixed to the values $val1, \ldots, valN$.

(Precondition)

$$\frac{\begin{array}{l} Prop \ \textsf{at}\ t \\ Prop \Rightarrow \textsf{pre}[\, Op \mid in1? = val1 \land \ldots \land inN? = valN \,] \end{array}}{Op_I.\llparenthesis\, in1? == val1, \ldots, inN? == valN \,\rrparenthesis \ \textsf{live}\ t}$$

Then we can deduce that the invocation event related to this operation and associating the input parameters with the given values is offered at time t.

The next rule concerns the relationship between the input parameters of an invocation event and the output parameters of the corresponding termination event.

The first three antecedents refer only to the CSP part of the concrete specification unit under analysis, namely to the occurrence of invocation and termination events. The last antecedent, which assumes that the operation schema Op establishes a relationship between the input and output parameters of each operation application, is related only to the Z part and can be discharged using the proof techniques of Z.

(InOutputRel)

$$\frac{\begin{array}{l} Op_I.\llparenthesis\, in1? == valI1, \ldots, inN? == valIN \,\rrparenthesis \ \textsf{at}\ t1 \\ Op_T.\llparenthesis\, out1! == valO1, \ldots, outM! == valOM \,\rrparenthesis \ \textsf{at}\ t2 \\ \forall\, t3 : [t1, t2) \bullet \neg\, Op_T \ \textsf{at}\ t3 \\ Op \vdash PropInOut[in1?/x1, \ldots, inN?/xN, out1!/y1, \ldots, outM!/yM] \end{array}}{PropInOut[valI1/x1, \ldots, valIN/xN, valO1/y1, \ldots, valOM/yM]}$$

The conclusion transfers the established relationship to the particular values transmitted with a pair consisting of an invocation and a termination event.

The last rule deals with additional state invariants valid during certain time intervals. Assume we consider a certain subset of operations *OpSet* that preserve an implicit invariant *Inv* in addition to the explicit state invariant *State*. This means that each operation of this subset has to terminate in a post state satisfying this implicit invariant, provided it is applied to a pre state satisfying it. Moreover, suppose we are able to show that within the interval (t, t') only termination and execution events belonging to this subset occur. Finally, if we know that the last data state recorded before this interval fulfils the implicit invariant mentioned above,

(PartialInvariant)

$$\frac{\begin{array}{l}\forall\, Op : OpSet \bullet (Inv \Rightarrow \neg\,(\mathsf{pre}\, Op)) \vee ((Op \wedge Inv) \Rightarrow Inv') \\ Inv \;\mathsf{at}\; t \\ s \restriction_{ttr} (\{op : OPS \setminus OpSet \bullet TermOf\; op\} \cup \\ \qquad \{op : OPS \setminus OpSet \bullet ExecOf\; op\}) \uparrow_{ttr} (t, t') = \langle\rangle\end{array}}{\forall\, t'' : (t, t') \bullet Inv \;\mathsf{at}\; t''} \quad [\, OpSet \subseteq OPS \,]$$

then we can deduce that all data states during this interval fulfil this implicit invariant, too.

To conclude, the above examples should have made clear the principle that properties depending on the two parts of a specification unit are reduced to properties depending only on a single part.

Methodological Support. According to Grieskamp et al. [2001], "a major drawback of formal techniques is that they are not easy to apply for the average software engineer. Besides the facts that users of formal techniques need an appropriate education and have to deal with lots of details, they are often left alone with a mere formalism without any guidance on how to use it. Hence, methodological support is a key issue to bring formal techniques into practice." We acknowledge that this also applies to RT-Z. As mentioned above, a future goal is to lift RT-Z to the level of formal methods.

Initial results towards this goal have already been achieved. In particular, in [Heisel and Sühl, 1997] we have presented methodological support for a predecessor of RT-Z. Heisel [1997] has dealt with the methodological and machine support of formal techniques in the context of software engineering more generally and more comprehensively.

Operational Semantics. There are several approaches to defining the semantics of specification languages. In this thesis, we have used the denotational approach. The advantage of the denotational approach has been the possibility to uniformly cover the concrete (e.g., process terms) and abstract language constructs (e.g., predicates).

There are, however, tasks for which an operational semantics is more appropriate. For instance, if RT-Z is to be supported by a model checker (analogously to the support of CSP by FDR [Formal Systems (Europe) Ltd, 2000]), an operational semantics is closer to the implementation of a model checker than the denotational semantics. The definition of an operational semantics must be accompanied by the demonstration of its congruence to the denotational semantics.

Formalisation in Isabelle/HOL. The formalisation of the semantic model of RT-Z in a theorem prover, e.g., the HOL instance of the generic theorem prover Isabelle [Paulson, 1996], is a promising task. It would allow us to implement the calculus discussed above in that theorem prover and thus to automate the process of discharging proof obligations. This automation would also increase the level of confidence in the validity of proofs.

Formalising RT-Z in Isabelle/HOL need not begin from scratch. There are formalisations of Z [Kolyang et al., 1996] and (plain) CSP [Tej and Wolff, 1997] in HOL/Isabelle, which could be taken as a basis.

Other Tool Support. For RT-Z to be able to cope with more complex systems than those case studies presented in the previous chapter, tool support is indispensable. We can distinguish between two types of tool support for RT-Z specifications.

- Existing tools for Z that can be applied to the Z part of RT-Z specification units.
- Tools that are specific to RT-Z.

The latter ones would be applied to specification units as a whole, e.g., in order to check dependencies between the parts. This would involve consistency checks and plausibility checks, e.g.,

- the consistency of data domains associated with corresponding ports of two specification units that are connected by aggregation;
- the presence of an operation schema in the Z part for each operation-related event referred to in the CSP part;
- the reference of each (non-auxiliary) operation schema in the Z part by the CSP part.

Proof support for RT-Z, e.g., for checking refinement between specification units, would go beyond mere consistency and plausibility checks. It would be based on a formalisation of the denotational semantics (or a congruent operational semantics) of concrete and abstract specification units and the transformation rules for aggregation and extension in a theorem prover or model checker, or both.

To conclude, in developing RT-Z we have made substantial contributions to the research field dealing with the integration of formalisms. Once the goals outlined in this section are achieved, RT-Z can be regarded as a full-fledged formal *method*.

Part V

Appendices

Appendix A

Base Formalisms

In this appendix, we give a brief overview of the two specification languages that constitute the integrated formalism RT-Z. The Z notation is dealt with in Section A.1, and timed CSP is the subject of Section A.2. In Section A.2.8, we refer to some real-time approaches related to timed CSP.

A.1 Z Notation

The Z notation is a formal specification language, which is in widespread use in academia and—to a restricted extent—also in industry. It is a strongly typed language based on first-order predicate logic and set theory.

We assume some familiarity with the Z notation throughout this thesis, for a detailed account of Z consult [Woodcock and Davies, 1996]. One of the merits of the Z notation is the emerging international standard, which will certainly promote the production of tool support for Z and its further dissemination. In the remainder of this section, we briefly discuss the definitions of the standard that we use in this thesis.

A.1.1 Standardisation

The ISO standard for Z is now near completion. The version of the standard to which we refer throughout this thesis is the Draft International Standard [ISO, 2002]. We term this document 'Z Standard.' In this section, we discuss the definitions of the Z Standard that we need in this thesis, in particular for the definition of the semantics of the integrated formalism RT-Z in Part III.

The scope of the Z Standard is the definition of the syntax, the type system and the semantics of the Z notation. No method of using the Z notation is prescribed.

Concerning the syntax, the Z Standard introduces one main novelty compared with the version of the Z notation defined in [Spivey, 1992]: Z specifications are structured into *sections*. More precisely, a Z specification consists of a sequence of sections. Each section in a Z specification can *inherit* from other sections that precede it. Inheritance means that all definitions

in the inherited section are available in the inheriting section. There is a dedicated section *prelude*, which is implicitly inherited by all sections of a Z specification; it contains general-purpose definitions, e.g., the natural numbers.

The Z Standard defines the formal meaning of Z specifications by means of a denotational semantics. Roughly speaking, the meaning of a Z specification is given "in terms of the semantic values that its global variables may take consistent with the constraints imposed on them by the specification" [ISO, 2002, p. 71].

The mathematical metalanguage used to define the denotational semantics is based on classical first-order logic and Zermelo–Fraenkel (ZF) set theory. It is similar in appearance to the notation of Z itself. In ZF set theory, there are only sets. Members of sets can only be other sets. Structures like tuples and natural numbers are not primitive in ZF set theory, but there are well established ways of representing them, see [Hamilton, 1982] for details.

The Z Standard introduces the symbol $\mathbb{U}$ to denote "the universe—a world of sets—providing semantic values for all Z expressions (including generic expressions)" [ISO, 2002]. $\mathbb{U}$ is assumed to be big enough to contain the set *NAME*, from which Z names are drawn, and an infinite set. Further, $\mathbb{U}$ is supposed to be closed under the formation of power sets and products. According to the Z Standard [ISO, 2002], "the formation of a suitable $\mathbb{U}$ comprising models of sets and tuples, as needed to model Z, is well-known in ZF set theory." On this basis, the symbol $\mathbb{W}$ is introduced by the Z Standard to denote the world of semantic values for Z expressions that are not generic, i.e., $\mathbb{W} \subset \mathbb{U}$.

The Z Standard proceeds with defining the structure of the denotations of Z specifications. To this end, the symbol *Model* is introduced to denote the set of models, which associate the names of global variables declared by a Z specification with semantic values ($Model == NAME \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \mathbb{U}$), and *SectionModels* is introduced to denote the set of section models, which are functions from section names to their sets of models ($SectionModels == NAME \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \mathbb{P}\, Model$).

The semantics of the different syntactic categories of the Z notation—specifications, sections, paragraphs, expressions, predicates and types—is defined by corresponding semantic relations from the Z syntax to the semantic universe in terms of operations of ZF set theory. We deal only with the semantic relations to which we refer in this thesis.

- $[\![spec]\!]^{\mathcal{Z}}$ denotes the meaning of the specification *spec*. The meaning of a Z specification is a function mapping its section names to sets of models. For each section name of a Z specification, each model related to that section name maps the global variables declared in that section to a semantic value that is consistent with all constraints imposed on the variable in that section.
- $[\![pred]\!]^{\mathcal{P}}$ denotes the meaning of the predicate *pred*. The meaning of a predicate is the set of models in which the predicate is true.
- $[\![expr]\!]^{\epsilon}$ denotes the meaning of the expression *expr*. The meaning of an expression is a function that maps each model to the semantic value that is the result of evaluating the expression with respect to the given model.

Although the Z Standard does not prescribe any method of using the Z notation, the prevalent conventions of using the Z notation in a state-oriented way (i.e., as a model-based

specification language) are described in an annex. Following these conventions, a system is specified in terms of its states and the operations on these states.

We refer to these conventions in this thesis, because they reflect the way in which the Z notation is used in the context of concrete specification units.

A.1.2 Discussion

Z is a powerful notation for describing and analysing a system in terms of its states and operations. It is particularly adequate for systems whose primary nature is that of a data-processing system, i.e., for systems that have a non-reactive nature.

The schema calculus is a characteristic feature of the Z notation, which makes it superior to an unstructured use of mathematical logic and set theory. It provides means for structuring Z specifications: "mathematical objects and their properties can be collected together in *schemas*: patterns of declarations and constraints" [Woodcock and Davies, 1996].

Moreover, the Z notation is based on well-established and well comprehensible concepts, namely set theory and predicate logic. This makes it relatively simple to learn using Z.

Another merit of Z is that it is strictly typed. Every entity in a Z specification has a unique type. This allows one to type-check Z specifications which helps detect certain classes of errors and improve the consistency of Z specifications.

In spite of these strengths, several drawbacks are attributed to the Z notation.

First, and most important, the Z notation is not designed to deal with behavioural aspects, e.g., the temporal order in which operations are to be performed or the distinction between operations that are to be available at the interface of a system and operations that have only auxiliary purposes.

Second, it is not adequate to fix architectural aspects using the Z notation, because it is not designed to define aspects of a system architecture such as concurrency, distribution or communication.

Finally, the Z notation provides only rudimentary means for structuring specifications. Schemas are the main facility of the Z notation to structure definitions, but they are not adequate to decompose the specification of a complex system into encapsulated components.

These drawbacks have been our main motivation—and also the motivation of others—to seek for a complementary formalism that has its strengths where the Z notation has its weaknesses. The next section deals with such a formalism.

A.2 Timed CSP

In this section, we introduce the real-time process algebra timed CSP. Since most people are not familiar with timed CSP, we treat it in greater depth.

Timed CSP is a real-time extension of the more widespread process algebra CSP, originally introduced by Hoare [1985] and further developed by Roscoe [1998]. We assume some familiarity with process algebras in general and with CSP in particular.[1]

Earlier versions of timed CSP (sometimes also termed real-time CSP) were described by Schneider and Davies [Schneider, 1990, Davies, 1993, Davies and Schneider, 1995]. The version of timed CSP that we use in this thesis is that treated by Schneider [1999b]. Our description of timed CSP in this section is closely based on the last reference.

Let us first outline the basic concepts underlying CSP. The language of Communicating Sequential Processes (CSP) was designed for the description and analysis of systems that consist of *interacting* components. Components are described in CSP by processes. *Processes* are thought of as being independent, self-contained entities with particular interfaces through which they interact with their environment. Processes can be combined with other processes to form larger processes, i.e., the way of describing processes is compositional.

Since processes interact with other processes only through their interface, the relevant information derived from a process description is its behaviour at the interface: the internal structure of processes is not relevant when describing and analysing the behaviour of the system that is constituted by these processes. The *interface* of a process consists of a set of events. An *event* represents a particular kind of atomic action that can be performed with the participation of the process. In describing a process, the first issue to be decided is the set of events which it can perform. The interface of a process can be considered as its static specification. Its dynamic specification describes how it will actually behave at the interface, i.e., it describes the patterns of events in which it can participate.

The universe of events is denoted by Σ. Events are thought of as occurring on particular *channels*, which link processes with each other. A channel can be associated with an alphabet, containing the values that can be carried by events occurring on that channel. An event occurring on a channel with an associated alphabet consists of two components: the channel c on which it occurs and the value v that is carried by the event.[2]

The language of (untimed) CSP and its associated semantic models, e.g., the traces model or the infinite failures–divergences model, are appropriate for specifying and analysing systems in terms of the order in which they can perform events. These models have deliberately abstracted from timing aspects such as the precise times at which events occur or the delays before events become enabled. For systems whose correct operation depends only on the order in which events are performed, these semantic models provide the appropriate level of abstraction, containing no unnecessary details.

Systems, by contrast, whose correct operation depends on the precise real-time behaviour can be analysed only in the context of a semantic model that introduces an explicit notion of time. Examples of relevant aspects of the real-time behaviour are maximal response times or scheduling conditions. The language of timed CSP and its associated semantic model extend CSP in two ways in order to introduce a notion of time. First, the process operators of

[1] Fidge [1994] provides an overview of several process algebras and their underlying concepts.

[2] In the remainder of this section, we abstract from the "internal structure" of events by denoting them by atomic identifiers.

CSP are adopted and re-interpreted on a more concrete, timed level. Second, additional constructs are introduced that allow the precise description of time-sensitive behaviour. These additional constructs are delay, timeout and timed interrupt.

This section is organised as follows. We introduce the language of timed CSP along with an outline of its operational semantics in Section A.2.2. The properties of the timed transition system that is induced by the operational semantics are discussed in Section A.2.3. The structure of timed observations that can be made of processes and the denotational semantics of timed CSP are treated in detail in Section A.2.4. Of particular interest when discussing the denotational semantics is the treatment of recursion. Then, in Section A.2.5, the predicate language of timed CSP is introduced, which allows one to specify properties that processes are required to satisfy. Finally, we outline the calculus provided by timed CSP in order to verify that processes satisfy specified properties.

A.2.1 Computational Model

We first discuss the assumptions that underlie the interpretation of timed CSP processes in the context of real-time.

Instantaneous events: Timed CSP considers processes as concurrent, synchronising components. Events represent synchronisations between concurrent processes. It is hence natural to consider events as instantaneous, occurring at a single instant of time and associated with no duration.

To model actions that need time, several events must be used in order to mark the corresponding intervals.

Newtonian time: It is assumed that there is a single conceptual global clock. Time progresses at a constant rate in all processes of a system with reference to this global clock. In this way, the simultaneity of events occurring in concurrent processes can be judged. The processes do not really read the global clock; rather it is a concept used to define the meaning of processes.

Real-time: The time domain of timed CSP are the positive real numbers. There is hence no minimal delay between events occurring at different time instants.

Maximal parallelism: Concurrent processes are supposed not to compete for resources, i.e., they are supposed to run on dedicated processors having sufficient processor time and memory. This maximal parallelism assumption means that concurrent processes need to synchronise only when this is explicitly specified. In particular, there are no implicit scheduling assumptions. When scheduling is to be taken into account, it needs to be modelled explicitly.

Maximal progress: An event occurs precisely when all participating processes are ready to engage in it. An internal event, which does not require the participation of the process environment, occurs as soon as the process is ready to perform it. An external event, by contrast, requires the participation of the process *and* its environment: it occurs as soon as both participants are ready to engage in it.

The computational model incorporates the most basic design decisions underlying the language of timed CSP. It embodies the decisions which aspects of the system behaviour are considered relevant (and can thus be modelled directly) and, conversely, from which other aspects can be abstracted.

A.2.2 Process Term Language and Operational Semantics

In this section, we introduce the language of timed CSP processes, and its operational semantics is defined in terms of the transitions that processes can undergo. The operational semantics induces a timed labelled transition system, which contains all executions that timed CSP processes can perform.

The possible executions of processes are described in terms of transitions. Due to the decision to treat events as instantaneous, taking no time, transitions that represent the occurrence of events are not associated with the passage of time. Therefore, additional evolution transitions are introduced representing the progress of time. This separation of event and evolution transitions has the advantage that the process operators already present in (untimed) CSP can be defined consistently with their meaning in (untimed) CSP: the event transitions can be adopted, and additional evolution transitions are defined.

In the timed context, the event transition $P \xrightarrow{\mu} P'$ means that process P is immediately able to perform event μ, and it will result in process P' when doing so. The transition from process P to process P' will take no time, i.e., the global clock will not change during the transition.

The evolution transition $P \overset{d}{\rightsquigarrow} P'$ means that process P is able to evolve for d time units, and will result in P' when doing so. After this transition, the state of the global clock will have advanced by d time units. During an evolution transition, no event transition can occur. If P is able to evolve through a sequence of states $\{P_i\}$ over all time without performing any events, then the abbreviation $P \overset{\infty}{\rightsquigarrow}$ is used to denote this. Formally,

$$P \overset{\infty}{\rightsquigarrow} \;= \exists \{P_i\}_{i \in \mathbb{N}}, \{d_i\}_{i \in \mathbb{N}} \bullet (\sum_{i=0}^{\infty} d_i = \infty \wedge P = P_0 \wedge \forall i \bullet P_i \overset{d_i}{\rightsquigarrow} P_{i+1}).$$

In the following, we introduce all operators of the process term language of timed CSP. For selected operators, we also provide their associated transition rules.

Performing Events

Deadlock. The timed CSP process *Stop* represents deadlock. Since this process is unable to perform any event, no event transition rule is associated with it. On the other hand, the process *Stop* allows time to progress, which results in the following evolution transition rule.

$$\frac{}{Stop \overset{d}{\rightsquigarrow} Stop}$$

Event Prefix. The process $a \to P$ is immediately ready to engage in the event a. It remains in its initial state, continually offering a, until a occurs. The subsequent behaviour is due to P. Since events are instantaneous, control passes instantly to process P when a occurs. The above is formalised by the following rules.

$$\frac{}{(a \to P) \xrightarrow{a} P} \qquad \frac{}{(a \to P) \overset{d}{\rightsquigarrow} (a \to P)}$$

Timed Event Prefix. The process $a@u \to P$ is a more general form of event prefix. It allows us to record the time elapsed between the initial offer of the event a (the time the process starts executing) and the occurrence of a. The time variable u can have free occurrences in P; it is bound at the instant of time at which a occurs.

The process is immediately ready to engage in a, and offers a continually until it occurs. The subsequent behaviour is that of P with the time variable u substituted by the time value at which a has occurred (the time elapsed between the process has started and a has occurred).

Prefix Choice. The behaviour of the prefix choice $x : A \to P(x)$ is similar to that of the event prefix. It is immediately ready to engage in each of the events in A. It remains in its initial state, continually offering all events in A, until an event in A, say a, occurs. In this case, the variable x is bound to a, and the subsequent behaviour is determined by $P(a)$.

Successful Termination. The process *Skip* represents the process that is immediately ready to terminate (modelled by the dedicated event ✓), remaining in this state until termination occurs, and that deadlocks afterwards.

Note that the termination event is treated like a usual external event, i.e., the termination of a process requires the participation of the environment.

Choice

Timeout. The timeout construction $P \rhd\{d\}\ Q$ provides a time-sensitive choice between the processes P and Q. When the process starts execution, P has control. If P performs an external event within the first d time units, then the choice is resolved in favour of P. It may also perform internal events, but this does not resolve the choice. When, on the other hand, d time units have elapsed without P having performed an external event, a timeout occurs and the choice is resolved in favour of Q, i.e., P is discarded and control is passed to Q, which starts executing at the instant of time at which the timeout occurs.

The transition rules for the timeout operator implement the mechanism that ensures that internal events are urgent: the timeout is modelled by the internal event, which is denoted by τ. It must occur exactly at the time instant when the timeout duration has elapsed. This is achieved by ensuring that all evolution transitions cannot lead beyond the timeout duration: when d time units have elapsed, the only transition that is enabled is the event transition related to the timeout. This forces the timeout event to occur precisely when the timeout duration has elapsed.

Before the timeout occurs, the transitions of the whole timeout process are that of P.

$$\frac{P \xrightarrow{\mu} P'}{P \triangleright\{d\}\ Q \xrightarrow{\mu} P'}\ [\ \mu \in \Sigma^{\checkmark}\ ] \qquad \frac{P \xrightarrow{\tau} P'}{P \triangleright\{d\}\ Q \xrightarrow{\tau} P' \triangleright\{d\}\ Q}$$

$$\frac{P \overset{d'}{\rightsquigarrow} P'}{P \triangleright\{d\}\ Q \overset{d'}{\rightsquigarrow} P' \triangleright\{d - d'\}\ Q}\ [\ d' \leq d\ ] \qquad \frac{}{P \triangleright\{0\}\ Q \xrightarrow{\tau} Q}$$

Delay. The process *Wait* d represents a delayed termination. It delays for exactly d time units, unable to perform any event, and is ready to terminate afterwards. This process is a derived process defined as $Stop \triangleright\{d\}\ Skip$.

External Choice. The process $P \,\square\, Q$ provides a choice between its component processes, which is resolved in favour of the process that performs the first external event. The external choice operator allows its component processes to evolve independently, enabling and withdrawing events as time progresses. The occurrence of internal events does not resolve the choice.

Note that the environment has control over the resolution of the choice by offering an initial external event of either component process at a particular time instant.

Internal Choice. The process $P \sqcap Q$ behaves either like P or like Q. Its environment, however, has no control over the resolution of this choice. It is not fixed when this choice is resolved. The resolution of this nondeterministic choice may be resolved by the implementor, or it may be resolved only at run-time. Even at run-time, it can be delayed until the point at which an event is performed that could be offered only by one of the two component processes.

The internal choice operator must be considered as a specification construct. It defines a process in terms of the alternative behaviours it can exhibit, but it gives the environment no control over which alternative is chosen.

Indexed Internal Choice. The indexed internal choice $\bigsqcap_{i \in I} P_i$ generalises the binary internal choice. The choice is resolved in favour of an arbitrary P_i with $i \in I$.

Concurrency

The assumptions underlying the computational model—in particular the maximal parallelism, the maximal progress and the Newtonian time assumptions—have a direct impact on the behaviour of concurrent processes. First, maximal parallelism in connection with Newtonian time means that concurrent processes must always perform evolution transitions together: they are not scheduled sequentially and time must progress at the same rate

in all of them. Second, maximal progress forces events that are in the synchronisation set[3] of concurrent processes to occur exactly when all participants are ready to engage in these events.

Alphabetised Parallel. In the parallel composition $P \,|[\, A \mid B \,]|\, Q$, each of the component processes is associated with an alphabet. Within the intersection of the alphabets, the component processes must synchronise, and outside this intersection they can perform events independently. Moreover, they must synchronise on termination. As mentioned, time must progress at the same rate in all component processes.

$$\frac{P \xrightarrow{\mu} P' \qquad Q \xrightarrow{\mu} Q'}{P \,|[\, A \mid B \,]|\, Q \xrightarrow{\mu} P' \,|[\, A \mid B \,]|\, Q'} \quad [\, \mu \in (A \cap B)^{\checkmark} \,]$$

$$\frac{P \xrightarrow{\mu} P'}{P \,|[\, A \mid B \,]|\, Q \xrightarrow{\mu} P' \,|[\, A \mid B \,]|\, Q} \quad [\, \mu \in (A \cup \{\tau\}) \setminus B \,]$$

$$\frac{Q \xrightarrow{\mu} Q'}{P \,|[\, A \mid B \,]|\, Q \xrightarrow{\mu} P \,|[\, A \mid B \,]|\, Q'} \quad [\, \mu \in (B \cup \{\tau\}) \setminus A \,]$$

$$\frac{P \overset{d}{\rightsquigarrow} P' \qquad Q \overset{d}{\rightsquigarrow} Q'}{P \,|[\, A \mid B \,]|\, Q \overset{d}{\rightsquigarrow} P' \,|[\, A \mid B \,]|\, Q'}$$

Interleaving. In the process $P \,|||\, Q$, the component processes can execute independently of each other. Each external and internal event is performed by precisely one of the components. The only exception is the termination event $\checkmark$: its occurrence requires the participation of the component processes. Time must progress in the parallel processes at the same rate.

There is also an indexed form of interleaving $|||_{i \in I} P_i$, in which a collection of processes P_i are interleaved, one process for each member of the index set I. The index set must be finite. Since the binary interleaving operator is commutative and associative, indexed interleaving can be traced back to the binary operator.

Interface Parallel. Interface parallel composition $P \,|[\, A \,]|\, Q$ is a hybrid form, i.e., it combines the features of the alphabetised parallel composition and interleaving. P and Q must synchronise on all events in the interface set $A^{\checkmark}$, and they interleave on all other events. Again, time must progress in the parallel processes at the same rate.

[3] The synchronisation set is the set of events on which the parallel processes must synchronise.

There is also an indexed form of interface parallel composition $\|_{i \in I}[A]P_i$, in which a collection of processes P_i are composed in parallel, one process for each member of the index set I. All processes P_i must synchronise on events in the set $A^{\checkmark}$, and they can perform events outside this set independently. The index set must be finite. Since the binary interface parallel operator is commutative and associative, the indexed interface parallel operator can be traced back to the binary one.

The interface parallel operator can be considered as the basic form of parallel composition from which the alphabetised composition and the interleaving composition can be derived as special cases.

Abstraction

Hiding. In the process $P \setminus A$, all events in A are hidden from the environment. Hiding an event means to remove it from the process interface, so the environment is no longer required to participate in its occurrence. The encapsulation achieved by the hiding operator consists in identifying *all* processes that participate in an event.

The maximal progress assumption requires an event to occur when all participating processes are ready to engage in it. This forces an internal event to become urgent. Urgency is captured in the following transition rule by ensuring that evolution transitions are inhibited when transition rules for internal events are enabled. That is, time is prevented from progressing beyond points at which internal events are enabled. Accordingly, the evolution transition rule has a negative premiss.

$$\frac{P \xrightarrow{\mu} P'}{P \setminus A \xrightarrow{\mu} P' \setminus A} \; [\, \mu \notin A \,] \qquad \frac{P \xrightarrow{a} P'}{P \setminus A \xrightarrow{\tau} P' \setminus A} \; [\, a \in A \,]$$

$$\frac{P \overset{d}{\rightsquigarrow} P' \quad \forall a : A \bullet \neg \, \exists P' \bullet P \xrightarrow{a} P'}{P \setminus A \overset{d}{\rightsquigarrow} P' \setminus A}$$

As discussed in the next section, the set of events that are offered by a process does not change during an evolution transition. This means with respect to the above evolution rule that if no event in A is possible for P, then no event in A will be possible for P'. This ensures that the maximal progress assumption is not violated, because no internal event can become enabled during the evolution transition.

Renaming. The renaming operators allow the synchronisation events of a process to be renamed. The renaming is achieved by means of a function $f : \Sigma \to \Sigma$ satisfying $f(\checkmark) = \checkmark$, i.e., the termination event must not be renamed.

The forwardly renamed process $f(P)$ is able to perform event $f(a)$ whenever P is able to perform a. The timing behaviour is not affected. Note that the renaming function need not

be injective. If it is not injective, then several events can be mapped to a single event. This can lead to additional nondeterminisms in the renamed process.[4]

The backwardly renamed process $f^{-1}(P)$ is able to perform event a whenever P is able to perform $f(a)$. Backward renaming does not introduce any additional nondeterminism, because different events performed by P are always mapped to different events performed by $f^{-1}(P)$. Again, the timing behaviour is not affected.

Flow of Control

Sequential Composition. The sequential composition $(P;\ Q)$ starts with executing P. When P terminates successfully, which is indicated by the occurrence of the event $\checkmark$, then control passes immediately from P to Q. Urgency of $\checkmark$ in the context of sequential composition means that it occurs precisely when the first process is ready to terminate.

Interrupt. The interrupt construction $P \triangle Q$ allows P to execute, which can be interrupted at any time by Q performing an external event. In effect, Q is executed concurrently with P until either P terminates or Q performs an interrupt event.

Unlike sequential composition, in the interrupt construction both P and Q must evolve together. The process Q is also able to perform internal events without triggering the interrupt; this allows Q to offer and retract external events as time progresses.

Timed Interrupt. The timed interrupt construction $P \triangle_d Q$ allows P to execute for exactly d time units, after which the interrupt is triggered and control is passed to Q, unless P has terminated before the interrupt occurred. In contrast to the event-driven interrupt construction discussed above, process Q is not performed concurrently with P, because its state does not have any impact on triggering the interrupt.

$$\frac{P \xrightarrow{\mu} P'}{P \triangle_d Q \xrightarrow{\mu} P' \triangle_d Q}\ [\ \mu \neq \checkmark\ ] \qquad \frac{P \xrightarrow{\checkmark} P'}{P \triangle_d Q \xrightarrow{\checkmark} P'}$$

$$\frac{P \overset{d'}{\rightsquigarrow} P'}{P \triangle_d Q \overset{d'}{\rightsquigarrow} P' \triangle_{d-d'} Q}\ [\ d' \leq d\ ] \qquad \frac{}{P \triangle_0 Q \xrightarrow{\tau} Q}$$

Recursion

Recursive process definitions $N = Q$ consist of a process variable N and a timed CSP process term Q, which may contain instances of N. The recursive invocation of a process takes no time. Any delay required for recursive invocations must therefore be explicitly included. In

[4] Consider the function $f \mathrel{\widehat{=}} \{a \mapsto c, b \mapsto c\}$ and the process $P \mathrel{\widehat{=}} (a \rightarrow P_1) \,\Box\, (b \rightarrow P_2)$, which is deterministic. The process $f(P) = (c \rightarrow f(P_1)) \,\Box\, (c \rightarrow f(P_2))$, however, is not deterministic, because it is equivalent to $c \rightarrow (f(P_1) \sqcap f(P_2))$.

this way the treatment of recursion is separated from timing considerations. The recursive definition $N = Q$ does really equate N and Q.

In the mutual recursion construction $\underline{N} = \underline{Q}$, a collection of process variables $\underline{N}$ are collectively bound to a collection of process definitions $\underline{Q}$, which may refer to those process variables. Any process variable appearing in a process definition must be bound.

A.2.3 Timed Transition Systems

The transition rules of the operational semantics, which define the meaning of the process constructors, induce a timed labelled transition system describing all possible executions of timed CSP processes.

In this section, we deal with the properties of this transition system, which concern the relationships between different transitions that might appear. These properties ensure that processes behave in a natural and reasonable way. Some properties are concerned with the way processes evolve through time, and others are concerned with the relationship between evolution and event transitions. They allow us to gain a better understanding of the nature of process executions.

Evolution Transitions

The following properties concern the relationships between evolution transitions.

Time determinism: Evolution transitions are deterministic, i.e., if a process evolves through time, the result is unique. Alternative behaviours arise only because of event transitions.

$$P \overset{d}{\rightsquigarrow} P' \wedge P \overset{d}{\rightsquigarrow} P'' \Rightarrow P' \equiv P''$$

Time additivity: If process P is able to evolve for d time units resulting in process P' and if process P' is able to evolve for d' time units resulting in process P'', then P must be able to evolve for $d + d'$ time units resulting in P''. In other words, the presence of an intermediate process state P' does not influence the process state that is finally reached.

$$P \overset{d}{\rightsquigarrow} P' \wedge P' \overset{d'}{\rightsquigarrow} P'' \Rightarrow P \overset{d+d'}{\rightsquigarrow} P''$$

Time interpolation: Conversely, if process P is able to evolve for $d + d'$ time units resulting in process P'', then it must have an intermediate process state P' at time d.

$$P \overset{d+d'}{\rightsquigarrow} P'' \Rightarrow \exists P' \bullet P \overset{d}{\rightsquigarrow} P' \wedge P' \overset{d'}{\rightsquigarrow} P''$$

Time closure: If process P is able to evolve for all durations less than d, it must also be able to evolve for d time units.

$$(\forall d' : \mathbb{R}_0^+ \mid d' < d \bullet \exists P' \bullet P \overset{d'}{\rightsquigarrow} P') \Rightarrow (\exists P' \bullet P \overset{d}{\rightsquigarrow} P')$$

An infinite sequence of evolution transitions can be described by an infinite sequence of processes $\{P_i\}_{i\in\mathbb{N}}$ and durations $\{d_i\}_{i\in\mathbb{N}}$. Such a sequence is called Zeno sequence, if the sum of all durations d_i is finite:

$$\sum_{i=0}^{\infty} d_i = d < \infty.$$

Such a sequence indicates that process P is able to evolve for all durations less than d time units. The time closure property allows us to deduce that P is also able to evolve for d time units. That is, all Zeno sequences have a limit process.

Together, the properties of time determinism and time interpolation ensure that each evolution transition passes through a unique continuum of process states.

Executions

An execution of a process is described in terms of an alternating sequence of process states and transitions, where a transition may be an event transition or an evolution transition. Each transition in an execution relates the two process states that are adjacent.

The properties of evolution transitions discussed above allow one to record evolution transitions more economically. First, time additivity allows one to collapse any adjacent pair of evolution transitions $P \overset{d}{\rightsquigarrow} P'$ and $P' \overset{d'}{\rightsquigarrow} P''$ to a single evolution transition $P \overset{d+d'}{\rightsquigarrow} P''$. Second, an infinite sequence of evolution transitions with initial process P can be collapsed in a similar way by exploiting the time closure property. If the sum of the associated durations is finite, then the infinite sequence can be collapsed into a single evolution transition $P \overset{d}{\rightsquigarrow} P'$; otherwise, if the sum is infinite, it can be collapsed into the evolution transition $P \overset{\infty}{\rightsquigarrow}$.

Altogether, the properties of time additivity and time closure allow one to represent each process execution as an alternating sequence of process states and transition labels that meets the condition that an evolution transition must not be followed by another evolution transition.

An execution may be finite or infinite. It is called *maximal*, if it cannot be extended by further transitions. An infinite execution must contain an infinite number of event transitions interspersed with evolution transitions. A finite maximal execution either ends with a process that cannot undergo an event or evolution transition; or it ends with the evolution transition $\overset{\infty}{\rightsquigarrow}$.

Process executions have further useful properties, which concern the relationships between event transitions the process can perform and the evolution transitions it can undergo.

Urgency of internal events: As a consequence of the maximal progress assumption, internal events occur as soon as they are enabled. So, if a process can perform an internal event it is not able to evolve for any duration.

$$P \overset{\tau}{\rightarrow} \;\Rightarrow (\forall\, d : \mathbb{R}_0^+ \bullet \neg\, P \overset{d}{\rightsquigarrow})$$

This represents the mechanism of the operational semantics to treat internal events as urgent.

Constancy of offers: The set of external event transitions that are enabled does not change when time progresses, i.e., when a process undergoes an evolution transition. In other words, when P evolves to P', then all events that P can perform must also be possible for P', and conversely, all events that P' can perform must have been possible for P.

$$P \overset{d}{\rightsquigarrow} P' \Rightarrow (\forall a : \Sigma^{\checkmark} \bullet P \overset{a}{\rightarrow} \Leftrightarrow P' \overset{a}{\rightarrow})$$

Any change of the set of events offered by a process must be accompanied by an event transition.

Unrealistic behaviours: In the world of implementations, Newtonian time progresses at a constant rate and cannot be blocked. The blockage of time is only a mechanism used by the operational semantics in order to express the urgency of internal events. However, it does not represent a reasonable behaviour in its own right, because it contradicts the unstoppable nature of time.

For this reason, any maximal execution, which is to represent the behaviour of a real process, must have an infinite duration, i.e., it must be possible to observe the process forever. A maximal execution of a process that does not progress beyond a particular point of time is not consistent with Newtonian time. There are different kinds of maximal executions with finite duration.

Timestop: A maximal execution can end with a process that is neither able to perform an internal or external event neither to undergo an evolution. Such a process is termed *timestop*, because it blocks time from progressing.

Spin: A maximal execution can have infinitely many transitions but only finitely many evolution transitions. In such a sequence, all evolution transitions must appear within an initial segment; the remaining segment contains only event transitions. These event transitions occur at the same instant of time. Such an execution is called *spin execution*.

Zeno: A maximal execution can consist of infinitely many evolution transitions; but these could sum only to a finite duration. Such an execution is termed *Zeno execution*, because it approaches a time without reaching it.

All of these kinds of unrealistic behaviours can arise within the timed transition system induced by the operational semantics; however, only in the context of recursive process equations that are not time-guarded, see the next section.

Well-timed Processes

All of the finite duration maximal executions arise in the context of recursive process equations $N = Q$ in which there is no sufficient delay between the time a process starts executing and the time it is invoked recursively. When such a recursive invocation occurs immediately, timestops and spin executions may be caused; recursive invocations with ever reducing delays may give rise to Zeno executions. These unrealistic behaviours are all prevented by introducing a minimum positive delay before the time it is invoked recursively.

A process term is *t-guarded* if any execution of that process term cannot reach a recursive invocation in less than t time units. A process term is *time-guarded* if there exists a $t > 0$ such that it is t-guarded.

A timed CSP process is *well-timed* if all associated single and mutual recursive definitions are time-guarded. Well-timed processes can never exhibit timestops, spin executions and Zeno executions. This means that if a process is well-timed then all of its maximal executions have infinite duration.

Finally, the executions of well-timed processes satisfy the property of *finite variability*. It states that any finite time interval of an execution should contain only finitely many events. Expressed the other way around: infinitely many events take infinitely long to occur.

A.2.4 Denotational Semantics

Processes in timed CSP are considered as interacting system components. That is, processes are judged only in terms of their behaviour at the external interface; their internal implementation is not relevant. Thus, two processes that cannot be distinguished by any context, into which they could be placed, are considered equal. In a timed setting, the context into which a process is placed can be time-sensitive: it can make the offer of events dependent on the progress of time.

The *timed failures model* is a semantic model for *well-timed* processes: it associates each process with a set of timed failures. A *timed failure* is a record of a process execution, consisting of a timed trace and a timed refusal.

The timed failures model is a compositional model; it formalises those aspects of the behaviour of a process that can be observed when interacting with it, and it abstracts from the other aspects.

Timed Observations

When a process executes, it performs events at its external interface. A *timed trace* is a record of performed events together with the times at which they were performed relative to the starting point of the process execution. Internal events are not recorded in a timed trace, because they are not visible to the environment.

A *timed event* is drawn from the set $\mathbb{R}_0^+ \times \Sigma$, consisting of a time and an event. A timed trace is a sequence of timed events in which the times are non-decreasing: the events are recorded in temporal order.[5]

[5] Note that even events that occur at the same time instant are arranged in sequence, i.e., a temporal order between such events is fixed in a timed trace.

On the one hand, this might appear counter-intuitive, because a causal relationship between simultaneous events is suggested. On the other hand, there is an issue of compatibility between (untimed) CSP and timed CSP: process expressions are used in both in order to define the temporal ordering of events. The processes

$P \mathrel{\widehat{=}} a \rightarrow b \rightarrow Stop$

$Q \mathrel{\widehat{=}} b \rightarrow a \rightarrow Stop$

e.g., define two temporal orderings of the events a and b, which are contradictory. Thus, if the two are com-

A timed trace may be a record of a finite or infinite duration execution. If it corresponds to a finite duration execution, then its length must be finite; this is dictated by finite variability. Otherwise, it may be finite, if the execution contains only finitely many external events, or it may be infinite. In the latter case, the times in the timed trace are not bounded.

Schneider [1999a] defined several functions on timed traces. They are treated in Appendix E.

The timed traces associated with a process can be determined from its executions. An execution, as introduced in Section A.2.3, is an alternating sequence of process states and transitions, so it contains information about events that are performed and their associated times. The timed traces associated with a process are those that are associated with its executions.

Processes interact with each other by synchronising on events at particular times. When a process is offered the opportunity to perform an event, it can make one of two possible responses. It can either perform the event, or else it can refuse to perform it. The timed trace is a record of the timed events performed during an execution, whereas the timed refusal is a record of which timed events were refused.

In the timed setting of timed CSP, event refusals must be considered at particular times and not only at the end of the execution: refusal information is available throughout an execution.[6] The refusal of an event a at time t reflects the fact that a is not possible in the process state reached at t. If the execution records the performance of several events at time t, passing through several states, then the refusal information is associated with the last of these states; the refusal information is subsequent to all events performed at that time instant. Accordingly, the states of an execution associated with refusal information are those states that are followed by an evolution.[7]

A timed refusal $\aleph$ is a set of timed events. Since refusals accompany evolutions, and event offers are constant over evolutions, the same set of events will be refused throughout an evolution. Refusal sets can therefore be structured such that the set of refused events is finitely variable with respect to time. This means that a refusal set will record intervals over which sets of events are refused. The appropriate intervals are half-open, corresponding to the stable states along evolutions.

Schneider [1999a] defined several functions on timed refusals. They are described in Appendix E.

---

posed in parallel, the result should be deadlock.

$P \parallel Q \equiv Stop$

If, however, the process $P \parallel Q$ is considered in the timed setting of timed CSP and if there would be no temporal order between events occurring at the same time, then the above equation would not be true, because P and Q are both able to perform a and b simultaneously, so their parallel composition would also have this ability.

This trade-off between intuition and the need to be compatible with CSP was resolved in timed CSP in favour of compatibility.

[6] This is a contrast to the failures–divergences model of CSP, in which refusal information is available only for the end of an execution.

[7] Note that an environment can offer a process several instances of a single event at a single time instant. It is thus possible that an event is recorded for a particular time instant in the timed trace *and* in the timed refusal. This means that the process performed as many instances of the event as recorded in the timed trace, but refused to perform any further instance as witnessed by the membership of this event in the timed refusal.

A timed refusal $\aleph$ is said to be *consistent* with an execution if every timed event (t, a) in the timed refusal can be refused during the execution. This means that an a transition is not possible for the particular process state reached at time t during the execution. Several timed refusals will be consistent with any particular execution. In particular, the empty set will vacuously be consistent with any execution. Every execution will have a unique maximal timed refusal consistent with it, containing all possible refused events for the duration of the execution. A set $\aleph$ will be consistent with an execution if, and only if, it is a subset of the maximal timed refusal.

Processes are considered in terms of the possible observations that can be made of their executions. These will be described in terms of timed traces and timed refusals. On the one hand, every execution is associated with exactly one timed trace. On the other hand, the events observed to be refused during the execution are not explicit in the transition. Possible timed refusals are those that are consistent with the execution, but the observation of a timed refusal depends on what was offered to the process during the execution: if nothing was offered, then no events will be observed to be refused.

An observation of an execution is a timed failure $(s, \aleph)$ consisting of a timed trace s and a timed refusal $\aleph$. The timed trace s is the sequence of events occurring in the execution, and the timed refusal $\aleph$ is some consistent set of timed events which could be refused during the execution. A natural way to understand the pair $(s, \aleph)$ as an observation of the process is as evidence that it is able to perform the sequence of events s, and to refuse all events in the set $\aleph$ if offered them while performing s. In general, it is a partial record of an execution.

A dual way to consider the pair $(s, \aleph)$ is from the point of view of the environment. The environment offers events at particular time instants: these offers are either accepted and appear in s, or else they are refused and appear in $\aleph$. The timed failure thus provides a record of all events that were offered to the process during the observation, indicating whether they were accepted or refused. It may be considered as an experiment conducted by the environment, offering particular events and eliciting responses.

Timed Failures Model

On the one hand, any timed CSP process is associated with a set of timed failures. On the other hand, not every arbitrary set of timed failures represents a timed CSP process: there are three properties (TF1–TF3) that must hold for any set of timed failures corresponding to some timed CSP process. The timed failures model consists of all sets S of timed failures that meet these properties.

Firstly, the empty timed failure $(\langle\rangle, \varnothing)$ must be a possible observation of any process (property TF1).

Secondly, any set of timed failures corresponding to a timed CSP process must be downwards closed with respect to the information order[8] (property TF2). That is, for each timed failure contained in S, the set must also contain all timed failures that capture its trace and refusal information only partially, i.e., up to a particular time instant.

[8] Schneider defined the *information order* $\preceq$ on timed failures as follows:

$$(s', \aleph') \preceq (s, \aleph) \Leftrightarrow \exists s'' \bullet s = s' \frown s'' \wedge \aleph' \subseteq \aleph \parallel\!\upharpoonright_{tref} \text{begin}_{ttr}(s'').$$

Finally, any timed failure $(s, \aleph)$ is associated with a complete timed refusal: the timed refusal $\aleph$ observed can be augmented with further refusal information to obtain a complete timed refusal $\aleph' \supseteq \aleph$ (property TF3). Completeness of $\aleph'$ means that for any time t, any event a not appearing in $\aleph'$ must be possible for the process at that time. This manifests itself in two ways:

C1: Any timed event (t, a) that does not appear in the timed refusal $\aleph'$ must provide a possible alternative continuation to the timed trace. Refusal information at t is subsequent to the timed trace up to and including that time, so the timed event (t, a) follows the timed trace up to and including t.

C2: If an event a is not refused as time t is approached, then it is possible throughout the evolution leading up to time t, so a must also be possible at time t. This is a consequence of the constancy of event offers and the time closure for evolutions.[9]

These considerations are formalised as follows:

$$(\langle\rangle, \varnothing) \in S \tag{TF1}$$

$$\begin{aligned}&(\forall\, s, s' : \mathit{TimedTrace};\ \aleph, \aleph' : \mathit{TimedRefusal} \bullet\\ &\quad (s, \aleph) \in S \wedge (s', \aleph') \preceq (s, \aleph) \Rightarrow (s', \aleph') \in S)\end{aligned} \tag{TF2}$$

$$\begin{aligned}&(\forall\, s : \mathit{TimedTrace};\ \aleph : \mathit{TimedRefusal} \mid (s, \aleph) \in S \bullet &&\text{(TF3)}\\ &\quad (\exists\, \aleph' : \mathit{TimedRefusal} \mid \aleph \subseteq \aleph' \wedge (s, \aleph') \in S \bullet\\ &\qquad (\forall\, t : \mathbb{R}_0^+;\ a : \mathit{Event} \bullet\\ &\qquad\quad ((t, a) \notin \aleph' \Rightarrow ((s \upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \Vert\!\upharpoonright_{tref} t) \in S) &&\text{(C1)}\\ &\qquad\quad \wedge\\ &\qquad\quad t > 0 \wedge \neg\, (\exists\, \epsilon : \mathbb{R} \setminus \{0\} \bullet [t - \epsilon, t) \times \{a\} \subseteq \aleph')\\ &\qquad\qquad \Rightarrow ((s \Vert\!\upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \Vert\!\upharpoonright_{tref} t) \in S)))\} &&\text{(C2)}\end{aligned}$$

There is a further property TF4, which holds only for those sets of timed failures representing timed CSP processes that do not introduce infinite nondeterminism. It states that the infinite duration timed failures of such a process are exactly those that can be deduced from its finite duration timed failures. In other words, the time closure of a set of timed failures associated with such a process must be equal to itself.

$$\overline{S} \mathrel{\widehat=} \{(s, \aleph) \mid \forall\, t : \mathbb{R}_0^+ \bullet (s \Vert\!\upharpoonright_{ttr} t, \aleph \Vert\!\upharpoonright_{tref} t) \in S\} = S \tag{TF4}$$

[9] Case C2 is necessary because case C1 does not cover so-called point nondeterminisms. Consider the process

$(a \to \mathit{Stop}) \rhd \{d\}\ \mathit{Stop}$.

At time d, and only at time d, its behaviour is nondeterministic. If the environment offers the event a at time d after not having offered it in the interval $[0, d)$, then the process may perform a or it may refuse to perform it, depending on whether control has already switched to *Stop*. Therefore, (d, a) is a member of the maximal timed refusal, and it is impossible to use C1 in order to deduce that a is a possible continuation to the timed trace.

However, since we know that a cannot be refused during the interval $[0, d)$, we can use C2 in order to make this deduction.

Timed Failures Semantics

The denotational semantics of timed CSP is *compositional*: the timed failures associated with a compound process are completely determined by the timed failures associated with its component processes and the operator composing the component processes. Accordingly, there is a single clause for each operator of the syntax of timed CSP.

Deadlock. The process *Stop* never engages in any event, i.e., it can always refuse any set of events offered.

$$timedfailures\,[\![Stop]\!] = \{(\langle\rangle, \aleph) \mid \aleph \in TimedRefusal\}$$

Event Prefix. The timed failures associated with the process $a \rightarrow P$ fall into two classes. On the one hand, the environment might never offer a, so it will never occur. In this case, a will never appear in $\aleph$. On the other hand, a may be offered by the environment at time t and will appear at the beginning of the timed trace. In this case, a could not have been offered before t. The behaviour subsequent to the occurrence of a is that of P. These two cases are reflected by the two sets in the following definition.

$$\begin{aligned} timedfailures\,[\![a \rightarrow P]\!] = {} & \{(\langle\rangle, \aleph) \mid a \notin \sigma_{tref}(\aleph)\} \\ & \cup \\ & \{(\langle(t, a)\rangle \frown (s +_{ttr} t), \aleph) \mid t \in \mathbb{R}_0^+ \\ & \qquad \wedge a \notin \sigma_{tref}(\aleph \restriction_{tref} t) \\ & \qquad \wedge (s, \aleph \dot{-}_{tref} t) \in timedfailures\,[\![P]\!]\} \end{aligned}$$

Timed Event Prefix. The semantic clause of the timed event prefix operator is similar to that of the previous operator. The only difference is the case in which event a occurs at time t. Then, the subsequent behaviour is that of P, in which the time variable u is substituted by t ($P[t/u]$).

$$\begin{aligned} timedfailures\,[\![a@u \rightarrow P]\!] = {} & \{(\langle\rangle, \aleph) \mid a \notin \sigma_{tref}(\aleph)\} \\ & \cup \\ & \{(\langle(t, a)\rangle \frown (s +_{ttr} t), \aleph) \mid t \in \mathbb{R}_0^+ \\ & \qquad \wedge a \notin \sigma_{tref}(\aleph \restriction_{tref} t) \\ & \qquad \wedge (s, \aleph \dot{-}_{ttr} t) \in timedfailures\,[\![P[t/u]]\!]\} \end{aligned}$$

Prefix Choice. Again, the timed failures associated with the process $x : A \rightarrow P(x)$ fall into two categories. Either the environment will never offer an event in A, or an event $a \in A$ occurs at time t. In the latter case, the subsequent behaviour is due to process $P(a)$ shifted through t time units.[10]

[10] In [Schneider, 1999a], this operator seems not to be defined correctly.

$$\begin{aligned} \mathit{timed\,failures}\,[\![x : A \rightarrow P(x)]\!] = \; & \{(\langle\rangle, \aleph) \mid A \cap \sigma_{tref}(\aleph) = \varnothing\} \\ & \cup \\ & \{(\langle(t,a)\rangle \frown (s +_{ttr} t), \aleph) \mid t \in \mathbb{R}_0^+ \wedge a \in A \\ & \quad \wedge A \cap \sigma_{tref}(\aleph \parallel\!\restriction_{tref} t) = \varnothing \\ & \quad \wedge (s, \aleph \mathbin{\dot{-}}_{tref} t) \in \mathit{timed\,failures}\,[\![P(a)]\!]\} \end{aligned}$$

Successful Termination. The process *Skip* is immediately ready to accept the termination event ✓ before it deadlocks. Its semantic clause is analogous to that of the event prefix operator with process P substituted by *Stop*.

$$\begin{aligned} \mathit{timed\,failures}\,[\![Skip]\!] = \; & \{(\langle\rangle, \aleph) \mid \checkmark \notin \sigma_{tref}(\aleph)\} \\ & \cup \\ & \{(\langle(t, \checkmark)\rangle, \aleph) \mid t \in \mathbb{R}_0^+ \wedge \checkmark \notin \sigma_{tref}(\aleph \parallel\!\restriction_{tref} t)\} \end{aligned}$$

Timeout. The time-sensitive choice $P \triangleright \{d\} Q$ is resolved in favour of P if the first event occurs before time t; otherwise it is resolved in favour of Q. Any behaviour of P that records an event occurrence before time d is also a behaviour of $P \triangleright \{d\} Q$. The other behaviours are those in which the timeout occurs. In this case, the timed refusal $\aleph$ restricted to the first d units must be consistent with P, and the behaviour after d is due to Q, shifted through d time units.

$$\begin{aligned} \mathit{timed\,failures}\,[\![P \triangleright \{d\} Q]\!] = \; & \{(s, \aleph) \mid \mathrm{begin}_{ttr}(s) \leqslant_{\mathbb{R}} d \wedge (s, \aleph) \in \mathit{timed\,failures}\,[\![P]\!]\} \\ & \cup \\ & \{(s, \aleph) \mid \mathrm{begin}_{ttr}(s) \geqslant_{\mathbb{R}} d \wedge (\langle\rangle, \aleph \parallel\!\restriction_{tref} d) \in \mathit{timed\,failures}\,[\![P]\!] \\ & \quad \wedge (s, \aleph) \mathbin{\dot{-}}_{tfail} d \in \mathit{timed\,failures}\,[\![Q]\!]\} \end{aligned}$$

The fact that $\mathrm{begin}_{ttr}(s) \leqslant_{\mathbb{R}} d$ and $\mathrm{begin}_{ttr}(s) \geqslant_{\mathbb{R}} d$ are true if $\mathrm{begin}_{ttr}(s) = d$ embodies the point nondeterminism at time d.

Delay. The process *Wait* d is a derived construct defined as *Stop* $\triangleright\{d\}$ *Skip*, so its semantics can be deduced from these three operators.

External Choice. The timed failures associated with the external choice $P \;\Box\; Q$ fall into three classes. The choice may have been resolved in favour of P, it may have been resolved in favour of Q or it may not have been resolved yet. In the last case, the timed trace is empty and the timed refusal is the joint responsibility of P and Q. In the first two cases, the two processes must have participated in the timed refusal up to the occurrence of the first event, resolving the choice.

$$\begin{aligned} \mathit{timed\,failures}\,[\![P \;\Box\; Q]\!] = \{(s, \aleph) \mid \; & (s, \aleph) \in \mathit{timed\,failures}\,[\![P]\!] \cup \mathit{timed\,failures}\,[\![Q]\!] \\ & \wedge \\ & (\langle\rangle, \aleph \parallel\!\restriction_{tref} \mathrm{begin}_{ttr}(s)) \in \mathit{timed\,failures}\,[\![P]\!] \\ & \qquad \cap \mathit{timed\,failures}\,[\![Q]\!]\} \end{aligned}$$

Internal Choice. The timed observations associated with an internal choice are those that are associated with either of its constituent processes.

$$\mathit{timed\,failures}\,[\![P \sqcap Q]\!] = \mathit{timed\,failures}\,[\![P]\!] \cup \mathit{timed\,failures}\,[\![Q]\!]$$

Indexed Internal Choice. The indexed internal choice $\sqcap_{i \in J} P_i$ can behave as any of its constituent processes. Its timed failures are hence all of those timed failures associated with any of its constituents.

$$\mathit{timed\,failures}\,[\![\sqcap_{i \in J} P_i]\!] = \bigcup_{i \in J} \mathit{timed\,failures}\,[\![P_i]\!]$$

Alphabetised Parallel. In the parallel composition $P \,|[\, A \mid B \,]|\, Q$, the events in $A^{\checkmark}$ can occur only with the participation of P, and the events in $B^{\checkmark}$ can occur only with the participation of Q. Thus, the constituent processes must agree on the events in $(A \cap B)^{\checkmark}$, or, conversely, if either of them refuses an event in this set, then the whole process refuses it. The combined process can engage only in events in $(A \cup B)^{\checkmark}$, so the timed refusal outside $(A \cup B)^{\checkmark}$ is arbitrary.

$$\begin{aligned}\mathit{timed\,failures}\,[\![P \,|[\, A \mid B \,]|\, Q]\!] = \{(s, \aleph) \mid \exists\, \aleph_1, \aleph_2 \bullet\ & \\ & \aleph \upharpoonright_{tref} (A \cup B)^{\checkmark} = (\aleph_1 \upharpoonright_{tref} A^{\checkmark}) \cup (\aleph_2 \upharpoonright_{tref} B^{\checkmark}) \\ & \wedge\, s = s \upharpoonright_{ttr} (A \cup B)^{\checkmark} \\ & \wedge\, (s \upharpoonright_{ttr} A^{\checkmark}, \aleph_1) \in \mathit{timed\,failures}\,[\![P]\!] \\ & \wedge\, (s \upharpoonright_{ttr} B^{\checkmark}, \aleph_2) \in \mathit{timed\,failures}\,[\![Q]\!]\}\end{aligned}$$

Interleaving. The interleaving $P \mid\mid\mid Q$ can engage in any event (except $\checkmark$) precisely when one of the component processes can engage in that event. Only termination requires the synchronisation of P and Q. Therefore, a timed refusal of the combined process must be a timed refusal of both components, and the timed trace s is an interleaving of the component timed traces.

$$\begin{aligned}\mathit{timed\,failures}\,[\![P \mid\mid\mid Q]\!] = \{(s, \aleph_1 \cup \aleph_2) \mid \exists\, s_1, s_2 \bullet\ & s \text{ interleaves } (s_1, s_2) \\ & \wedge\, (s_1, \aleph_1) \in \mathit{timed\,failures}\,[\![P]\!] \\ & \wedge\, (s_2, \aleph_2) \in \mathit{timed\,failures}\,[\![Q]\!] \\ & \wedge\, \aleph_1 \upharpoonright_{tref} \Sigma = \aleph_2 \upharpoonright_{tref} \Sigma\}\end{aligned}$$

Interface Parallel. The interface parallel operator $P \,|[\, A \,]|\, Q$ combines the features of the alphabetised parallel and the interleaving operator. It requires the synchronisation on events of the interface set A and allows its component processes to interleave on events outside this interface set. That is, events within A can be refused by either process, but events outside A can be refused only if both component processes do so.

$$\begin{aligned}\mathit{timed\,failures}\,[\![P \,|[\, A \,]|\, Q]\!] = \{(s, \aleph_1 \cup \aleph_2) \mid \exists\, s_1, s_2 \bullet\ & s \text{ synch}_A\ (s_1, s_2) \\ & \wedge\, \aleph_1 \setminus_{tref} A^{\checkmark} = \aleph_2 \setminus_{tref} A^{\checkmark} \\ & \wedge\, (s_1, \aleph_1) \in \mathit{timed\,failures}\,[\![P]\!] \\ & \wedge\, (s_2, \aleph_2) \in \mathit{timed\,failures}\,[\![Q]\!]\}\end{aligned}$$

Hiding. When hiding a set of events from the interface of a process, the process is given the full control over the occurrence of these events. Maximal progress dictates that such an event occurs as soon as the process is ready to accept it.

The timed failures of the process $P \setminus A$ are determined by considering those timed failures of P in which the events in A are continually refused, meaning that the environment of the process is always ready to offer more instances of events in A than P is able to accept. In other words, these timed failures represent observations in which events in A occur as soon as they are accepted by P. The hiding makes the events in A internal, so they do not appear in the resulting timed trace.

$$\mathit{timedfailures}\,[\![P \setminus A]\!] = \{(s \setminus_{ttr} A, \aleph) \mid (s, \aleph \cup ([0,\infty) \times A)) \in \mathit{timedfailures}\,[\![P]\!]\}$$

Renaming. The forward renamed process $f(P)$ maps each performed event a to $f(a)$. It is able to refuse an event a if P is able to refuse all events that are mapped to a (f need not be injective).

$$\mathit{timedfailures}\,[\![f(P)]\!] = \{(f(s), \aleph) \mid (s, f^{-1}(\aleph)) \in \mathit{timedfailures}\,[\![P]\!]\}$$

In the above equation, $f(s)$ denotes the timed trace obtained by applying f to all elements of s individually. Moreover,

$$f^{-1}(\aleph) = \{(t,a) \mid (t, f(a)) \in \aleph\}$$
$$f(\aleph) = \{(t, f(a)) \mid (t,a) \in \aleph\}.$$

The backward renamed process $f^{-1}(P)$ can perform the event a when P can perform $f(a)$, and it can refuse a whenever P can refuse $f(a)$.

$$\mathit{timedfailures}\,[\![f^{-1}(P)]\!] = \{(s, \aleph) \mid (f(s), f(\aleph)) \in \mathit{timedfailures}\,[\![P]\!]\}$$

Sequential Composition. In the sequential composition $(P;\ Q)$, P is executed until it terminates at which point control is immediately passed to Q. The termination of P is completely under the control of the sequential composition, i.e., P terminates as soon as it is ready to do so. This is reflected in the following definition by considering only particular timed failures associated with P, analogous to the definition of the hiding operator.

A timed failure of $(P;\ Q)$ either corresponds to a non-terminating behaviour of P, or it corresponds to a terminating behaviour of P immediately followed by a behaviour of Q. The termination of P, if any, does not appear in the resulting timed trace, because it does not represent the termination of the sequential composition.

$$\begin{aligned}\mathit{timedfailures}\,[\![P;\ Q]\!] = \;& \{(s, \aleph) \mid \checkmark \notin \sigma_{ttr}(s) \wedge (s, \aleph \cup [0,\infty) \times \{\checkmark\}) \in \mathit{timedfailures}\,[\![P]\!]\} \\ & \cup \\ & \{(s_1 \frown s_2, \aleph) \mid \exists\, t \bullet \checkmark \notin \sigma_{ttr}(s_1) \\ & \qquad \wedge (s_2, \aleph) \mathbin{\dot{-}}_{tfail} t \in \mathit{timedfailures}\,[\![Q]\!] \\ & \qquad \wedge (s_1 \frown \langle (t, \checkmark) \rangle, \aleph \upharpoonright_{tref} t \cup ([0,t) \times \{\checkmark\})) \in \mathit{timedfailures}\,[\![P]\!]\}\end{aligned}$$

Interrupt. The process $P \triangle Q$ starts by executing P. Simultaneously, initial events of Q are enabled; and as soon as one of these occurs, P is interrupted and control is passed to Q.

A timed failure associated with $P \triangle Q$ either corresponds to an uninterrupted behaviour of P, or it corresponds to an interrupted behaviour of P followed by a behaviour Q. The fact that Q is enabled while P is executing means that the timed refusal up to an interrupt must be a joint refusal of P and Q. After an interrupt, the timed failure is due to Q.

$$\mathit{timedfailures}\,[\![P \triangle Q]\!] = \{(s_1 \frown s_2, \aleph) \mid (s_1, \aleph \parallel\!\upharpoonright_{tref} \text{begin}_{ttr}(s_2)) \in \mathit{timedfailures}\,[\![P]\!] \\ \wedge\ (s_2, \aleph \parallel\!\upharpoonright_{tref} \text{begin}_{ttr}(s_1 \upharpoonright_{ttr} \{\checkmark\})) \in \mathit{timedfailures}\,[\![Q]\!]\}$$

Note that the notation $s_1 \frown s_2$ implies that the catenation is a valid timed trace. This has two consequences. First, the times recorded in s_2 must be greater than or equal to all times recorded in s_1, ensuring that P stops performing events as soon as Q has started. Second, unless s_2 is empty, s_1 must not contain the termination event, because in a timed trace it can appear only as the last event. This ensures that the whole process stops when P terminates.

Timed Interrupt. The timed interrupt $P \triangle_d Q$ first executes P for d time units, and then passes control to Q unless P has already terminated.

The timed failures associated with $P \triangle_d Q$ consist of two parts: the initial part corresponding to the execution of P for d time units, and the remaining part corresponding to the execution of Q beginning at time d.

$$\mathit{timedfailures}\,[\![P \triangle_d Q]\!] = \{(s_1 \frown (s_2 +_{ttr} d), \aleph) \mid \\ \text{end}_{ttr}(s_1) \leq d \\ \wedge\ (s_1, \aleph \parallel\!\upharpoonright_{tref} d) \in \mathit{timedfailures}\,[\![P]\!] \\ \wedge\ (\checkmark \notin \sigma_{ttr}(s_1) \Rightarrow (s_2, \aleph \dot{-}_{tref} d) \in \mathit{timedfailures}\,[\![Q]\!])\}$$

Again, the fact that $s_1 \frown (s_2 +_{ttr} d)$ must be a valid timed trace implies that s_2 must be empty if $\checkmark \in \sigma_{ttr}(s_1)$.

Recursion. In the dialect of timed CSP [Schneider, 1999a] that we use in RT-Z, there is no explicit process operator for defining recursion.[11] Processes with recurring behaviour are defined by recursive equations

$$N = Q.$$

The above equation identifies the process variable N and the process term Q, which can contain references to N. In this case, N is defined in terms of itself.

To solve recursive process equations, a structure is imposed on the timed failures model based on the refinement order and the theory of metric spaces.

The refinement order relates processes in terms of the behaviours they can exhibit. The more behaviours a process can exhibit, the more nondeterministic it is. Process P is less

[11] This is in contrast to the older dialect described in [Davies, 1993], which introduces the fixed point operator μ.

deterministic than process Q (P is refined by Q), written $P \sqsubseteq Q$, if the set of behaviours associated with Q is a subset of that associated with P.

$$P \sqsubseteq Q \Leftrightarrow \mathit{timedfailures}\,[\![Q]\!] \subseteq \mathit{timedfailures}\,[\![P]\!]$$

In the timed failures model, the most nondeterministic process corresponds to the set of all timed failures, and the most deterministic processes are the maximal processes under the nondeterminism partial order (i.e., processes whose nondeterminism cannot be further reduced).

Any well-timed CSP equation $N = Q$ has a *least* solution (fixed point) under the nondeterminism order. It is considered as the natural solution, because all other solutions (fixed points) refine it: the least solution contains less restrictions (design decisions) than all other solutions. Schneider [1999a] has claimed that the least fixed point computed in the timed failures model coincides with the failure information that can be extracted from the operational semantics.

The partial order imposed by the nondeterminism order is combined with a metric space structure in order to ensure the existence and uniqueness of fixed points for well-timed CSP processes. The metric space structure is based on a distance between processes, which has the property that the longer it takes to tell two processes apart, the closer together they are.

To define the distance function, the projection of a process S (more precisely, its associated set of timed failures) to the time interval ending at time t is introduced as follows.

$$S \Vert\!\upharpoonright_{tcsp} t = \{(s, \aleph) \mid (s, \aleph) \in S \wedge s \Vert\!\upharpoonright_{ttr} t = s \wedge \aleph \Vert\!\upharpoonright_{tref} t = \aleph\}$$

This projection extracts all timed failures that contain trace or refusal information beyond time t. The case $t = \infty$ is treated separately.

$$S \Vert\!\upharpoonright_{tcsp} \infty = \{(s, \aleph) \mid (s, \aleph) \in S \wedge \exists\, t : \mathbb{R}_0^+ \bullet \mathrm{end}_{tfail}(s, \aleph) = t\}$$

On this basis, the distance function on processes is defined as follows.

$$d(S, T) = \inf\{2^{-t} \mid t \in \mathbb{R}_0^+ \wedge S \Vert\!\upharpoonright_{tcsp} t = T \Vert\!\upharpoonright_{tcsp} t\}$$

If $S \Vert\!\upharpoonright_{tcsp} t = T \Vert\!\upharpoonright_{tcsp} t$, then S and T are indistinguishable up to time t. Any observer who wishes to tell S and T apart must wait at least t time units. The distance between two processes can be at most 1. If the distance between two processes is 0, then they must have the same set of *finite duration* timed failures. In this case, if the processes are not identical, then they must be associated with different sets of infinite duration timed failures.

Schneider [1999a] introduced the finite timed failures model (FTF) in order to make available the theory of metric spaces, which provides powerful results concerning the existence and *uniqueness* of fixed points. The FTF model identifies each process with the set of finite duration timed failures associated with it. It is the projection of the timed failures model onto finite duration behaviours. Any two processes P and Q satisfying $d(P, Q) = 0$ correspond to the same element in the FTF model, and any two distinct elements will have some strictly positive distance. As Schneider pointed out, the finite timed failures model together with the distance function d forms a *complete* metric space.[12]

[12] The (general) timed failures model together with the distance function does not form a metric space. The

The following definitions and theorem are drawn from [Sutherland, 1975].

Definition A.1
A metric space $M = (A, d)$ *consists of a non-empty set* A *together with a map* $d : A \times A \to \mathbb{R}$ *satisfying*

- $d(x, y) \geq 0 \wedge (d(x, y) = 0 \Leftrightarrow x = y)$
- $d(x, y) = d(y, x)$
- $d(x, y) + d(y, z) \geq d(x, z)$

for all x, y, z *in* A.

Definition A.2
A metric space M *is* complete *if every Cauchy sequence in* A *converges to a point in* A.

Definition A.3
A map $f : A \to A$ *of a metric space* M *with metric* d *is a* contraction *if there exists a constant* $K < 1$ *such that* $d(f(x), f(y)) \leq K * d(x, y)$ *for all* x, y *in* A.

The key result is the following theorem, which states that any contraction in a complete metric space must have a unique fixed point.

Theorem A.1 (Banach Fixed Point Theorem)
If $f : A \to A$ *is a contraction of a complete metric space* $M = (A, d)$, *then* f *has a* unique *fixed point in* A.

Assuming that a contraction f has different fixed points

$$f(x) = x \neq x' = f(x')$$

leads to a contradiction, because the distance between $f(x)$ and $f'(x)$ must be less than the distance between x and x'. Thus, the application of f to x and x' must change at least one of them; this, however, is in contrast to the property of a fixed point.

Furthermore, the unique fixed point can be determined by choosing an arbitrary member of A, say x, and by considering the sequence

$$\langle x, f(x), f^2(x), f^3(x), \ldots \rangle.$$

---

distance between the processes

$$P \mathrel{\hat{=}} \bigsqcap_{t \in \mathbb{R}_0^+} Wait\ t;\ a \to Stop \qquad \text{and} \qquad Q \mathrel{\hat{=}} Stop \sqcap P$$

e.g., is 0, because they are associated with the same set of finite duration timed failures. However, they are not identical, because Q is able to refuse a forever, whereas P is not able to do this; the infinite duration timed failures associated with them are different.

$$timed\,failures\,[\![Q]\!] \setminus timed\,failures\,[\![P]\!] = \{(\langle\rangle, [0, \infty) \times \{a\}\}$$

Successive elements become closer and closer; the sequence is a Cauchy sequence, which must have a limit—the fixed point of f—because of the completeness of the metric space. All such sequences, whatever the starting point, have the same limit. Consequently,

$$fix(f) = \lim_{n \to \infty} f^n(x)$$

for all x in A.

Having introduced the theoretical background, we now demonstrate that each t-guarded process term corresponds to a contraction mapping. According to Section A.2.3, a process term is t-guarded, if any execution of this term cannot reach a recursive call in less than t time units. Assume that the process term Q in the equation

$$N = Q$$

is t-guarded for $t > 0$. Let F_Q be the function on the finite timed failures model corresponding to Q. If $N_1 \Vert\!\upharpoonright_{tcsp} u = N_2 \Vert\!\upharpoonright_{tcsp} u$ then $F_Q(N_1) \Vert\!\upharpoonright_{tcsp} (u + t) = F_Q(N_2) \Vert\!\upharpoonright_{tcsp} (u + t)$ and it follows that

$$d(F_Q(N_1), F_Q(N_2)) \leq 2^{-t} * d(N_1, N_2)$$

where $2^{-t} < 1$.

F_Q is hence a contraction mapping on the complete metric space constituted by the FTF model and the distance d. Thus, Banach's fixed point theorem applies, assuring the existence and uniqueness of a fixed point of F_Q.

In conclusion, any well-timed CSP term has a unique fixed point in the *finite* timed failures model, which is the model underlying RT-Z.[13] The unique fixed point can be computed by using the following equation

$$timedfailures\, [\![N]\!] \Vert\!\upharpoonright_{tcsp} \infty = \bigcup_{n \in \mathbb{N}} (timedfailures\, [\![Q_n]\!] \Vert\!\upharpoonright_{tcsp} n * t)$$

where Q_n is defined inductively

$$Q_{n+1} = Q[Q_n / N]$$

and Q_0 can be chosen arbitrarily.

We have dealt only with simple recursion. The application of the above discussion to mutual recursion, i.e., to collections of recursively defined processes, is straightforward. We have hence omitted the discussion of mutual recursion.

A.2.5 Predicate Language

A timed specification is a predicate $S(s, \aleph)$ on timed failures. It describes the behaviour required of a process in terms of constraints on its timed traces and timed refusals. A process

[13] According to Schneider [1999a], the situation is different in the (general) timed failures model. In this model, for a well-timed CSP term to have a unique fixed point, it must not contain any infinite nondeterminism, i.e., it must not contain any operator introducing unbounded nondeterminism.

P meets a specification $S(s, \aleph)$ if, and only if, all timed failures that are associated with P satisfy S. Formally,

$$P \ \mathbf{sat} \ S(s, \aleph) \Leftrightarrow \forall (s, \aleph) \in \mathit{timedfailures} \, [\![P]\!] \bullet S(s, \aleph).$$

Timed specifications are usually expressed in terms of functions on timed traces, refusals and failures, which are dealt with in Appendix E.

A timed specification $S(s, \aleph)$ is *admissible* if the fact that all finite approximations of a timed failure of infinite length meet S implies that S holds of this infinite timed failure as well.

$$(\forall t : \mathbb{R}_0^+ \bullet S(s \upharpoonright_{ttr} t, \aleph \upharpoonright_{tref} t)) \Rightarrow S(s, \aleph)$$

Admissible specifications have the merit that to check a process for satisfaction it suffices to check all finite duration timed failures for satisfaction.

If the failure of a predicate to hold of a timed behaviour can always be realised within finite time, then that predicate corresponds to an admissible specification. Inadmissible predicates are those that may hold of all finite approximations of a timed failure $(s, \aleph)$ but not of $(s, \aleph)$ itself. An example of a specification that is not admissible is $\#s < \infty$.

Specification macros

Building timed specifications directly as predicates on timed traces s and timed refusals $\aleph$ becomes cumbersome and error-prone as problems exceed a certain level of complexity. Furthermore, there are frequently occurring specification patterns with regard to safety and liveness requirements as well as to assumptions about the process environment. *Specification macros* are abbreviations of predicates; they can be used as the building blocks of frequently occurring specification patterns. Specification macros are useful not only for specifying requirements, but they also allow us to reason about specifications on a higher level of abstraction. Relationships between macros can be established, which allow us to reason about specifications at the level of macros rather than directly at the level of timed traces and refusals. Specification macros are given in terms of the free variables s and $\aleph$.

In this section, we introduce only some important specification macros that we need in the remainder of this thesis.

The specification macro at simply expresses that a particular event is recorded in the timed trace as having occurred at a particular time.

$$a \ \mathsf{at} \ t \equiv \langle (t, a) \rangle \ \mathbf{in} \ s$$

The liveness of a process at time t with respect to a particular event a can have two consequences: either the event is performed and hence appears in the timed trace, or else it does not appear in the timed refusal.

$$a \ \mathsf{live} \ t \equiv a \ \mathsf{at} \ t \vee (t, a) \notin \aleph$$

An equivalent definition is $(t, a) \in \aleph \Rightarrow a \ \mathsf{at} \ t$, which states that if a is offered by the environment, then it must occur.

The following specification macro abbreviates a more intricate aspect of liveness. A process might enable an event a at some time t, and continue to offer it at least until some event in A occurs.

a live from t until $A \equiv ([t, \text{begin}_{ttr}((s \upharpoonleft_{ttr} t) \restriction_{ttr} A)) \times \{a\}) \cap \aleph = \varnothing$

The definition states that between t and the first occurrence of a subsequent event from A, the event a cannot be refused.

The following specification macro is used in order to express assumptions about the environment of a process. An event a is open at time t if the environment of the process will not block the occurrence of the timed event (t, a). Thus, if the event does not occur, it is due to the refusal of the process to perform it, so $(t, a) \in \aleph$.

a open $t \equiv a$ at $t \vee (t, a) \in \aleph$

Either a is actually performed at t, which clearly reveals that the environment was open to the occurrence of a, or it appears in the timed refusal.

Another form of environmental assumption is that the environment always allows all occurrences of events from A that the process wishes to perform. Such an environment will make a process *active* on the set A, because it will perform all occurrences of A urgently.

A active $\equiv [0, \infty) \times A \subseteq \aleph$

Several relationships between specification macros have been established by appealing to their definitions. These relationships allow one to reason about specifications on a higher level of abstraction.

A straightforward relationship between the macros at, live and open is the following:

a at $t \Leftrightarrow a$ live $t \wedge a$ open t.

If a has occurred at time t, then the process and the environment must have offered this event at this time. Conversely, if the process and the environment are known to enable event a at time t, then it must occur at this time in accordance with the maximal progress assumption.

A.2.6 Verification

Schneider [1999b] has developed a complete calculus for proving timed CSP processes correct with respect to timed specifications.

In theory, it is possible to prove a timed CSP process correct with respect to a timed specification by checking that each timed failure that is associated with the process meets the specifying predicate. In practice, however, a more structured approach to verification is necessary.

In Section A.2.4, we presented the denotational semantics of timed CSP, which is compositional. Compositionality means that the denotations of compound processes are defined in terms of the denotations of their component processes. Analogously, it is possible to provide

a compositional proof system, which reflects the structure of the semantic equations. Each proof rule allows the deduction of a timed specification satisfied by a compound process from the timed specifications that are met by its component specifications. There is a single inference rule or axiom for each operator of the timed CSP syntax.

In the remainder of this section, we present the proof rules only for selected operators.

Event Prefix. Provided the process P meets the specification S, the behaviour of the process $a \to P$ subsequent to an occurrence of a will satisfy S. The behaviour up to the occurrence of a is the continual offer of a.

$$\frac{P \textbf{ sat } S(s, \aleph)}{\begin{array}{ll} a \to P \textbf{ sat} & a \text{ live from } 0 \text{ until } \{a\} \\ & \wedge\ s \neq \langle\rangle \Rightarrow \mathrm{first}(s) = a \wedge S((\mathrm{tail}(s), \aleph) - \mathrm{begin}_{ttr}(s)) \end{array}}$$

The timed specifications satisfied by $a \to P$ are determined in terms of the timed specifications satisfied by P.

Note the correspondence between the structure of the previous inference rule and the structure of the semantic equation of the event prefix operator. The correctness of the inference rule is backed up by the semantic equation.

Timeout. The following inference rule provides an example of reasoning about time-sensitive behaviour. A behaviour of $P \rhd\{d\}\ Q$ is either a timed failure associated with P, whose first event is recorded before time d, or else a timed failure of Q delayed by d time units. In the latter case, the refusal information up to the timeout must be consistent with P.

$$\frac{\begin{array}{l} P \textbf{ sat } S(s, \aleph) \\ Q \textbf{ sat } T(s, \aleph) \end{array}}{\begin{array}{ll} P \rhd\{d\}\ Q \textbf{ sat} & (\mathrm{begin}_{ttr}(s) \leqslant_{\mathbb{R}} d \wedge S(s, \aleph) \\ & \vee\ \mathrm{begin}_{ttr}(s) \geqslant_{\mathbb{R}} d \wedge S(\langle\rangle, \aleph \upharpoonright_{tref} d) \wedge T((s, \aleph) \mathbin{\dot{-}}_{tfail} d)) \end{array}}$$

Again, the structure of the inference rule is analogous to the structure of the corresponding semantic equation.

Recursion Induction. To prove that a recursively defined process meets some specification, we need a corresponding induction principle.

As discussed in Section A.2.4, the finite duration timed failures of a well-timed recursive definition $N = P$ are uniquely determined. Let $F(X) = P[X/N]$ be the function on timed failures corresponding to the body P of the recursive definition. If S is an admissible specification, which is satisfiable (for which there is at least one process P_0 satisfying it), and if S is preserved by applications of F, then the recursively defined process N meets S. Formally,

$$\frac{\forall X \bullet X \textbf{ sat } S(s, \aleph) \Rightarrow F(X) \textbf{ sat } S(s, \aleph)}{N \textbf{ sat } S(s, \aleph)} \quad \left[\begin{array}{l} \exists P_0 \bullet P_0 \textbf{ sat } S(s, \aleph) \\ S(s, \aleph) \text{ is admissible} \\ N = F(N) \end{array}\right]$$

This induction rule is correct for the following reason.

There is a process P_0 satisfying S by assumption. The antecedent of the rule ensures that also $F^n(P_0)$ meets S for all $n \in \mathbb{N}$. According to Section A.2.4, each finite duration behaviour of N is contained in $F^n(P_0)$ for some $n \in \mathbb{N}$, because

$$\mathit{timed\,failures}\,[\![N]\!] \Vert\!\restriction_{tcsp} \infty = \bigcup_{n \in \mathbb{N}} (\mathit{timed\,failures}\,[\![F^n(P_0)]\!] \Vert\!\restriction_{tcsp} n * t).$$

This means that all *finite duration* timed failures associated with N satisfy S.

Furthermore, for each infinite duration timed failure $(s, \aleph)$ of N, all of its finite approximations must also be associated with N, because[14]

$$\mathit{timed\,failures}\,[\![N]\!] \subseteq \overline{\bigcup_{n \in \mathbb{N}} (\mathit{timed\,failures}\,[\![F^n(P_0)]\!] \Vert\!\restriction_{tcsp} n * t)}.$$

Since S holds of all these finite duration behaviours and since S is an admissible specification, also the infinite duration timed failures of N must satisfy S. Thus, all timed failures (finite or infinite) associated with N meet S, hence N **sat** S.

A.2.7 Discussion

Timed CSP, as introduced in this section, has several strengths.

It provides a powerful notation, its formal foundation and also verification techniques for defining and analysing the external behaviour of a system in terms of the temporal ordering and the timing of its interactions with the environment. It is particularly useful for specifying systems that are reactive in nature.

Furthermore, timed CSP, as an extension of CSP, is able to define certain aspects of system and software architectures, e.g., concurrency, distribution and communication. This was demonstrated by Allen and Garlan, who designed the architectural description language (ADL) Wright [Allen, 1997], which is closely based on CSP.

On the other hand, there are also relevant shortcomings attributed to timed CSP.

Timed CSP is not designed to specify the structure and content of the information that is communicated between processes. Channels between processes can be associated with alphabets—sets of values carried by events—but the notation of timed CSP for defining alphabets and its values is only ad-hoc; and, more important, it is not covered by the semantic model of timed CSP. In brief, timed CSP lacks an integrated notation for defining abstract data types.

Moreover, processes are the only mechanism for structuring system specifications. The weakness of process descriptions, which makes them inadequate as a high-level structuring mechanism, is the fact that process interfaces are not defined explicitly, i.e., they must be derived which is hard for complex processes, consisting of several sub-processes. We think that the availability of the interface is a major prerequisite when describing system components and their interdependencies.

[14] We have not dealt with the computation of fixed points in the (general) timed failures model. The following subset relationship is motivated by [Schneider, 1999b, p. 361].

To conclude, the merits and drawbacks of timed CSP and the ones of Z discussed in the previous section seem to complement each other, making their integration a worthwhile task.

A.2.8 Other Approaches to Modelling and Reasoning about Real-Time

Nicollin and Sifakis [1991] gave an introduction to timed process algebras by comparing several instances, including (a previous version of) timed CSP. They presented fundamental assumptions about timed systems and the nature of time (cf. Section A.2.1) and introduced the concept of timed transition systems, on which the operational semantics of timed CSP is based. They also proposed some general principles for the real-time extension of untimed process algebras, which concern the compatibility of the interpretations associated with a process in the untimed and the timed setting.

Let us outline four important instances of timed process algebras.

Baeten and Bergstra [1991] introduced the real-time process algebra ACP_ρ, which is an extension of ACP (Algebra of Communicating Processes). As a characteristic feature of ACP_ρ (and also ACP), its formal meaning is defined by an axiomatic semantics, in contrast to the operational and denotational approaches taken by the other timed process algebras. In fact, ACP and ACP_ρ are collections of axioms that are defined with respect to a simple process algebra syntax.

ET-LOTOS, developed by Léonard and Leduc [1997], is a real-time extension of LOTOS [Bolognesi et al., 1995]. The time domain underlying ET-LOTOS is not fixed; it is modelled by a sort, which must be instantiated by any ET-LOTOS specification by using the algebraic specification language ACT ONE [Ehrig and Mahr, 1985] (which is a base language of LOTOS). Syntactically, ET-LOTOS basically extends LOTOS by two constructs: the life reducer and the delay operator. The life reducer allows the specifier to restrict the interval in which an action is offered (which roughly corresponds to the timeout operator of timed CSP), and the delay operator prefixes a behaviour expression by a delay. The authors demonstrated that all relevant patterns of time-sensitive behaviour can be modelled by combining these two basic concepts. The operational semantics of ET-LOTOS was defined by strictly separating event and time transitions, analogously to the definition of the operational semantics of timed CSP presented in Section A.2.2. ET-LOTOS and timed CSP are very close in spirit, being based on the same computational model; this is witnessed by Bryans et al. [1995] who defined a denotational semantics for ET-LOTOS similar to that of timed CSP.

Moller and Tofts [1990] developed a real-time extension of Milner's Calculus of Communicating Systems [Milner, 1989], called TCCS. It is based on a discrete model of time. Its syntax extends that of CCS by delay operators and a so-called *weak* choice operator (as opposed to the *strong* choice operator adopted from CCS), which allows the passage of time to resolve a choice and therefore to express timeouts. The operational semantics of TCCS is defined analogously to the approach taken by timed CSP and also ET-LOTOS: the inference rules characterising the possible action transitions of processes (already defined by CCS) are extended by additional rules characterising the time transitions of processes. The definition of a bisimulation equivalence relation on TCCS processes and the derivation of an equational theory, which is sound with respect to bisimulation, is a prerequisite for reasoning about TCCS processes.

Hennessy and Regan [1995] proposed a (deliberately) minor extension to the process algebra CCS, called Timed Process Language (TPL), with a mathematically simple notion of time. The extension basically consists in introducing a special action σ, which denotes idling until the next clock cycle (the passage of time). In other words, the approach is based on a discrete model of time. The semantic theory of TPL is based on testing; it is an extension of the theory of testing from de Nicola and Hennessy [1984] to the timed setting.

TPL was not designed to be universally applicable, but focuses on application areas in which time plays a restricted rôle such as protocol verification. For these applications, TPL is adequate because it does not introduce unnecessary complexity. TPL is considered by the authors "to provide a sound basis on which to develop more extensive theories of timed systems."

In addition to timed process algebras, there are other approaches to modelling and reasoning about time-sensitive systems. Surveys of the most important approaches can be found in [Heitmeyer and Mandrioli, 1996] and [de Bakker et al., 1991]. In the remainder, we outline three representative examples of these approaches.

The formalism TTM/RTTL [Ostroff, 1991] comprises two notations. An automaton model (Timed Transition Model), mainly extended by assigning lower and upper time bounds to state transitions, is used to construct models of real-time systems; and a real-time temporal logic (RTTL) serves to abstractly express properties that such models are required to satisfy. This notational separation between a system model and the properties that it is required to satisfy is a crucial language feature, which is also shared by timed CSP.

The formalism Modechart/RTL [Jahanian and Mok, 1994] also consists of a pair of specification languages. Modechart is a graphical language for constructing models of real-time systems. It is basically a restriction of Statecharts [Harel, 1987] to those language constructs that allow a concise and succinct specification of real-time systems while avoiding the semantic problems associated with the more powerful constructs of Statecharts. Properties that a Modechart must satisfy are specified in terms of the Real Time Logic (RTL), which is also the logic used to define the semantics of Modecharts. One benefit of this approach is the provision of tool support for checking a Modechart for consistency and completeness and for verifying the satisfaction of properties expressed as RTL assertions.

Both TTM/RTTL and Modechart/RTL are based on a discrete time model, whereas the underlying time domain of timed CSP is dense.

The Duration Calculus [Chaochen et al., 1991] is a real-time interval logic based on state durations. A system is modelled in terms of a collection of state variables that are functions of time, which is modelled by real numbers. System requirements are specified by means of duration formulas that are predicates containing integrals over state assertions (predicates on state variables). That is, the Duration Calculus has a completely different underlying model than timed CSP. This underlying model is ideal for specifying requirements concerning the duration of intervals in which particular conditions are satisfied, but it is less appropriate for specifying other aspects of the dynamic behaviour, e.g., the temporal ordering of interactions. Which underlying model is more appropriate depends on the specific real-time system to be developed.

Appendix B

Satisfaction of the Properties of the Timed Failures Model

The aim of this appendix is to prove that our semantic definitions in Section 9.2.4 are consistent with the properties of the timed failures model formalised on p. 126 and explained on pp. 249 ff. More precisely, we aim to establish that each Z specification (constituting the Z part of a concrete specification unit) is mapped to a set of timed failures satisfying these properties.

B.1 Lemmata

We first state some lemmata we need throughout the main proof. We do not prove these lemmata but provide informal arguments why they are true.

The first two lemmata are immediate consequences of the definition of the function $traces_Z$ and the relation *PreHist* in Sections 9.2.3 and 9.2.4.

Lemma B.1
Given a trace tr that is related to the history h. Then, each prefix tr' of tr is related to some history h' that is a pre-history of h.

$$\frac{tr = traces_Z\, h \qquad tr' \ \mathbf{prefix}\ tr}{\exists\, h' : HS \mid (h', h) \in PreHist \bullet tr' = traces_Z\, h'}$$

Lemma B.2
Given a trace tr and one of its prefixes tr' such that the two are related to the histories h and h', respectively. If there is a history h^* of which both h and h' are pre-histories, then either h must be a pre-history of h' or vice versa. Since we have assumed that h' is related to a trace

that is a prefix of a trace that is related to h*, we can conclude that* h' *is a pre-history of* h*.*

$$\frac{\begin{array}{l} tr = traces_Z\, h \\ tr' = traces_Z\, h' \\ tr' \ \mathbf{prefix}\ tr \\ \exists\, h^* : HS \bullet (h, h^*) \in PreHist \land (h', h^*) \in PreHist \end{array}}{(h', h) \in PreHist}$$

Lemma B.3

Given a timed trace s*. Let* h*,* $pref1$ *and* $pref2$ *be histories that are related to the entire timed trace* s*, to the projection of* s *onto the interval* $[0, t1]$ *and to the projection of* s *onto the interval* $[0, t2]$*, respectively, and suppose that both* $pref1$ *and* $pref2$ *are pre-histories of* h*. If* $t2 \geqslant_{\mathbb{R}} t1$*, then* $pref1$ *must be a pre-history of* $pref2$*.*

$$\frac{\begin{array}{l} \text{strip}\ s = traces_Z\, h \\ \text{strip}(s \upharpoonright_{ttr} t1) = traces_Z\, pref1 \land (pref1, h) \in PreHist \\ \text{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref2 \land (pref2, h) \in PreHist \\ t2 \geqslant_{\mathbb{R}} t1 \end{array}}{(pref1, pref2) \in PreHist}$$

The lemma is an immediate consequence of Lemma B.2, because $\text{strip}(s \upharpoonright_{ttr} t1)$ *is a prefix of* $\text{strip}(s \upharpoonright_{ttr} t2)$ *for all* $t2 \geqslant_{\mathbb{R}} t1$*.*

Lemma B.4

Let s *and* s' *be two timed traces such that* s' *is a prefix of* s*. Let further the traces corresponding to* s *and* s' *be related to the histories* h *and* h'*, respectively, and let* h' *be a pre-history of* h*. If the time values recorded in the timed trace* s *are consistent with the simultaneity information recorded in the component* $simultan_of$ *of the history* h*, then we can conclude that also the time values recorded in the timed trace* s' *are consistent with the simultaneity information recorded in the history* h'*.*

$$\frac{\begin{array}{l} s' \ \mathbf{prefix}\ s \\ \text{strip}\, s = traces_Z\, h \\ \text{strip}\, s' = traces_Z\, h' \\ (h', h) \in PreHist \\ (\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h \bullet time_of(s\, i) = time_of(s\, j)) \end{array}}{(\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\, i) = time_of(s'\, j))}$$

According to the definition of $PreHist$*, the component* $simultan_of$ *of* h' *must be a subset of that of* h*. Furthermore, the time values recorded in* s' *are identical with those recorded in the corresponding timed events of* s*. The consistency of the time values of* s *with the simultaneity information fixed by* h *thus guarantees the consistency of the time values of* s' *with the simultaneity information fixed by* h'*.*

Lemma B.5

Given a timed trace s*. Let* h *and* $pref$ *be histories that are related to the entire timed trace* s *and to the projection of* s *onto the interval* $[0, t2]$*, respectively, and let* $pref$ *be a pre-history of*

h. Finally let ext be a history of which $pref$ is a pre-history. Then, the component $simultan_of$ of h and ext projected onto the interval $[0, t2]$ are equal.

$$\frac{\begin{array}{l}\mathrm{strip}\ s = traces_Z\ h \\ \mathrm{strip}(s \upharpoonright_{ttr} t2) = traces_Z\ pref \\ (pref, h) \in PreHist \\ (pref, ext) \in PreHist\end{array}}{\begin{array}{l}simultan_of\ ext \cap \{i, j : \mathbb{N} \mid time_of(s\ i) \leqslant_{\mathbb{R}} t2 \wedge time_of(s\ j) \leqslant_{\mathbb{R}} t2\} = \\ simultan_of\ h \cap \{i, j : \mathbb{N} \mid time_of(s\ i) \leqslant_{\mathbb{R}} t2 \wedge time_of(s\ j) \leqslant_{\mathbb{R}} t2\}\end{array}}$$

The projection of s onto the interval $[0, t2]$ is entirely covered by the history $pref$. Since $pref$ is a pre-history of both h and ext, their components $simultan_of$ restricted to the part covered by $pref$ must be equal. This is a consequence of the definition of $PreHist$.

B.2 Properties of the Timed Failures Model

Theorem B.1
Let CSU be a concrete specification unit and $\mathcal{M}$ be a model associated with its Z part. Let

$HS == Histories(\!|\{(\mathcal{M}, CSU.OPS)\}|\!)$.

Then

$\bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\} \in TimedFailuresModel$.

That is, each set of timed failures that is associated with the Z part of a concrete specification unit satisfies the properties of the timed failures model defined on p. 126 and explained on pp. 249 ff.

Proof.

The proof is organised according to the decomposition into the conditions TF1, TF2, TF3/C1 and TF3/C2, see the explanation on pp. 249 ff. Throughout the following proof, we mark those parts of the formulae that are manipulated in the current step by a rectangle.

TF 1

We assume that the *Init* schema of the Z part of *CSU* is satisfiable. Each consistent specification unit must meet this condition.

$$\frac{\exists InitSt : InitModel(\!|\{\mathcal{M}\}|\!) \cap StateModel(\!|\{\mathcal{M}\}|\!)}{(\langle\rangle, \varnothing) \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}}$$

$\exists InitSt : InitModel(\!|\{\mathcal{M}\}|\!) \cap StateModel(\!|\{\mathcal{M}\}|\!)$

$\vdash$ [def. HS, $Histories$]

$((\langle\rangle, \varnothing), \langle InitSt\rangle, \langle\rangle) \in HS$

$\vdash$ [def. $traces_Z$]

$\langle\rangle \in \{h : HS \bullet traces_Z\, h\}$

$\vdash$ [def. $failures_Z$]

$(\langle\rangle, \varnothing) \in \bigcup\{h : HS \bullet failures_Z(\!|\{(h, HS)\}|\!)\}$

$\vdash$ [def. $timedfailures_Z$]

$(\langle\rangle, \varnothing) \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}$

$\square$

TF 2

$$\frac{\begin{array}{l}(s, \aleph) \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\} \\ \exists s^* \bullet s = s' \frown s^* \wedge \aleph' \subseteq \aleph \parallel\!\restriction_{tref} (\text{begin}_{ttr}\ s^*)\end{array}}{(s', \aleph') \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}}$$

$(s, \aleph) \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}$

$\vdash$ [def. $timedfailures_Z$]

$\exists h : HS \bullet$
$\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$
$\quad \text{strip}\, s = traces_Z\, h \wedge$
$\quad (\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet$
$\qquad (\text{strip}(s \restriction_{ttr} t), \{e : Event \mid (t, e) \in \aleph\}) \in failures_Z(\!|\{(prefix, HS)\}|\!))$

$\vdash$ [Lemma B.1; s' **prefix** $s \Rightarrow \text{strip}(s')$ **prefix** $\text{strip}(s)$]

$\exists h, h' : HS \mid \boxed{(h', h) \in PreHist} \bullet$
$\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$
$\quad \text{strip}\, s = traces_Z\, h \wedge$
$\quad \boxed{\text{strip}\, s' = traces_Z\, h'} \wedge$
$\quad (\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet$
$\qquad (\text{strip}(s \restriction_{ttr} t), \{e : Event \mid (t, e) \in \aleph\}) \in failures_Z(\!|\{(prefix, HS)\}|\!))$

$\vdash$ [Lemma B.4]

$\exists h, h' : HS \mid (h', h) \in PreHist \bullet$
$\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ \boxed{h'} \bullet time_of(\boxed{s'}\, i) = time_of(\boxed{s'}\, j)) \wedge$
$\quad \text{strip}\, s' = traces_Z\, h' \wedge$
$\quad (\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet$
$\qquad (\text{strip}(s \restriction_{ttr} t), \{e : Event \mid (t, e) \in \aleph\}) \in failures_Z(\!|\{(prefix, HS)\}|\!)))$

$\vdash$ [def. $failures_Z$]

$\exists h, h' : HS \mid (h', h) \in PreHist \bullet$
$\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \wedge$
$\quad \mathrm{strip}\, s' = traces_Z\, h' \wedge$
$\quad (\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet$

$$\boxed{\begin{array}{l} \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\ (\forall e : \{a : Event \mid (t, a) \in \aleph\} \bullet \\ \quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\} \\ \quad \vee \\ \quad (\exists e' : Event \mid (t, e') \notin \aleph \bullet chan_of(e) = chan_of(e') \wedge \\ \quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\ \quad\quad \vee\ chan_of(e) \in \mathsf{ran}\, ExecOf \\ \quad\quad\quad \wedge\ Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e')))) \end{array}}$$

$))$

$\vdash [t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \vee t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^*]$

$$\begin{array}{l}
\exists h, h' : HS \mid (h', h) \in PreHist \bullet \\
\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \wedge \\
\quad \mathrm{strip}\, s' = traces_Z\, h' \wedge \\
\quad (\forall t : \mathbb{T} \bullet \\
\quad\quad \boxed{t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^*} \wedge \\
\quad\quad (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad\quad (\forall e : \{a : Event \mid (t, a) \in \aleph\} \bullet \\
\quad\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad\quad\quad \vee\ chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad\quad\quad\quad \wedge\ Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e'))))) \\
\quad\quad \vee \\
\quad\quad \boxed{t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^*} \wedge \\
\quad\quad (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad\quad (\forall e : \{a : Event \mid (t, a) \in \aleph\} \bullet \\
\quad\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}
\end{array}$$

$$
\begin{array}{l}
\quad\quad\quad \vee \\
\quad\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad\quad \vee\, chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad\quad\quad \wedge\, Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e'))))))
\end{array}
$$

$$
\begin{array}{l}
\vdash [t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \Rightarrow (s \upharpoonright_{ttr} t = s' \upharpoonright_{ttr} t \\
\quad\quad\quad\quad \wedge\, \{e : Event \mid (t, e) \in \aleph'\} \subseteq \{e : Event \mid (t, e) \in \aleph\}); \\
\quad t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \Rightarrow \{e : Event \mid (t, e) \in \aleph'\} = \varnothing; \\
\quad \forall e : \varnothing \bullet \mathsf{false}; \\
\quad B \subseteq A \wedge (\forall x : A \bullet P(x)) \Rightarrow (\forall x : B \bullet P(x)); \\
\quad B \subseteq A \wedge x \notin A \Rightarrow x \notin B]
\end{array}
$$

$$
\begin{array}{l}
\exists h, h' : HS \mid (h', h) \in PreHist \bullet \\
\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \wedge \\
\quad \mathrm{strip}\, s' = traces_Z\, h' \wedge \\
\quad (\forall t : \mathbb{T} \bullet \\
\quad\quad t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \wedge \\
\quad\quad (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(\boxed{s'} \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad\quad (\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph'}\} \bullet \\
\quad\quad\quad\quad \mathrm{strip}(s' \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad\quad\quad \vee\, chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad\quad\quad\quad \wedge\, Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e'))))) \\
\quad\quad \vee \\
\quad\quad t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \wedge \\
\quad\quad (\exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad\quad (\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph'}\} \bullet \boxed{\mathsf{false}})))
\end{array}
$$

$$
\vdash [\text{Lemma B.1};\ (s' \upharpoonright_{ttr} t)\ \mathbf{prefix}\ (s \upharpoonright_{ttr} t)]
$$

$$
\begin{array}{l}
\exists h, h' : HS \mid (h', h) \in PreHist \bullet \\
\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \wedge \\
\quad \mathrm{strip}\, s' = traces_Z\, h' \wedge \\
\quad (\forall t : \mathbb{T} \bullet \\
\quad\quad t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \wedge
\end{array}
$$

$(\exists\, prefix : HS \mid (prefix, h) \in PreHist \bullet$

$\quad \mathrm{strip}(s' \restriction_{ttr} t) = traces_Z\, prefix \land$

$\quad (\forall\, e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$

$\qquad \mathrm{strip}(s' \restriction_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$

$\qquad \lor$

$\qquad (\exists\, e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land$

$\qquad\quad (chan_of(e) \in \mathrm{ran}\, TermOf$

$\qquad\quad \lor\, chan_of(e) \in \mathrm{ran}\, ExecOf$

$\qquad\qquad \land\, Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e')))))$

$\lor$

$t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \land$

$(\exists\, prefix : HS \mid (prefix, h) \in PreHist \bullet$

$\quad (\boxed{\exists\, prefix' : HS \mid (prefix', prefix) \in PreHist} \bullet$

$\qquad \boxed{\mathrm{strip}(s' \restriction_{ttr} t) = traces_Z\, prefix'} \land$

$\qquad (\forall\, e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$

$$\boxed{\begin{array}{l} \mathrm{strip}(s' \restriction_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix', ext) \in PreHist \bullet traces_Z\, ext\} \\ \lor \\ (\exists\, e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land \\ \quad (chan_of(e) \in \mathrm{ran}\, TermOf \\ \quad \lor\, chan_of(e) \in \mathrm{ran}\, ExecOf \\ \qquad \land\, Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e'))) \end{array}}$$

$))))$

$\vdash$ [transitivity of $PreHist$]

$\exists\, h, h' : HS \mid (h', h) \in PreHist \bullet$

$\quad (\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \land$

$\quad \mathrm{strip}\, s' = traces_Z\, h' \land$

$\quad (\forall\, t : \mathbb{T} \bullet$

$\qquad t <_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \land$

$\qquad (\exists\, prefix : HS \mid (prefix, h) \in PreHist \bullet$

$\qquad\quad \mathrm{strip}(s' \restriction_{ttr} t) = traces_Z\, prefix \land$

$\qquad\quad (\forall\, e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$

$\qquad\qquad \mathrm{strip}(s' \restriction_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$

$\qquad\qquad \lor$

$\qquad\qquad (\exists\, e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land$

$$(chan_of(e) \in \mathrm{ran}\, TermOf$$
$$\lor\ chan_of(e) \in \mathrm{ran}\, ExecOf \land Inputs \lhd (val_of\ e) =$$
$$Inputs \lhd (val_of\ e')))))$$

$$\lor$$

$$t \geqslant_{\mathbb{R}} \mathrm{begin}_{ttr}\, s^* \land$$
$$(\exists\, prefix' : HS \mid \boxed{(prefix', h) \in PreHist} \bullet$$
$$\mathrm{strip}(s' \upharpoonright_{ttr} t) = traces_Z\, prefix' \land$$
$$(\forall\, e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$$
$$\mathrm{strip}(s' \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix', ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\lor$$
$$(\exists\, e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land$$
$$(chan_of(e) \in \mathrm{ran}\, TermOf$$
$$\lor\ chan_of(e) \in \mathrm{ran}\, ExecOf \land Inputs \lhd (val_of\ e) =$$
$$Inputs \lhd (val_of\ e'))))))$$

$\vdash [A \land B \lor C \land B \equiv (A \lor C) \land B]$

$$\exists\, h, h' : HS \mid (h', h) \in PreHist \bullet$$
$$(\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \land$$
$$\mathrm{strip}\, s' = traces_Z\, h' \land$$
$$(\forall\, t : \mathbb{T} \bullet (\exists\, prefix : HS \mid (prefix, h) \in PreHist \bullet$$
$$\mathrm{strip}(s' \upharpoonright_{ttr} t) = traces_Z\, prefix \land$$
$$(\forall\, e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$$
$$\mathrm{strip}(s' \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\lor$$
$$(\exists\, e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land$$
$$(chan_of(e) \in \mathrm{ran}\, TermOf$$
$$\lor\ chan_of(e) \in \mathrm{ran}\, ExecOf \land Inputs \lhd (val_of\ e) =$$
$$Inputs \lhd (val_of\ e'))))))$$

$\vdash$ [def. $failures_Z$]

$$\exists\, h, h' : HS \mid (h', h) \in PreHist \bullet$$
$$(\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \land$$
$$\mathrm{strip}\, s' = traces_Z\, h' \land$$
$$(\forall\, t : \mathbb{T} \bullet \exists\, prefix : HS \mid (prefix, h) \in PreHist \bullet$$
$$(\mathrm{strip}(s' \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph'\}) \in failures_Z (\!| \{(prefix, HS)\} |\!))$$

$\vdash$ [Lemma B.2; $\mathrm{strip}(s' \upharpoonright_{ttr} t)$ **prefix** $\mathrm{strip}\ s'$]

$$\exists\, h' : HS \bullet$$

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h' \bullet time_of(s'\ i) = time_of(s'\ j)) \wedge$$
$$\text{strip}\, s' = traces_Z\, h' \wedge$$
$$\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, \boxed{h'}) \in PreHist \bullet$$
$$(\text{strip}(s' \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph'\}) \in failures_Z(\!|\{(prefix, HS)\}|\!)$$

$\vdash$ [def. $timedfailures_Z$]

$$(s', \aleph') \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}$$

□

TF 3

$$\frac{(s, \aleph) \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}}{\begin{array}{l}\exists \aleph' : TimedRefusal \mid \aleph \subseteq \aleph' \wedge \\ \quad (s, \aleph') \in \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\} \bullet \\ \quad (\forall t : \mathbb{T};\ a : Event \bullet \\ \qquad ((t, a) \notin \aleph' \Rightarrow ((s \upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \Vert\!\upharpoonright_{tref} t) \in \\ \qquad\quad \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\}) \\ \qquad \wedge \\ \qquad (t >_{\mathbb{R}} 0_{\mathbb{R}} \wedge \neg (\exists \epsilon : \mathbb{T}^+ \bullet [t -_{\mathbb{R}} \epsilon, t) \times \{a\} \subseteq \aleph') \Rightarrow \\ \qquad ((s \Vert\!\upharpoonright_{ttr} t) \frown \langle (t, a) \rangle, \aleph' \Vert\!\upharpoonright_{tref} t) \in \\ \qquad\quad \bigcup\{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\})\end{array}}$$

Let $(s, \aleph) \in timedfailures_Z(\!|\{(h, HS)\}|\!)$ for an arbitrary history $h \in HS$.

Our first goal is to construct, with respect to the chosen timed refusal $\aleph$, the set of all maximal timed refusals $\aleph'$, because we must establish the existence of such a maximal timed refusal in *TF*3. To this end, we need two auxiliary functions.

$$Equiv_{Term} == \lambda\, ch : \mathsf{ran}\, TermOf \bullet \{e : Event \mid chan_of(e) = ch\}$$
$$Equiv_{Exec} == \lambda\, ch : \mathsf{ran}\, ExecOf \bullet \lambda\, in : Binding \bullet$$
$$\{e : Event \mid chan_of(e) = ch \wedge Inputs \lhd val_of(e) = in\}$$

The function $Equiv_{Term}$ maps each channel that carries termination events to the class (set) of all termination events that can occur on that channel. Similarly, the function $Equiv_{Exec}$ maps each channel that carries execution events to a function, which, in turn, maps each input parameter binding to the class (set) of all execution events that can occur on the given channel and that associate the given values with the input parameters.

We now define the set of all maximal timed refusals with respect to the chosen $\aleph$. Each maximal timed refusal $\aleph'$ in this set

- must contain $\aleph$,
- must contain for each time instant all events that are not possible continuations of the timed trace s and

- must contain for each time instant and for each channel carrying termination or execution events all but one event of the respective class.

$$
\begin{aligned}
\aleph_{max} == \{ & \aleph' : TimedRefusal \mid \forall t : \mathbb{T} \bullet \\
& \exists_1 ref == \{e : Event \mid (t, e) \in \aleph\};\ ref' == \{e : Event \mid (t, e) \in \aleph'\} \bullet \\
& \exists nmem_{Term} : \operatorname{ran} TermOf \rightarrow Event;\ nmem_{Exec} : \operatorname{ran} ExecOf \rightarrow Binding \rightarrow Event \mid \\
& \quad \forall ch : \operatorname{ran} TermOf \bullet (nmem_{Term}\, ch) \in (Equiv_{Term}\, ch) \setminus ref \land \\
& \quad \forall ch : \operatorname{ran} ExecOf;\ in : Binding \bullet (nmem_{Exec}\, ch\, in) \in (Equiv_{Exec}\, ch\, in) \setminus ref \bullet \\
& \quad ref' = ref \\
& \quad \cup \\
& \quad \{e : Event \mid \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \\
& \qquad \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \land \\
& \qquad\qquad (pref, h) \in PreHist \land (pref, ext) \in PreHist \bullet traces_Z\, ext\}\} \\
& \quad \cup \\
& \quad \bigcup\{ch : \operatorname{ran} TermOf \bullet (Equiv_{Term}\, ch) \setminus \{nmem_{Term}\, ch\}\} \\
& \quad \cup \\
& \quad \bigcup\{ch : \operatorname{ran} ExecOf;\ in : Binding \bullet (Equiv_{Exec}\, ch\, in) \setminus \{nmem_{Exec}\, ch\, in\}\}
\end{aligned}
$$

The following lemmata, which are local with respect to TF3, are immediate consequences of the definition of the set $\aleph_{max}$.

Lemma B.6

$$
\frac{\aleph' \in \aleph_{max} \qquad (t, e) \in \aleph'}
{\begin{aligned}
& (t, e) \in \aleph \\
& \lor \\
& \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \land \\
& \qquad (pref, h) \in PreHist \land (pref, ext) \in PreHist \bullet traces_Z\, ext\} \\
& \lor \\
& (\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land \\
& \quad (chan_of(e) \in \operatorname{ran} TermOf \\
& \quad \lor chan_of(e) \in \operatorname{ran} ExecOf \land Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))
\end{aligned}}
$$

Lemma B.7

$$
\frac{\aleph' \in \aleph_{max} \qquad (t, e) \notin \aleph'}
{\begin{aligned}
& \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \in \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \land \\
& \qquad (pref, h) \in PreHist \land (pref, ext) \in PreHist \bullet traces_Z\, ext\}
\end{aligned}}
$$

Lemma B.8

$$\frac{\begin{array}{l}\aleph' \in \aleph_{max} \\ (t, e) \in \aleph \\ \text{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \in \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \wedge \\ \qquad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\}\end{array}}{\begin{array}{l}(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge \\ \quad (chan_of(e) \in \mathsf{ran}\, TermOf \\ \quad \vee\, chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))\end{array}}$$

Let $\aleph'$ be a maximal timed refusal with respect to $\aleph$, i.e., $\aleph' \in \aleph_{max}$.

We must prove four subgoals.

1. $\aleph \subseteq \aleph'$
2. $(s, \aleph') \in timedfailures_Z(\{(h, HS)\})$
3. TF 3/C1

   $\forall t2 : \mathbb{T};\ a : Event \mid (t2, a) \notin \aleph' \bullet$
   $\quad ((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle, \aleph' \Vert_{tref}\, t2) \in \bigcup\{h : HS \bullet timedfailures_Z(\{(h, HS)\})\}$
4. TF 3/C2

   $\forall t0 : \mathbb{T};\ a : Event \mid t0 >_{\mathbb{R}} 0_{\mathbb{R}} \wedge \neg (\exists \epsilon : \mathbb{T}^+ \bullet [t0 -_{\mathbb{R}} \epsilon, t0) \times \{a\} \subseteq \aleph') \bullet$
   $\quad ((s \Vert_{ttr}\, t0) \frown \langle (t0, a) \rangle, \aleph' \Vert_{tref}\, t0) \in \bigcup\{h : HS \bullet timedfailures_Z(\{(h, HS)\})\}$

ad 1.

$\vdash [\aleph' \in \aleph_{max};\ \text{def. } \aleph_{max}]$

$\aleph \subseteq \aleph'$

$\square$

ad 2.

$(s, \aleph) \in timedfailures_Z(\{(h, HS)\})$

$\vdash [\text{def. } timedfailures_Z]$

$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$

$\text{strip}\, s = traces_Z\, h \wedge$

$(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet$

$\qquad (\text{strip}(s \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph\}) \in failures_Z(\{(prefix, HS)\}))$

$\vdash [\text{def. } failures_Z; \forall e : \{a : S \mid P(a)\} \bullet P(e)]$

$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$

$$
\begin{array}{l}
\mathrm{strip}\, s = traces_Z\, h \wedge \\
(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad \mathrm{strip}(s \restriction_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad (\forall e : \{a : Event \mid (t, a) \in \aleph\} \bullet \\
\quad\quad\quad \mathrm{strip}(s \restriction_{ttr} t) \frown \langle e \rangle \notin \{pref, ext : HS \mid \mathrm{strip}(s \restriction_{ttr} t) = traces_Z\, pref \wedge \\
\quad\quad\quad\quad\quad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad\quad \vee \\
\quad\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad\quad \vee chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad\quad\quad \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))) \\
\quad\quad \wedge
\end{array}
$$

$$
\boxed{
\begin{array}{l}
(\forall e : \{a : Event \mid \\
\quad \mathrm{strip}(s \restriction_{ttr} t) \frown \langle a \rangle \notin \{pref, ext : HS \mid \mathrm{strip}(s \restriction_{ttr} t) = traces_Z\, pref \wedge \\
\quad\quad\quad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad \vee \\
\quad (\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(a) = chan_of(e') \wedge \\
\quad\quad (chan_of(a) \in \mathsf{ran}\, TermOf \\
\quad\quad \vee chan_of(a) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad \wedge Inputs \lhd (val_of\, a) = Inputs \lhd (val_of\, e')))\} \bullet \\
\quad \mathrm{strip}(s \restriction_{ttr} t) \frown \langle e \rangle \notin \{pref, ext : HS \mid \mathrm{strip}(s \restriction_{ttr} t) = traces_Z\, pref \wedge \\
\quad\quad\quad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad \vee \\
\quad (\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad \vee chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e'))))
\end{array}
}
$$

$)$

$\vdash$ [Lemma B.8; $((\forall x : A \bullet P(x)) \wedge (\forall x : B \bullet P(x)) \Rightarrow (\forall x : A \cup B \bullet P(x)))$]

$$
\begin{array}{l}
(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge \\
\mathrm{strip}\, s = traces_Z\, h \wedge \\
(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\quad \mathrm{strip}(s \restriction_{ttr} t) = traces_Z\, prefix \wedge \\
\quad\quad (\forall e : \{a : Event \mid
\end{array}
$$

$$(t, a) \in \aleph$$
$$\vee$$
$$\text{strip}(s \upharpoonright_{ttr} t) \frown \langle a \rangle \notin \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(a) = chan_of(e') \wedge (chan_of(a) \in \mathsf{ran}\, TermOf \vee chan_of(a) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, a) = Inputs \lhd (val_of\, e')))$$

$\} \bullet$

$$\text{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge (chan_of(e) \in \mathsf{ran}\, TermOf \vee chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))))$$

$\vdash$ [Lemma B.6; $A \subseteq B \Rightarrow ((\forall x : B \bullet P(x)) \Rightarrow (\forall x : A \bullet P(x)))$]

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$$
$$\text{strip}\, s = traces_Z\, h \wedge$$
$$(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet$$
$$\text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge$$
$$(\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph'}\} \bullet$$
$$\text{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge (chan_of(e) \in \mathsf{ran}\, TermOf \vee chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))))$$

$\vdash$ [$\text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge (prefix, h) \in PreHist \Rightarrow \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet ext\} = \{ext : HS \mid (prefix, ext) \in PreHist\}$; $B \subseteq A \Rightarrow (x \notin R(\!| A |\!) \Rightarrow x \notin R(\!| B |\!))$]

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge$$
$$\text{strip}\, s = traces_Z\, h \wedge$$
$$(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet$$

$$
\begin{array}{l}
\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge \\
\quad (\forall e : \{a : Event \mid (t,a) \in \aleph'\} \bullet \\
\qquad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \boxed{\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}} \\
\qquad \vee \\
\qquad (\exists e' : Event \mid (t,e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge \\
\qquad\quad (chan_of(e) \in \mathrm{ran}\, TermOf \\
\qquad\quad \vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))))
\end{array}
$$

$\vdash$ [def. $failures_Z$]

$$
\begin{array}{l}
(\forall i,j : \mathbb{N} \mid (i,j) \in simultan_of\, h \bullet time_of(s\, i) = time_of(s\, j)) \wedge \\
\mathrm{strip}\, s = traces_Z\, h \wedge \\
(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h) \in PreHist \bullet \\
\qquad\qquad (\mathrm{strip}(s \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph'\}) \in failures_Z(\!|\{(prefix, HS)\}|\!))
\end{array}
$$

$\vdash$ [def. $timed failures_Z$]

$$(s, \aleph') \in timed failures_Z(\!|\{(h, HS)\}|\!)$$

$\square$

ad 3.

$$(t2, a) \notin \aleph'$$

$\vdash$ [Lemma B.7]

$$
\begin{array}{l}
\mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle \in \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref \wedge \\
\qquad\qquad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet traces_Z\, ext\}
\end{array}
$$

$\vdash$ [Lemma B.5]

$$
\begin{array}{l}
\exists h^* : \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref \wedge \\
\qquad\qquad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet ext\} \mid \\
\qquad simultan_of\, h^* = simultan_of\, h \\
\qquad\quad \cap \{i,j : \mathbb{N} \mid time_of(s\, i) \leqslant_{\mathbb{R}} t2 \wedge time_of(s\, j) \leqslant_{\mathbb{R}} t2\} \\
\qquad\qquad \cup \{i : \mathbb{N} \mid time_of(s\, i) = t2 \bullet (i, \#(s \upharpoonright_{ttr} t2) + 1)\} \bullet \\
\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^*
\end{array}
$$

$\vdash$ [$(s, \aleph) \in timed failures_Z(\!|\{(h, HS)\}|\!)$;

$$
\begin{array}{l}
\quad (i,j) \in simultan_of\, h^* \Rightarrow \\
\qquad ((i,j) \in simultan_of\, h \vee j = \#(s \upharpoonright_{ttr} t2) + 1 \wedge time_of(s\, i) = t2); \\
\quad ((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle)(\#(s \upharpoonright_{ttr} t2) + 1) = (t2, a)]
\end{array}
$$

$$
\begin{array}{l}
\exists h^* : \{pref, ext : HS \mid \mathrm{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref \wedge \\
\qquad\qquad (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet ext\} \bullet \\
\quad (\forall i,j : \mathbb{N} \mid (i,j) \in simultan_of\, h^* \bullet time_of(((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) i) =
\end{array}
$$

$$time_of(((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle)j)) \wedge$$

$$\text{strip}(s \restriction_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^*$$

$$\vdash [t2 \leqslant_{\mathbb{R}} t \vee t2 >_{\mathbb{R}} t]$$

$$\exists h^* : \{pref, ext : HS \mid \text{strip}(s \restriction_{ttr} t2) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet ext\} \bullet$$

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle)i) = time_of(((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle)j)) \wedge$$

$$\text{strip}(s \restriction_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^* \wedge$$

$$\forall t : \mathbb{T} \bullet t2 \leqslant_{\mathbb{R}} t \vee t2 >_{\mathbb{R}} t$$

$$\vdash [(h^*, h^*) \in PreHist;\ \forall e : \varnothing \bullet \textsf{false};\ (s, \aleph') \in timedfailures_Z (\!| \{(h, HS)\} |\!)]$$

$$\exists h^* : \{pref, ext : HS \mid \text{strip}(s \restriction_{ttr} t2) = traces_Z\, pref \wedge (pref, h) \in PreHist \wedge (pref, ext) \in PreHist \bullet ext\} \bullet$$

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle)i) = time_of(((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle)j)) \wedge$$

$$\text{strip}(s \restriction_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^* \wedge$$

$$(\forall t : \mathbb{T} \bullet$$

$$t2 \leqslant_{\mathbb{R}} t \wedge$$

$$(\exists prefix : HS \mid \boxed{(prefix, h^*) \in PreHist} \bullet$$

$$\text{strip}((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle) = traces_Z\, prefix \wedge$$

$$(\forall e : \boxed{\varnothing} \bullet$$

$$\text{strip}((s \restriction_{ttr} t2) \frown \langle(t2, a)\rangle) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$

$$\vee$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \Vert\!\restriction_{tref} t2 \bullet chan_of(e) = chan_of(e') \wedge$$

$$(chan_of(e) \in \textsf{ran}\, TermOf$$

$$\vee\ chan_of(e) \in \textsf{ran}\, ExecOf$$

$$\wedge\ Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))))$$

$$\vee$$

$$
\begin{array}{l}
t2 >_{\mathbb{R}} t \wedge \\
(\exists prefix^* : HS \mid (prefix^*, h) \in PreHist \bullet \\
\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix^* \wedge \\
\quad (\forall e : \{a : Event \mid (t, a) \in \aleph'\} \bullet \\
\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix^*, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad \vee \\
\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad \vee\, chan_of(e) \in \mathsf{ran}\, ExecOf \\
\quad\quad\quad\quad \wedge\, Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e'))))))
\end{array}
$$

)

$\vdash [\exists x : S \bullet P(x) \wedge S \subseteq T \Rightarrow \exists x : T \bullet P(x);$ Lemma B.3; transitivity of $PreHist]$

$$
\begin{array}{l}
\exists h^* : \boxed{HS} \bullet \\
\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) i) = \\
\qquad\qquad\qquad\qquad\qquad\qquad time_of(((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) j)) \wedge \\
\quad \mathrm{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, h^* \wedge \\
\quad (\forall t : \mathbb{T} \bullet \\
\quad\quad t2 \leqslant_{\mathbb{R}} t \wedge \\
\quad\quad (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = traces_Z\, prefix \wedge \\
\quad\quad\quad (\forall e : \varnothing \bullet \\
\quad\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle \frown \langle e \rangle \notin \\
\qquad\qquad\qquad\qquad\qquad \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\} \\
\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad (\exists e' : Event \mid (t, e') \notin \aleph' \Vert\!\upharpoonright_{tref} t2 \bullet chan_of(e) = chan_of(e') \wedge \\
\quad\quad\quad\quad\quad (chan_of(e) \in \mathsf{ran}\, TermOf \\
\quad\quad\quad\quad\quad \vee\, chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Inputs \lhd (val_of\, e'))))) \\
\quad\quad \vee \\
\quad\quad t2 >_{\mathbb{R}} t \wedge \\
\quad\quad (\exists prefix^* : HS \mid (prefix^*, \boxed{h^*}) \in PreHist \bullet \\
\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix^* \wedge \\
\quad\quad\quad (\forall e : \{a : Event \mid (t, a) \in \aleph'\} \bullet \\
\quad\quad\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix^*, ext) \in PreHist \bullet traces_Z\, ext\}
\end{array}
$$

$$
\begin{aligned}
&\qquad\qquad\lor\\
&\qquad\qquad(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \land\\
&\qquad\qquad\quad (chan_of(e) \in \mathrm{ran}\, TermOf\\
&\qquad\qquad\quad \lor chan_of(e) \in \mathrm{ran}\, ExecOf \land Inputs \lhd (val_of\ e) =\\
&\qquad\qquad\qquad\qquad\qquad\qquad Inputs \lhd (val_of\ e'))))))
\end{aligned}
$$

$$
\begin{aligned}
\vdash\ & [t2 \leqslant_{\mathbb{R}} t \Rightarrow (\mathrm{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) = \mathrm{strip}(s \restriction_{ttr} t2) \frown \langle a \rangle \land\\
&\{e : Event \mid (t, e) \in \aleph' \Vert\!\restriction_{tref} t2\} = \varnothing)\\
&t2 >_{\mathbb{R}} t \Rightarrow (\mathrm{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) = \mathrm{strip}(s \restriction_{ttr} t) \land\\
&\{e : Event \mid (t, e) \in \aleph' \Vert\!\restriction_{tref} t2\} = \{e : Event \mid (t, e) \in \aleph'\})]
\end{aligned}
$$

$$
\begin{aligned}
&\exists h^* : HS \bullet\\
&\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h^* \bullet time_of((s \restriction_{ttr} t2 \frown \langle (t2, a) \rangle) i) =\\
&\qquad\qquad\qquad\qquad time_of((s \restriction_{ttr} t2 \frown \langle (t2, a) \rangle) j)) \land\\
&\quad \mathrm{strip}((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, h^* \land\\
&\quad (\forall t : \mathbb{T} \bullet\\
&\qquad t2 \leqslant_{\mathbb{R}} t \land\\
&\qquad (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet\\
&\qquad\quad \mathrm{strip}(\boxed{((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t}) = traces_Z\, prefix \land\\
&\qquad\quad (\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph' \Vert\!\restriction_{tref} t2}\} \bullet\\
&\qquad\qquad \mathrm{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin\\
&\qquad\qquad\qquad\qquad \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}\\
&\qquad\qquad \lor\\
&\qquad\qquad (\exists e' : Event \mid (t, e') \notin \aleph' \Vert\!\restriction_{tref} t2 \bullet chan_of(e) = chan_of(e') \land\\
&\qquad\qquad\quad (chan_of(e) \in \mathrm{ran}\, TermOf\\
&\qquad\qquad\quad \lor chan_of(e) \in \mathrm{ran}\, ExecOf \land Inputs \lhd (val_of\ e) =\\
&\qquad\qquad\qquad\qquad\qquad\qquad Inputs \lhd (val_of\ e')))))\\
&\qquad \lor\\
&\qquad t2 >_{\mathbb{R}} t \land\\
&\qquad (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet\\
&\qquad\quad \mathrm{strip}(\boxed{((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t}) = traces_Z\, prefix \land\\
&\qquad\quad (\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph' \Vert\!\restriction_{tref} t2}\} \bullet\\
&\qquad\qquad \mathrm{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin\\
&\qquad\qquad\qquad\qquad \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}\\
&\qquad\qquad \lor
\end{aligned}
$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \|\!\restriction_{tref}\, t2 \bullet chan_of(e) = chan_of(e') \wedge (chan_of(e) \in \mathsf{ran}\, TermOf \vee chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e'))))))$$

$\vdash [A \wedge B \vee C \wedge B \equiv (A \vee C) \wedge B]$

$$\exists h^* : HS \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) i) = time_of(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) j)) \wedge$$
$$\text{strip}((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, h^* \wedge$$
$$(\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet$$
$$\text{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) = traces_Z\, prefix \wedge$$
$$(\forall e : \{a : Event \mid (t, a) \in \aleph' \|\!\restriction_{tref}\, t2\} \bullet$$
$$\text{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \|\!\restriction_{tref}\, t2 \bullet chan_of(e) = chan_of(e') \wedge (chan_of(e) \in \mathsf{ran}\, TermOf \vee chan_of(e) \in \mathsf{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e'))))))$$

$\vdash$ [def $failures_Z$]

$$\exists h^* : HS \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of((s \restriction_{ttr} t2 \frown \langle (t2, a) \rangle) i) = time_of((s \restriction_{ttr} t2 \frown \langle (t2, a) \rangle) j)) \wedge$$
$$\text{strip}((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, h^* \wedge$$
$$(\forall t : \mathbb{T} \bullet \exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet$$
$$(\text{strip}(((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) \restriction_{ttr} t), \{e : Event \mid (t, e) \in \aleph' \|\!\restriction_{tref}\, t2\}) \in failures_Z(\!|\{(prefix, HS)\}|\!))$$

$\vdash$ [def. $timedfailures_Z$]

$$((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle, \aleph' \|\!\restriction_{tref}\, t2) \in \bigcup \{h : HS \bullet timedfailures_Z(\!|\{(h, HS)\}|\!)\})$$

□

ad 4.

$$t0 >_{\mathbb{R}} 0_{\mathbb{R}} \wedge \neg \exists \epsilon : \mathbb{T}^+ \bullet [t0 -_{\mathbb{R}} \epsilon, t0) \times \{a\} \subseteq \aleph'$$

$\vdash$

$\forall t1 : [0_{\mathbb{R}}, t0) \bullet \exists t2 : (t1, t0) \bullet (t2, a) \notin \aleph'$

$\aleph' \in \mathit{TimedRefusal}$

$\vdash$ [def. *TimedRefusal*: union of finite set of refusal tokens]

$\exists t1 : [0_{\mathbb{R}}, t0) \bullet \forall t : (t1, t0) \bullet \{e : \mathit{Event} \mid (t, e) \in \aleph'\} = \{e : \mathit{Event} \mid (t1, e) \in \aleph'\}$

$s \in \mathit{TimedTrace}$

$\vdash$ [def. *TimedTrace*: infinite timed traces must not be bounded in time]

$\exists t1 : [0_{\mathbb{R}}, t0) \bullet s \uparrow [t1, t0) = \langle\rangle$

Let $t1 : [0_{\mathbb{R}}, t0)$ such that $\forall t : (t1, t0) \bullet \{e : \mathit{Event} \mid (t, e) \in \aleph'\} = \{e : \mathit{Event} \mid (t1, e) \in \aleph'\} \wedge s \uparrow [t1, t0) = \langle\rangle$. Let $t2 : (t1, t0)$ such that $(t2, a) \notin \aleph'$.

$(t2, a) \notin \aleph'$

$\vdash$ [Lemma B.7]

$$\mathrm{strip}(s \restriction_{ttr} t2) \frown \langle a\rangle \in \{\mathit{pref}, \mathit{ext} : \mathit{HS} \mid \mathrm{strip}(s \restriction_{ttr} t2) = \mathit{traces}_Z\, \mathit{pref} \wedge (\mathit{pref}, h) \in \mathit{PreHist} \wedge (\mathit{pref}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{traces}_Z\, \mathit{ext}\}$$

$\vdash$ [Lemma B.5]

$$\exists h^* : \{\mathit{pref}, \mathit{ext} : \mathit{HS} \mid \mathrm{strip}(s \restriction_{ttr} t2) = \mathit{traces}_Z\, \mathit{pref} \wedge (\mathit{pref}, h) \in \mathit{PreHist} \wedge (\mathit{pref}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{ext}\} \mid \mathit{simultan_of}\, h^* = \mathit{simultan_of}\, h \cap \{i, j : \mathbb{N} \mid \mathit{time_of}(s\, i) <_{\mathbb{R}} t2 \wedge \mathit{time_of}(s\, j) <_{\mathbb{R}} t2\} \bullet \mathrm{strip}(s \restriction_{ttr} t2) \frown \langle a\rangle = \mathit{traces}_Z\, h^*$$

$\vdash [s \uparrow_{ttr} [t2, t0) = \langle\rangle]$

$$\exists h^* : \{\mathit{pref}, \mathit{ext} : \mathit{HS} \mid \mathrm{strip}(s \restriction_{ttr} t2) = \mathit{traces}_Z\, \mathit{pref} \wedge (\mathit{pref}, h) \in \mathit{PreHist} \wedge (\mathit{pref}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{ext}\} \mid \mathit{simultan_of}\, h^* = \mathit{simultan_of}\, h \cap \{i, j : \mathbb{N} \mid \mathit{time_of}(s\, i) <_{\mathbb{R}} t0 \wedge \mathit{time_of}(s\, j) <_{\mathbb{R}} t0\} \bullet \mathrm{strip}(s \restriction_{ttr} t2) \frown \langle a\rangle) = \mathit{traces}_Z\, h^*$$

$\vdash [(s, \aleph) \in \mathit{timedfailures}_Z (\!| \{(h, \mathit{HS})\} |\!);$
$\forall i : \mathrm{dom}\, s \mid \mathit{time_of}(s\, i) <_{\mathbb{R}} t0 \bullet ((s \Vert\!\restriction_{ttr} t0) \frown \langle(t0, a)\rangle)\, i = s\, i]$

$$\exists h^* : \{\mathit{pref}, \mathit{ext} : \mathit{HS} \mid \mathrm{strip}(s \restriction_{ttr} t2) = \mathit{traces}_Z\, \mathit{pref} \wedge (\mathit{pref}, h) \in \mathit{PreHist} \wedge (\mathit{pref}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{ext}\} \bullet (\forall i, j : \mathbb{N} \mid (i, j) \in \mathit{simultan_of}\, h^* \bullet \mathit{time_of}(((s \Vert\!\restriction_{ttr} t0) \frown \langle(t0, a)\rangle)\, i) = \mathit{time_of}(((s \Vert\!\restriction_{ttr} t0) \frown \langle(t0, a)\rangle)\, j)) \wedge$$

$$\text{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^*$$

$$\vdash [t0 >_{\mathbb{R}} t2]$$

$$\exists h^* : \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref \land (pref, h) \in PreHist \land (pref, ext) \in PreHist \bullet ext\} \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of((s \Vert_{ttr} t0 \frown \langle (t0, a) \rangle)\, i) = time_of((s \Vert_{ttr} t0 \frown \langle (t0, a) \rangle)\, j)) \land$$
$$\text{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^* \land$$
$$\forall t : \mathbb{T} \bullet t <_{\mathbb{R}} t2 \lor t2 \leqslant_{\mathbb{R}} t <_{\mathbb{R}} t0 \lor t \geqslant_{\mathbb{R}} t0$$

$$\vdash [(h^*, h^*) \in PreHist;\ \forall e : \varnothing \bullet \textsf{false};\ (s, \aleph') \in timedfailures_Z (\!|\{(h, HS)\}|\!)]$$

$$\exists h^* : \{pref, ext : HS \mid \text{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, pref \land (pref, h) \in PreHist \land (pref, ext) \in PreHist \bullet ext\} \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \Vert_{ttr} t0) \frown \langle (t0, a) \rangle)\, i) = time_of(((s \Vert_{ttr} t0) \frown \langle (t0, a) \rangle)\, j)) \land$$
$$\text{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = traces_Z\, h^* \land$$
$$(\forall t : \mathbb{T} \bullet$$
$$t \geqslant_{\mathbb{R}} t0 \land$$
$$(\exists prefix : HS \mid \boxed{(prefix, h^*) \in PreHist} \bullet$$
$$\text{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, prefix \land$$
$$(\forall e : \boxed{\varnothing} \bullet$$
$$\text{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\lor$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \Vert_{tref} t0 \bullet chan_of(e) = chan_of(e') \land$$
$$(chan_of(e) \in \textsf{ran}\, TermOf$$
$$\lor chan_of(e) \in \textsf{ran}\, ExecOf \land Inputs \lhd (val_of\, e) = Inputs \lhd (val_of\, e')))))$$
$$\lor$$
$$t2 \leqslant_{\mathbb{R}} t <_{\mathbb{R}} t0 \land$$

$$
\boxed{\begin{array}{l}
(\exists \mathit{prefix} : \mathit{HS} \mid (\mathit{prefix}, h) \in \mathit{PreHist} \bullet \\
\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) = \mathit{traces}_Z\, \mathit{prefix} \land \\
\quad (\forall e : \{a : \mathit{Event} \mid (t2, a) \in \aleph'\} \bullet \\
\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle e \rangle \notin \{\mathit{ext} : \mathit{HS} \mid (\mathit{prefix}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{traces}_Z\, \mathit{ext}\} \\
\quad\quad \lor \\
\quad\quad (\exists e' : \mathit{Event} \mid (t, e') \notin \aleph' \bullet \mathit{chan_of}(e) = \mathit{chan_of}(e') \land \\
\quad\quad\quad (\mathit{chan_of}(e) \in \mathsf{ran}\, \mathit{TermOf} \\
\quad\quad\quad \lor \mathit{chan_of}(e) \in \mathsf{ran}\, \mathit{ExecOf} \\
\quad\quad\quad\quad \land \mathit{Inputs} \lhd (\mathit{val_of}\ e) = \mathit{Inputs} \lhd (\mathit{val_of}\ e')))))
\end{array}}
$$

$\lor$

$t <_{\mathbb{R}} t2 \land$

$$
\boxed{\begin{array}{l}
(\exists \mathit{prefix} : \mathit{HS} \mid (\mathit{prefix}, h) \in \mathit{PreHist} \bullet \\
\quad \mathrm{strip}(s \upharpoonright_{ttr} t) = \mathit{traces}_Z\, \mathit{prefix} \land \\
\quad (\forall e : \{a : \mathit{Event} \mid (t, a) \in \aleph'\} \bullet \\
\quad\quad \mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{\mathit{ext} : \mathit{HS} \mid (\mathit{prefix}, \mathit{ext}) \in \mathit{PreHist} \bullet \mathit{traces}_Z\, \mathit{ext}\} \\
\quad\quad \lor \\
\quad\quad (\exists e' : \mathit{Event} \mid (t, e') \notin \aleph' \bullet \mathit{chan_of}(e) = \mathit{chan_of}(e') \land \\
\quad\quad\quad (\mathit{chan_of}(e) \in \mathsf{ran}\, \mathit{TermOf} \\
\quad\quad\quad \lor \mathit{chan_of}(e) \in \mathsf{ran}\, \mathit{ExecOf} \\
\quad\quad\quad\quad \land \mathit{Inputs} \lhd (\mathit{val_of}\ e) = \mathit{Inputs} \lhd (\mathit{val_of}\ e')))))
\end{array}}
$$

$)$

$\vdash [\exists x : S \bullet P(x) \land S \subseteq T \Rightarrow \exists x : T \bullet P(x);\ \text{Lemma B.3}]$

$$
\begin{array}{l}
\exists h^* : \boxed{\mathit{HS}} \bullet \\
\quad (\forall i, j : \mathbb{N} \mid (i, j) \in \mathit{simultan_of}\ h^* \bullet \mathit{time_of}(((s \Vert_{ttr} t0) \frown \langle (t0, a) \rangle)\, i) = \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{time_of}(((s \Vert_{ttr} t0) \frown \langle (t0, a) \rangle)\, j)) \land \\
\quad \mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle a \rangle = \mathit{traces}_Z\, h^* \land \\
\quad (\forall t : \mathbb{T} \bullet \\
\quad\quad t \geqslant_{\mathbb{R}} t0 \land \\
\quad\quad (\exists \mathit{prefix} : \mathit{HS} \mid (\mathit{prefix}, h^*) \in \mathit{PreHist} \bullet \\
\quad\quad\quad \mathrm{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) = \mathit{traces}_Z\, \mathit{prefix} \land \\
\quad\quad\quad (\forall e : \varnothing \bullet \\
\quad\quad\quad\quad \mathrm{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) \frown \langle e \rangle \notin
\end{array}
$$

$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$

$$\vee$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \,\|\!\upharpoonright_{tref}\, t0 \bullet chan_of(e) = chan_of(e') \wedge$$

$$(chan_of(e) \in \mathrm{ran}\, TermOf$$

$$\vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\; e) =$$

$$Inputs \lhd (val_of\; e')))))$$

$$\vee$$

$$t2 \leqslant_{\mathbb{R}} t <_{\mathbb{R}} t0 \wedge$$

$$(\exists prefix : HS \mid (prefix, \boxed{h^*}) \in PreHist \bullet$$

$$\mathrm{strip}(s \upharpoonright_{ttr} t2) = traces_Z\, prefix \wedge$$

$$(\forall e : \{a : Event \mid (t2, a) \in \aleph'\} \bullet$$

$$\mathrm{strip}(s \upharpoonright_{ttr} t2) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$

$$\vee$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge$$

$$(chan_of(e) \in \mathrm{ran}\, TermOf$$

$$\vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\; e) =$$

$$Inputs \lhd (val_of\; e')))))$$

$$\vee$$

$$t <_{\mathbb{R}} t2 \wedge$$

$$(\exists prefix : HS \mid (prefix, \boxed{h^*}) \in PreHist \bullet$$

$$\mathrm{strip}(s \upharpoonright_{ttr} t) = traces_Z\, prefix \wedge$$

$$(\forall e : \{a : Event \mid (t, a) \in \aleph'\} \bullet$$

$$\mathrm{strip}(s \upharpoonright_{ttr} t) \frown \langle e \rangle \notin \{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$

$$\vee$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \bullet chan_of(e) = chan_of(e') \wedge$$

$$(chan_of(e) \in \mathrm{ran}\, TermOf$$

$$\vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\; e) =$$

$$Inputs \lhd (val_of\; e'))))))$$

$$\vdash [t \geqslant_{\mathbb{R}} t0 \Rightarrow$$

$$((s \,\|\!\upharpoonright_{ttr}\, t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t = (s \,\|\!\upharpoonright_{ttr}\, t0) \frown \langle (t0, a) \rangle = (s \upharpoonright_{ttr} t2) \frown \langle (t0, a) \rangle \wedge$$

$$\{e : Event \mid (t, e) \in \aleph' \,\|\!\upharpoonright_{tref}\, t0\} = \varnothing;$$

$$\mathrm{strip}(((s \,\|\!\upharpoonright_{ttr}\, t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t) = \mathrm{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle)$$

$$t2 \leqslant_{\mathbb{R}} t <_{\mathbb{R}} t0 \Rightarrow$$

$$((s \,\|\!\upharpoonright_{ttr}\, t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t = s \upharpoonright_{ttr} t = s \upharpoonright_{ttr} t2 \wedge$$

$$\{e : Event \mid (t2, e) \in \aleph'\} = \{e : Event \mid (t, e) \in \aleph'\} =$$

$$\{e : Event \mid (t, e) \in \aleph' \,\|\!\upharpoonright_{tref}\, t0\}$$

$$t <_{\mathbb{R}} t2 (<_{\mathbb{R}} t0) \Rightarrow$$
$$((s \upharpoonleft\!\!|_{ttr}\ t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t = s \upharpoonright_{ttr} t \wedge$$
$$\{e : Event \mid (t, e) \in \aleph' \upharpoonleft\!\!|_{tref}\ t0\} = \{e : Event \mid (t, e) \in \aleph'\}]$$

$$\exists h^{*} : HS \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h^{*} \bullet time_of((s \upharpoonleft\!\!|_{ttr}\ t0 \frown \langle (t0, a) \rangle)\ i) =$$
$$time_of((s \upharpoonleft\!\!|_{ttr}\ t0 \frown \langle (t0, a) \rangle)\ j)) \wedge$$
$$\mathrm{strip}((s \upharpoonright_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\ h^{*} \wedge$$
$$(\forall t : \mathbb{T} \bullet$$
$$t \geqslant_{\mathbb{R}} t0 \wedge$$
$$(\exists prefix : HS \mid (prefix, h^{*}) \in PreHist \bullet$$
$$\mathrm{strip}(\boxed{((s \upharpoonleft\!\!|_{ttr}\ t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t}) = traces_Z\ prefix \wedge$$
$$(\forall e : \{e : Event \mid (t, e) \in \boxed{\aleph' \upharpoonleft\!\!|_{tref}\ t0}\} \bullet$$
$$\mathrm{strip}(((s \upharpoonleft\!\!|_{ttr}\ t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t) \frown \langle e \rangle \notin$$
$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\ ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \upharpoonleft\!\!|_{tref}\ t0 \bullet chan_of(e) = chan_of(e') \wedge$$
$$(chan_of(e) \in \mathrm{ran}\ TermOf$$
$$\vee\ chan_of(e) \in \mathrm{ran}\ ExecOf \wedge Inputs \lhd (val_of\ e) =$$
$$Inputs \lhd (val_of\ e')))))$$
$$\vee$$
$$t2 \leqslant_{\mathbb{R}} t <_{\mathbb{R}} t0 \wedge$$
$$(\exists prefix : HS \mid (prefix, h^{*}) \in PreHist \bullet$$
$$\mathrm{strip}(\boxed{((s \upharpoonleft\!\!|_{ttr}\ t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t}) = traces_Z\ prefix \wedge$$
$$(\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph' \upharpoonleft\!\!|_{tref}\ t0}\} \bullet$$
$$\mathrm{strip}(((s \upharpoonleft\!\!|_{ttr}\ t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t) \frown \langle e \rangle \notin$$
$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\ ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \upharpoonleft\!\!|_{tref}\ t0 \bullet chan_of(e) = chan_of(e') \wedge$$
$$(chan_of(e) \in \mathrm{ran}\ TermOf$$
$$\vee\ chan_of(e) \in \mathrm{ran}\ ExecOf \wedge Inputs \lhd (val_of\ e) =$$
$$Inputs \lhd (val_of\ e')))))$$
$$\vee$$
$$t <_{\mathbb{R}} t2 \wedge$$
$$(\exists prefix : HS \mid (prefix, h^{*}) \in PreHist \bullet$$

$$\mathrm{strip}(\boxed{((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t}) = traces_Z\, prefix \wedge$$

$$(\forall e : \{a : Event \mid (t, a) \in \boxed{\aleph' \parallel\!\restriction_{tref} t0}\} \bullet$$

$$\mathrm{strip}(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin$$
$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \parallel\!\restriction_{tref} t0 \bullet chan_of(e) = chan_of(e') \wedge$$
$$(chan_of(e) \in \mathrm{ran}\, TermOf$$
$$\vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) =$$
$$Inputs \lhd (val_of\, e'))))))$$

$$\vdash [A \wedge D \vee B \wedge D \vee C \wedge D \equiv (A \vee B \vee C) \wedge D]$$

$$\exists h^* : HS \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle)\, i) =$$
$$time_of(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle)\, j)) \wedge$$
$$\mathrm{strip}((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) = traces_Z\, h^* \wedge$$
$$(\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet$$
$$\mathrm{strip}(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t) = traces_Z\, prefix \wedge$$
$$(\forall e : \{a : Event \mid (t, a) \in \aleph' \parallel\!\restriction_{tref} t0\} \bullet$$
$$\mathrm{strip}(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin$$
$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$
$$\vee$$
$$(\exists e' : Event \mid (t, e') \notin \aleph' \parallel\!\restriction_{tref} t0 \bullet chan_of(e) = chan_of(e') \wedge$$
$$(chan_of(e) \in \mathrm{ran}\, TermOf$$
$$\vee\, chan_of(e) \in \mathrm{ran}\, ExecOf \wedge Inputs \lhd (val_of\, e) =$$
$$Inputs \lhd (val_of\, e'))))))$$

$$\vdash [s \restriction_{ttr} t2 = s \parallel\!\restriction_{ttr} t0;\ \mathrm{strip}((s \restriction_{ttr} t2) \frown \langle (t2, a) \rangle) = \mathrm{strip}((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle)]$$

$$\exists h^* : HS \bullet$$
$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\, h^* \bullet time_of(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle)\, i) =$$
$$time_of(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle)\, j)) \wedge$$
$$\mathrm{strip}(\boxed{(s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle}) = traces_Z\, h^* \wedge$$
$$(\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet$$
$$\mathrm{strip}(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t) = traces_Z\, prefix \wedge$$
$$(\forall e : \{a : Event \mid (t, a) \in \aleph' \parallel\!\restriction_{tref} t0\} \bullet$$
$$\mathrm{strip}(((s \parallel\!\restriction_{ttr} t0) \frown \langle (t0, a) \rangle) \restriction_{ttr} t) \frown \langle e \rangle \notin$$

$$\{ext : HS \mid (prefix, ext) \in PreHist \bullet traces_Z\, ext\}$$

$$\vee$$

$$(\exists e' : Event \mid (t, e') \notin \aleph' \upharpoonright_{tref} t0 \bullet chan_of(e) = chan_of(e') \wedge (chan_of(e) \in \operatorname{ran} TermOf \vee chan_of(e) \in \operatorname{ran} ExecOf \wedge Inputs \lhd (val_of\ e) = Inputs \lhd (val_of\ e'))))))$$

$\vdash$ [def. $failures_Z$]

$$\exists h^* : HS \bullet$$

$$(\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h^* \bullet time_of(((s \upharpoonright_{ttr} t0) \frown \langle (t0, a) \rangle)\, i) = time_of(((s \upharpoonright_{ttr} t0) \frown \langle (t0, a) \rangle)\, j)) \wedge$$

$$\mathrm{strip}((s \upharpoonright_{ttr} t0) \frown \langle (t0, a) \rangle) = traces_Z\, h^* \wedge$$

$$(\forall t : \mathbb{T} \bullet (\exists prefix : HS \mid (prefix, h^*) \in PreHist \bullet (\mathrm{strip}(((s \upharpoonright_{ttr} t0) \frown \langle (t0, a) \rangle) \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph' \upharpoonright_{tref} t0\}) \in failures_Z(\!|\{(prefix, HS)\}|\!)))$$

$\vdash$ [def. $timed failures_Z$]

$$((s \upharpoonright_{ttr} t0) \frown \langle (t0, a) \rangle, \aleph' \upharpoonright_{tref} t0) \in \bigcup\{h : HS \bullet timed failures_Z(\!|\{(h, HS)\}|\!)\})$$

□

Appendix C

Relationship between Simulation and Refinement in the Timed Failures/States Model

In this appendix we prove some lemmata that we needed in Chapter 10 in order to establish that forward and backward simulations between concrete specification units imply refinement in the timed failures/states model.

Lemma C.1

$$\frac{tr = traces_Z\, h}{\begin{array}{l} tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \bullet traces_Z\, ext\} \\ \Leftrightarrow \\ tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \wedge \\ \qquad \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\, ext\} \end{array}}$$

We need the following lemmata to prove the current lemma.

Lemma C.2
If trace tr is associated with history h, then trace $tr \frown \langle e \rangle$ must be associated with some history ext whose event sequence component extends that of h by a single operation application.

$$\frac{tr = traces_Z\, h}{tr \frown \langle e \rangle \in \{ext : History \mid \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\, ext\}}$$

Lemma C.3

$$\frac{x \in A}{x \notin B \Leftrightarrow x \notin B \cap A}$$

Lemma C.4

The following rule formalises the condition under which an intersection operator (or logical conjunction operator) inside a relational image can be distributed to the outside.

$$\frac{R(\!| B \setminus A |\!) \cap R(\!| A |\!) = \varnothing}{R(\!| A |\!) \cap R(\!| B |\!) = R(\!| A \cap B |\!)}$$

Lemma C.5

The conclusion of the following lemma matches the antecedent of the previous one.

$$\begin{array}{l} \{ext : S \mid (h, ext) \in PreHist \land \\ \qquad \neg\ \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\ ext\} \\ \cap\ \{ext : S \mid \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\ ext\} \\ = \varnothing \end{array}$$

It is valid because the lengths of the traces on the left hand side of the intersection operator are different from the lengths of the traces on the right hand side.

Proof of Lemma C.1.

$$tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \bullet traces_Z\ ext\}$$

$\Leftrightarrow$ [Lemmata C.2, C.3]

$$\begin{array}{l} tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \bullet traces_Z\ ext\} \\ \qquad \cap\ \{ext : History \mid \exists\, op : POP_APP \bullet \\ \qquad\qquad events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\ ext\} \end{array}$$

$\Leftrightarrow$ [Lemmata C.4, C.5; $\{x : S \bullet P\} \cap \{x : S \bullet Q\} = \{x : S \bullet P \land Q\}$]

$$\begin{array}{l} tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \land \\ \qquad \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\ ext\} \end{array}$$

□

Lemma C.6

The following lemma is an immediate consequence of the definition of $traces_Z$: a trace derived from a history is simply the translation (ev_of_op) of its sequence of operation applications.

$$\frac{tr = traces_Z\ h}{\begin{array}{l} tr \frown \langle e \rangle \notin \{ext : S \mid (h, ext) \in PreHist \bullet \\ \qquad \exists\, op : POP_APP \bullet events_of\ ext = (events_of\ h) \frown \langle op \rangle \bullet traces_Z\ ext\} \\ \Leftrightarrow \\ e \notin ev_of_op(\!| \{op : POP_APP \mid \exists\, ext : S \mid (h, ext) \in PreHist \bullet \\ \qquad events_of\ ext = (events_of\ h) \frown \langle op \rangle\} |\!) \end{array}}$$

Proof of Theorem 10.4.

Let $h_c \in HSC$ and $h_a \in HSA$ such that $(h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)$. Let further be $D2$ the second disjunct within the universal quantifier of the definition of $failures_Z$ in Section 9.2.4. Note that $D2$ is independent of the history h for which the set of failures is computed. We do thus not need to consider $D2$ in our proof.

$(tr_c, X_c) \in failures_Z(\!|\{(h_c, HSC)\}|\!)$

$\vdash$ [def. $failures_Z$]

$tr_c = traces_Z\, h_c \wedge$
$\forall e : X_c \bullet$
$\quad tr_c \frown \langle e \rangle \notin \{ext : HSC \mid (h_c, ext) \in PreHist \bullet traces_Z\, ext\}$
$\quad \vee D2$

$\vdash$ [Lemma C.1]

$tr_c = traces_Z\, h_c \wedge$
$\forall e : X_c \bullet$
$\quad tr_c \frown \langle e \rangle \notin \{ext : HSC \mid (h_c, ext) \in PreHist \wedge$
$\qquad \exists op : POP_APP \bullet events_of\, ext = (events_of\, h_c) \frown \langle op \rangle \bullet traces_Z\, ext\}$
$\quad \vee D2$

$\vdash$ [Lemma C.6]

$tr_c = traces_Z\, h_c \wedge$
$\forall e : X_c \bullet$
$\quad e \notin ev_of_op(\!|\{op : POP_APP \mid \exists ext : HSC \mid (h_c, ext) \in PreHist \bullet$
$\qquad events_of\, ext = (events_of\, h_c) \frown \langle op \rangle\}|\!)$
$\quad \vee D2$

$\vdash$ [Theorem 10.2]

$tr_c = traces_Z\, h_c \wedge$
$\forall e : X_c \bullet$
$\quad e \notin Retr_{ev}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|ev_of_op(\!|\{op : POP_APP \mid \exists ext : HSA \mid (h_a, ext) \in PreHist \wedge$
$\qquad events_of\, ext = (events_of\, h_a) \frown \langle op \rangle\}|\!)|\!)$
$\quad \vee D2$

$\vdash [(\forall e : X_c \bullet e \notin R(\!|Y|\!)) \Rightarrow (\forall e : R^{\sim}(\!|X_c|\!) \bullet e \notin Y)]$

$tr_c = traces_Z\, h_c \wedge$
$\forall e : Retr_{ev}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)^{\sim}(\!|X_c|\!) \bullet$
$\quad e \notin ev_of_op(\!|\{op : POP_APP \mid \exists ext : HSA \mid (h_a, ext) \in PreHist \wedge$
$\qquad events_of\, ext = (events_of\, h_a) \frown \langle op \rangle\}|\!)$
$\quad \vee D2$

$\vdash$ [Theorem 10.3]

$tr_c \in Retr_{tr}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{traces_Z\, h_a\}|\!) \wedge$
$\quad \forall\, e : Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)^\sim(\!|X_c|\!) \bullet$
$\quad\quad e \notin ev_of_op(\!|\{op : POP_APP \mid \exists\, ext : HSA \mid (h_a, ext) \in PreHist \wedge$
$\quad\quad\quad events_of\, ext = (events_of\, h_a) \frown \langle op \rangle\}|\!)$
$\quad\quad \vee\, D2$

$\vdash$ [Lemma C.6]

$tr_c \in Retr_{tr}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{traces_Z\, h_a\}|\!) \wedge$
$\quad \forall\, e : Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)^\sim(\!|X_c|\!) \bullet$
$\quad\quad (traces_Z\, h_a) \frown \langle e \rangle \notin \{ext : HSA \mid (h_a, ext) \in PreHist \wedge$
$\quad\quad\quad \exists\, op : POP_APP \bullet events_of\, ext = (events_of\, h_a) \frown \langle op \rangle \bullet traces_Z\, ext\}$
$\quad\quad \vee\, D2$

$\vdash$ [Lemma C.1]

$tr_c \in Retr_{tr}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{traces_Z\, h_a\}|\!) \wedge$
$\quad \forall\, e : Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)^\sim(\!|X_c|\!) \bullet$
$\quad\quad (traces_Z\, h_a) \frown \langle e \rangle \notin \{ext : HSA \mid (h_a, ext) \in PreHist \bullet traces_Z\, ext\}$
$\quad\quad \vee\, D2$

$\vdash$

$\exists\, tr_a : Trace;\ X_a : Refusal \mid$
$\quad tr_c \in Retr_{tr}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{tr_a\}|\!) \wedge X_a = Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)^\sim(\!|X_c|\!) \bullet$
$\quad tr_a = traces_Z\, h_a \wedge$
$\quad \forall\, e : X_a \bullet$
$\quad\quad tr_a \frown \langle e \rangle \notin \{ext : HSA \mid (h_a, ext) \in PreHist \bullet traces_Z\, ext\}$
$\quad\quad \vee\, D2$

$\vdash$ [def. $failures_Z$; $D2$ is independent of history; $X \subseteq R(\!|R^\sim(\!|X|\!)|\!)$]

$\exists\, tr_a : Trace;\ X_a : Refusal \mid$
$\quad (tr_a, X_a) \in failures_Z(\!|\{(h_a, HSA)\}|\!) \bullet$
$\quad tr_c \in Retr_{tr}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{tr_a\}|\!) \wedge X_c \subseteq Retr_{ev}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|X_a|\!)$

$\vdash$ [def. $Retr_{fail}$]

$(tr_c, X_c) \in Retr_{fail}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|failures_Z(\!|\{(h_a, HSA)\}|\!)|\!)$

$\square$

Lemma C.7
The following lemma states that the component simultan_of of a history does not influence its membership to the set of histories associated with a specification unit: two histories differing

only in the component simultan_of are either both members or both no members.

$$\frac{h \in Histories(\!|\{(\mathcal{M}, OPS)\}|\!) \quad events_of\ h = events_of\ h^* \wedge states_of\ h = states_of\ h^* \wedge ops_of\ h = ops_of\ h^*}{h^* \in Histories(\!|\{(\mathcal{M}, OPS)\}|\!)}$$

Lemma C.8

If two histories h_c and h_a are related by $Retr_{hist}$, then for each prefix of h_c there is a prefix of h_a such that these prefixes are related by $Retr_{hist}$.

$$\frac{h_c \in HSC;\ h_a \in HSA \quad (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)}{\forall\, pref_c : HSC \mid (pref_c, h_c) \in PreHist \bullet \exists\, pref_a : HSA \mid (pref_a, h_a) \in PreHist \bullet (pref_c, pref_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)}$$

Proof of Theorem 10.5.

Let $h_c \in HSC$ be arbitrary.

$(s_c, \aleph_c) \in timedfailures_Z(\!|\{(h_c, HSC)\}|\!)$

$\vdash$ [def. $timedfailures_Z$]

$(\forall\, i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h_c \bullet time_of(s_c\ i) = time_of(s_c\ j)) \wedge$

$\text{strip}\ s_c = traces_Z\ h_c \wedge$

$\forall\, t : \mathbb{T} \bullet \exists\, pref_c : HSC \mid (pref_c, h_c) \in PreHist \bullet$

$\quad (\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\}) \in failures_Z(\!|\{(pref_c, HSC)\}|\!)$

$\vdash$ [Theorem 10.2]

$\exists\, h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$

$\quad \text{strip}\ s_c = traces_Z\ h_c \wedge$

$\quad \forall\, t : \mathbb{T} \bullet \exists\, pref_c : HSC \mid (pref_c, h_c) \in PreHist \bullet$

$\quad\quad (\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\}) \in failures_Z(\!|\{(pref_c, HSC)\}|\!)$

$\vdash$ [Lemma C.8]

$\exists\, h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$

$\quad \text{strip}\ s_c = traces_Z\ h_c \wedge$

$\quad \forall\, t : \mathbb{T} \bullet \exists\, pref_c : HSC \mid (pref_c, h_c) \in PreHist \bullet$

$\quad\quad \exists\, pref_a : HSA \mid (pref_a, h_a) \in PreHist \wedge (pref_c, pref_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$

$\quad\quad\quad (\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\}) \in failures_Z(\!|\{(pref_c, HSC)\}|\!)$

$\vdash$ [Theorem 10.4]

$\exists\, h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$

$\quad \text{strip}\ s_c = traces_Z\ h_c$

$\quad \forall\, t : \mathbb{T} \bullet \exists\, pref_c : HSC \mid (pref_c, h_c) \in PreHist \bullet$

$$\exists\, pref_a : HSA \mid (pref_a, h_a) \in PreHist \land (pref_c, pref_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$(\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph_c\}) \in$$
$$Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|failures_Z(\!|\{(pref_a, HSA)\}|\!)|\!)$$

$\vdash$ [Theorem 10.3]

$$\exists\, h_a : HSA \mid (h_c, h_a) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$\text{strip } s_c \in Retr_{tr}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{traces_Z\, h_a\}|\!) \land$$
$$\forall\, t : \mathbb{T} \bullet \exists\, pref_a : HSA \mid (pref_a, h_a) \in PreHist \bullet$$
$$(\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph_c\}) \in$$
$$Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|failures_Z(\!|\{(pref_a, HSA)\}|\!)|\!)$$

$\vdash$ [Lemma C.7]

$$\exists\, h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$(\forall\, i,j : \mathbb{N} \mid (i,j) \in simultan_of\ h_a^* \bullet time_of(s_c\, i) = time_of(s_c\, j)) \land$$
$$\text{strip } s_c \in Retr_{tr}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{traces_Z\, h_a^*\}|\!)$$
$$\forall\, t : \mathbb{T} \bullet \exists\, pref_a : HSA \mid (pref_a, h_a^*) \in PreHist \bullet$$
$$(\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph_c\}) \in$$
$$Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|failures_Z(\!|\{(pref_a, HSA)\}|\!)|\!)$$

$\vdash$

$$\exists\, h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$(\forall\, i,j : \mathbb{N} \mid (i,j) \in simultan_of\ h_a^* \bullet time_of(s_c\, i) = time_of(s_c\, j)) \land$$
$$\text{strip } s_c \in Retr_{tr}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{traces_Z\, h_a^*\}|\!)$$
$$\forall\, t : \mathbb{T} \bullet \exists\, pref_a : HSA \mid (pref_a, h_a^*) \in PreHist \bullet$$
$$\exists\, tr_a^* : Trace;\ X_a^* : Refusal \mid (tr_a^*, X_a^*) \in failures_Z(\!|\{(pref_a, HSA)\}|\!) \bullet$$
$$(\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph_c\}) \in$$
$$Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{(tr_a^*, X_a^*)\}|\!)$$

$\vdash$

$$\exists\, h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$\exists\, s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$$
$$\mathsf{dom}\, s_a = \mathsf{dom}\, s_c \land (\forall\, i : \mathsf{dom}\, s_c \bullet time_of(s_c\, i) = time_of(s_a\, i)) \bullet$$
$$(\forall\, i,j : \mathbb{N} \mid (i,j) \in simultan_of\ h_a^* \bullet time_of(s_a\, i) = time_of(s_a\, j)) \land$$
$$\text{strip } s_c \in Retr_{tr}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{traces_Z\, h_a^*\}|\!) \land \text{strip } s_a = traces_Z\, h_a^* \land$$
$$\forall\, t : \mathbb{T} \bullet \exists\, pref_a : HSA \mid (pref_a, h_a^*) \in PreHist \bullet$$
$$\exists\, tr_a^* : Trace;\ X_a^* : Refusal \mid (tr_a^*, X_a^*) \in failures_Z(\!|\{(pref_a, HSA)\}|\!) \bullet$$
$$\text{strip}(s_a \upharpoonright_{ttr} t) = tr_a^* \land \{e : Event \mid (t,e) \in \aleph_a\} = X_a^* \land$$
$$(\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t,e) \in \aleph_c\}) \in Retr_{fail}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|\{(tr_a^*, X_a^*)\}|\!)$$

$\vdash$

$\exists h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$
$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$
$\quad \mathsf{dom}\, s_a = \mathsf{dom}\, s_c \wedge (\forall i : \mathsf{dom}\, s_c \bullet time_of(s_c\, i) = time_of(s_a\, i)) \bullet$
$\quad (\forall i, j : \mathbb{N} \mid (i, j) \in simultan_of\ h_a^* \bullet time_of(s_a\, i) = time_of(s_a\, j)) \wedge$
$\quad \text{strip}\ s_a = traces_Z\, h_a^* \wedge$
$\quad \forall t : \mathbb{T} \bullet \exists pref_a : HSA \mid (pref_a, h_a^*) \in PreHist \bullet$
$\quad\quad (\text{strip}(s_a \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_a\}) \in failures_Z(\!|\{(pref_a, HSA)\}|\!) \wedge$
$\quad\quad (\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\}) \in$
$\quad\quad\quad Retr_{fail}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{(\text{strip}(s_a \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_a\})\}|\!)$

$\vdash$ [def. $timed failures_Z$]

$\exists h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$
$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$
$\quad \mathsf{dom}\, s_a = \mathsf{dom}\, s_c \wedge (\forall i : \mathsf{dom}\, s_c \bullet time_of(s_c\, i) = time_of(s_a\, i)) \bullet$
$\quad (s_a, \aleph_a) \in timed failures_Z(\!|\{(h_a^*, HSA)\}|\!) \wedge$
$\quad \forall t : \mathbb{T} \bullet (\text{strip}(s_c \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_c\}) \in$
$\quad\quad\quad Retr_{fail}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{(\text{strip}(s_a \upharpoonright_{ttr} t), \{e : Event \mid (t, e) \in \aleph_a\})\}|\!) \wedge$

$\vdash$ [def. $Retr_{tf}$]

$\exists h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$
$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \bullet$
$\quad (s_c, \aleph_c) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|\{(s_a, \aleph_a)\}|\!) \wedge (s_a, \aleph_a) \in timed failures_Z(\!|\{(h_a^*, HSA)\}|\!)$

$\vdash$

$\exists h_a^* : HSA \mid (h_c, h_a^*) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$
$\quad (s_c, \aleph_c) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)(\!|timed failures_Z(\!|\{(h_a^*, HSA)\}|\!)|\!)$

□

Lemma C.9
The first lemma concerns the relation $timedst_of_stseq$*, which was defined in Chapter 9 in order to facilitate the definition of* $timed states_Z$*. For a pair* (h_c, h_a) *of histories related by* $Retr_{hist}$*, we can always find a timed state* tst_a *for each timed state* tst_c *computed by* $timedst_of_stseq$ *such that they are related by* $Retr_{tst}$*. We must simply choose the identical injective function* $index_to_time$ *for* tst_c *and* tst_a*.*

$$\frac{(h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)}{\forall tst_c : timedst_of_stseq(\!|\{h_c\}|\!) \bullet \exists tst_a : timedst_of_stseq(\!|\{h_a\}|\!) \bullet (tst_c, tst_a) \in Retr_{tst}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)}$$

Lemma C.10

For each pair of timed failures related by $Retr_{tf}$, the time instants recorded in their timed traces are identical.

$$\forall s_a, s_c : TimedTrace;\ \aleph_a, \aleph_c : TimedRefusal \mid ((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$$
$$\{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_c \bullet chan_of(ev_of(s_c\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_c\, i) = t\}$$
$$= \{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_a \bullet chan_of(ev_of(s_a\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_a\, i) = t\}$$

Proof of Theorem 10.6.

Let $(s_a, \aleph_a) \in TFA$ and $(s_c, \aleph_c) \in TFC$ be arbitrary such that $((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)$.

$tst_c \in timed\,states_Z(\!|\{h_c\}|\!)(\!|\{(s_c, \aleph_c)\}|\!)$

$\vdash$ [def. $timed\,states_Z$]

$tst_c \in timedst_of_stseq(\!|\{h_c\}|\!) \land$
$\mathrm{dom}\, tst_c = \{0_{\mathbb{R}}\} \cup \{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_c \bullet chan_of(ev_of(s_c\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_c\, i) = t\}$

$\vdash$ [Lemma C.9]

$\exists tst_a : timedst_of_stseq(\!|\{h_a\}|\!) \mid (tst_c, tst_a) \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$
$\mathrm{dom}\, tst_c = \{0_{\mathbb{R}}\} \cup \{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_c \bullet chan_of(ev_of(s_c\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_c\, i) = t\}$

$\vdash$ [def. $Retr_{tst}$]

$\exists tst_a : timedst_of_stseq(\!|\{h_a\}|\!) \mid (tst_c, tst_a) \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$
$\mathrm{dom}\, tst_a = \{0_{\mathbb{R}}\} \cup \{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_c \bullet chan_of(ev_of(s_c\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_c\, i) = t\}$

$\vdash$ [Lemma C.10]

$\exists tst_a : timedst_of_stseq(\!|\{h_a\}|\!) \mid (tst_c, tst_a) \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$
$\mathrm{dom}\, tst_a = \{0_{\mathbb{R}}\} \cup \{t : \mathbb{T} \mid \exists i : \mathrm{dom}\, s_a \bullet chan_of(ev_of(s_a\, i)) \in \mathrm{ran}\, TermOf \cup \mathrm{ran}\, ExecOf \land time_of(s_a\, i) = t\}$

$\vdash$ [def. $timed\,states_Z$]

$\exists tst_a : timedst_of_stseq(\!|\{h_a\}|\!) \mid (tst_c, tst_a) \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!) \bullet$
$tst_a \in timed\,states_Z(\!|\{h_a\}|\!)(\!|\{(s_a, \aleph_a)\}|\!)$

$\vdash$

$tst_c \in Retr_{tst}(\!|\{\mathcal{M}_{\mathcal{R}}\}|\!)(\!|timed\,states_Z(\!|\{h_a\}|\!)(\!|\{(s_a, \aleph_a)\}|\!)|\!)$

$\square$

Lemma C.11
The following lemma states that the relation $Retr_{tf}$ distributes over the parallel composition operator.

$$\frac{\begin{array}{l} ((s_z^a, \aleph_z^a), (s_z^c, \aleph_z^c)) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \\ ((s_{csp}^a, \aleph_{csp}^a), (s_{csp}^c, \aleph_{csp}^c)) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \\ ((s_z^c, s_{csp}^c), ORE) \text{ synch } s_c \\ \aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \\ ((s_z^a, s_{csp}^a), ORE) \text{ synch } s_a \\ \aleph_z^a \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^a \setminus (\mathbb{T} \times ORE^{\checkmark}) \end{array}}{((s_a, \aleph_z^a \cup \aleph_{csp}^a), (s_c, \aleph_z^c \cup \aleph_{csp}^c)) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!)}$$

Proof of Theorem 10.7.

$((s_c, \aleph_c), tst_c) \in tfs_of_model_{INT}(\!|\{(CSU_C, \mathcal{M}_\mathcal{C})\}|\!)$

$\vdash$ [def. $tfs_of_model_{INT}$]

...

$\exists\, h_c : HSC;\ s_z^c, s_{csp}^c : TimedTrace;\ \aleph_z^c, \aleph_{csp}^c : TimedRefusal \bullet$

$(s_z^c, \aleph_z^c) \in timed\,failures_Z(\!|\{(h_c, HSC)\}|\!) \wedge$

$((s_{csp}^c, \aleph_{csp}^c) \notin timed\,failures_{ETFM}\,[\![\,CSU_C.EA\,]\!]_{(\mathcal{M}_\mathcal{C}, CSU_C.I \cup CSU_C.L)}$

$\vee\ (s_{csp}^c, \aleph_{csp}^c) \in$

$timed\,failures_{ETCSP}\,[\![\,CSU_C.Behav\ Behaviour\,]\!]_{(CSU_C.Behav, (\mathcal{M}_\mathcal{C}, \mathcal{M}_\mathcal{C}), CSU_C.I \cup CSU_C.L)}) \wedge$

$((s_z^c, s_{csp}^c), ORE) \text{ synch } s_c \wedge$

$\aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge \aleph_c = \aleph_z^c \cup \aleph_{csp}^c \wedge$

$((s_z^c, \aleph_z^c), tst_c) \in timed\,states_Z(\!|\{h_c\}|\!)$

$\vdash$ [Theorem 10.5]

...

$\exists\, h_c : HSC;\ s_z^c, s_{csp}^c : TimedTrace;\ \aleph_z^c, \aleph_{csp}^c : TimedRefusal \bullet$

$\exists\, h_a : HSA;\ s_z^a : TimedTrace;\ \aleph_z^a : TimedRefusal \mid$

$(h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \wedge ((s_z^a, \aleph_z^a), (s_z^c, \aleph_z^c)) \in Retr_{tf}(\!|\{\mathcal{M}_\mathcal{R}\}|\!) \bullet$

$(s_z^a, \aleph_z^a) \in timed\,failures_Z(\!|\{(h_a, HSA)\}|\!) \wedge$

$((s_{csp}^c, \aleph_{csp}^c) \notin timed\,failures_{ETFM}\,[\![\,CSU_C.EA\,]\!]_{(\mathcal{M}_\mathcal{C}, CSU_C.I \cup CSU_C.L)} \wedge$

$\vee\ (s_{csp}^c, \aleph_{csp}^c) \in$

$timed\,failures_{ETCSP}\,[\![\,CSU_C.Behav\ Behaviour\,]\!]_{(CSU_C.Behav, (\mathcal{M}_\mathcal{C}, \mathcal{M}_\mathcal{C}), CSU_C.I \cup CSU_C.L)}) \wedge$

$((s_z^c, s_{csp}^c), ORE) \text{ synch } s_c \wedge$

$\aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge \aleph_c = \aleph_z^c \cup \aleph_{csp}^c \wedge$

$((s_z^c, \aleph_z^c), tst_c) \in timed\,states_Z(\!|\{h_c\}|\!)$

$\vdash$ [Theorem 10.6]

$\ldots$

$\exists h_c : HSC;\ s_z^c, s_{csp}^c : TimedTrace;\ \aleph_z^c, \aleph_{csp}^c : TimedRefusal \bullet$

$\exists h_a : HSA;\ s_z^a : TimedTrace;\ \aleph_z^a : TimedRefusal \mid (h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M_R}\}|\!) \wedge$

$\quad ((s_z^a, \aleph_z^a), (s_z^c, \aleph_z^c)) \in Retr_{tf}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists tst_a : TimedState \mid (tst_a, tst_c) \in Retr_{tst}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\quad (s_z^a, \aleph_z^a) \in timed\,failures_Z(\!|\{(h_a, HSA)\}|\!) \wedge$

$\quad ((s_{csp}^c, \aleph_{csp}^c) \notin timed\,failures_{ETFM} [\![CSU_C.EA]\!]_{(\mathcal{M_C}, CSU_C.I \cup CSU_C.L)} \wedge$

$\quad \vee (s_{csp}^c, \aleph_{csp}^c) \in$

$\quad timed\,failures_{ETCSP} [\![CSU_C.Behav\ Behaviour]\!]_{(CSU_C.Behav, (\mathcal{M_C}, \mathcal{M_C}), CSU_C.I \cup CSU_C.L)}) \wedge$

$\quad ((s_z^c, s_{csp}^c), ORE)\ \text{synch}\ s_c \wedge$

$\quad \aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge \aleph_c = \aleph_z^c \cup \aleph_{csp}^c \wedge$

$\quad tst_a \in timed\,states_Z(\!|\{h_a\}|\!)(\!|\{(s_z^a, \aleph_z^a)\}|\!)$

$\vdash$ [Assumptions 10.3 and 10.4]

$\ldots$

$\exists h_c : HSC;\ s_z^c, s_{csp}^c : TimedTrace;\ \aleph_z^c, \aleph_{csp}^c : TimedRefusal \bullet$

$\exists s_{csp}^a : TimedTrace;\ \aleph_{csp}^a : TimedRefusal \mid ((s_{csp}^a, \aleph_{csp}^a), (s_{csp}^c, \aleph_{csp}^c)) \in Retr_{tf}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists h_a : HSA;\ s_z^a : TimedTrace;\ \aleph_z^a : TimedRefusal \mid (h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M_R}\}|\!) \wedge$

$\quad ((s_z^a, \aleph_z^a), (s_z^c, \aleph_z^c)) \in Retr_{tf}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists tst_a : TimedState \mid (tst_a, tst_c) \in Retr_{tst}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\quad (s_z^a, \aleph_z^a) \in timed\,failures_Z(\!|\{(h_a, HSA)\}|\!) \wedge$

$\quad ((s_{csp}^a, \aleph_{csp}^a) \notin timed\,failures_{ETFM} [\![CSU_A.EA]\!]_{(\mathcal{M_A}, CSU_A.I \cup CSU_A.L)}$

$\quad \vee (s_{csp}^a, \aleph_{csp}^a) \in$

$\quad timed\,failures_{ETCSP} [\![CSU_A.Behav\ Behaviour]\!]_{(CSU_A.Behav, (\mathcal{M_A}, \mathcal{M_A}), CSU_A.I \cup CSU_A.L)}) \wedge$

$\quad ((s_z^c, s_{csp}^c), ORE)\ \text{synch}\ s_c \wedge$

$\quad \aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge \aleph_c = \aleph_z^c \cup \aleph_{csp}^c \wedge$

$\quad tst_a \in timed\,states_Z(\!|\{h_a\}|\!)(\!|\{(s_z^a, \aleph_z^a)\}|\!)$

$\vdash$ [parallel composition is total]

$\ldots$

$\exists h_c : HSC;\ s_z^c, s_{csp}^c : TimedTrace;\ \aleph_z^c, \aleph_{csp}^c : TimedRefusal \bullet$

$\exists s_{csp}^a : TimedTrace;\ \aleph_{csp}^a : TimedRefusal \mid ((s_{csp}^a, \aleph_{csp}^a), (s_{csp}^c, \aleph_{csp}^c)) \in Retr_{tf}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists h_a : HSA;\ s_z^a : TimedTrace;\ \aleph_z^a : TimedRefusal \mid (h_a, h_c) \in Retr_{hist}(\!|\{\mathcal{M_R}\}|\!) \wedge$

$\quad ((s_z^a, \aleph_z^a), (s_z^c, \aleph_z^c)) \in Retr_{tf}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists tst_a : TimedState \mid (tst_a, tst_c) \in Retr_{tst}(\!|\{\mathcal{M_R}\}|\!) \bullet$

$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$

$\quad ((s_z^a, s_{csp}^a), ORE)\ \text{synch}\ s_a \wedge \aleph_z^a \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^a \setminus (\mathbb{T} \times ORE^{\checkmark}) \bullet$

$(s_z^a, \aleph_z^a) \in timedfailures_Z (\!| \{(h_a, HSA)\} |\!) \wedge$
$((s_{csp}^a, \aleph_{csp}^a) \notin timedfailures_{ETFM} [\![CSU_A.EA]\!]_{(\mathcal{M}_\mathcal{A}, CSU_A.I \cup CSU_A.L)}$
$\vee (s_{csp}^a, \aleph_{csp}^a) \in$
$timedfailures_{ETCSP} [\![CSU_A.Behav\ Behaviour]\!]_{(CSU_A.Behav, (\mathcal{M}_\mathcal{A}, \mathcal{M}_\mathcal{A}), CSU_A.I \cup CSU_A.L)}) \wedge$
$((s_z^c, s_{csp}^c), ORE)\ \text{synch}\ s_c \wedge$
$\aleph_z^c \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^c \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge \aleph_c = \aleph_z^c \cup \aleph_{csp}^c \wedge$
$tst_a \in timed\,states_Z (\!| \{h_a\} |\!) (\!| \{(s_z^a, \aleph_z^a)\} |\!)$

$\vdash$ [Lemma C.11]

...

$\exists s_{csp}^a : TimedTrace;\ \aleph_{csp}^a : TimedRefusal \bullet$
$\exists h_a : HSA;\ s_z^a : TimedTrace;\ \aleph_z^a : TimedRefusal \bullet$
$\exists tst_a : TimedState \mid (tst_a, tst_c) \in Retr_{tst} (\!| \{\mathcal{M}_\mathcal{R}\} |\!) \bullet$
$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \bullet$
$((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf} (\!| \{\mathcal{M}_\mathcal{R}\} |\!) \wedge$
$((s_z^a, s_{csp}^a), ORE)\ \text{synch}\ s_a \wedge \aleph_z^a \setminus (\mathbb{T} \times ORE^{\checkmark}) = \aleph_{csp}^a \setminus (\mathbb{T} \times ORE^{\checkmark}) \wedge$
$(s_z^a, \aleph_z^a) \in timedfailures_Z (\!| \{(h_a, HSA)\} |\!) \wedge$
$((s_{csp}^a, \aleph_{csp}^a) \notin timedfailures_{ETFM} [\![CSU_A.EA]\!]_{(\mathcal{M}_\mathcal{A}, CSU_A.I \cup CSU_A.L)}$
$\vee (s_{csp}^a, \aleph_{csp}^a) \in$
$timedfailures_{ETCSP} [\![CSU_A.Behav\ Behaviour]\!]_{(CSU_A.Behav, (\mathcal{M}_\mathcal{A}, \mathcal{M}_\mathcal{A}), CSU_A.I \cup CSU_A.L)}) \wedge$
$tst_a \in timed\,states_Z (\!| \{h_a\} |\!) (\!| \{(s_z^a, \aleph_z^a)\} |\!)$

$\vdash$ [def. $tfs_of_model_{INT}$]

$(tst_a, tst_c) \in Retr_{tst} (\!| \{\mathcal{M}_\mathcal{R}\} |\!) \wedge$
$((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf} (\!| \{\mathcal{M}_\mathcal{R}\} |\!) \wedge$
$((s_a, \aleph_a), tst_a) \in tfs_of_model_{INT} (\!| \{(CSU_A, \mathcal{M}_\mathcal{A})\} |\!)$

$\vdash$ [def. $Retr_{Interface/State}$]

$((s_c, \aleph_c), tst_c) \in Retr_{Interface/State} (\!| \{RSU\} |\!) (\!| tfs_of_model_{INT} (\!| \{(CSU_A, \mathcal{M}_\mathcal{A})\} |\!) |\!)$

□

Lemma C.12
For the transition between $tfs_of_model_{INT}$ and $tfs_of_model_{EXT}$, we need a lemma similar to Lemma C.11: the relation $Retr_{tf}$ must distribute over the hiding operator.

$$\frac{\begin{array}{l} ((s_a^*, \aleph_a^*), (s_c^*, \aleph_c^*)) \in Retr_{tf} (\!| \{\mathcal{M}_\mathcal{R}\} |\!) \\ s_c \restriction_{ttr} ORE = s_c^* \\ \aleph_c^* = \aleph_c \cup (\mathbb{T} \times ORE) \\ s_a \restriction_{ttr} ORE = s_a^* \\ \aleph_a^* = \aleph_a \cup (\mathbb{T} \times ORE) \end{array}}{((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf} (\!| \{\mathcal{M}_\mathcal{R}\} |\!)}$$

Proof of Theorem 10.8.

$((s_c, \aleph_c), tst_c) \in tfs_of_model_{EXT}(\{(CSU_C, \mathcal{M}_C)\})$

$\vdash$ [def. $tfs_of_model_{EXT}$]

$\exists s_c^* : TimedTrace;\ \aleph_c^* : TimedRefusal \bullet$
$\quad ((s_c^*, \aleph_c^*), tst_c) \in tfs_of_model_{INT}(\{(CSU_C, \mathcal{M}_C)\}) \wedge$
$\quad s_c \upharpoonright_{ttr} ORE = s_c^* \wedge \aleph_c^* = \aleph_c \cup (\mathbb{T} \times ORE)$

$\vdash$ [Theorem 10.7]

$\exists s_c^* : TimedTrace;\ \aleph_c^* : TimedRefusal \bullet$
$\exists s_a^* : TimedTrace;\ \aleph_a^* : TimedRefusal;\ tst_a : TimedState \mid$
$\quad ((s_a^*, \aleph_a^*), (s_c^*, \aleph_c^*)) \in Retr_{tf}(\{\mathcal{M}_\mathcal{R}\}) \wedge (tst_a, tst_c) \in Retr_{tst}(\{\mathcal{M}_\mathcal{R}\}) \bullet$
$\quad ((s_a^*, \aleph_a^*), tst_a) \in tfs_of_model_{INT}(\{(CSU_A, \mathcal{M}_\mathcal{A})\}) \wedge$
$\quad s_c \upharpoonright_{ttr} ORE = s_c^* \wedge \aleph_c^* = \aleph_c \cup (\mathbb{T} \times ORE)$

$\vdash$ [hiding operator is total]

$\exists s_c^* : TimedTrace;\ \aleph_c^* : TimedRefusal \bullet$
$\exists s_a^* : TimedTrace;\ \aleph_a^* : TimedRefusal;\ tst_a : TimedState \mid$
$\quad ((s_a^*, \aleph_a^*), (s_c^*, \aleph_c^*)) \in Retr_{tf}(\{\mathcal{M}_\mathcal{R}\}) \wedge (tst_a, tst_c) \in Retr_{tst}(\{\mathcal{M}_\mathcal{R}\}) \bullet$
$\exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$
$\quad s_a \upharpoonright_{ttr} ORE = s_a^* \wedge \aleph_a^* = \aleph_a \cup (\mathbb{T} \times ORE) \bullet$
$\quad ((s_a^*, \aleph_a^*), tst_a) \in tfs_of_model_{INT}(\{(CSU_A, \mathcal{M}_\mathcal{A})\}) \wedge$
$\quad s_c \upharpoonright_{ttr} ORE = s_c^* \wedge \aleph_c^* = \aleph_c \cup (\mathbb{T} \times ORE)$

$\vdash$ [Lemma C.12]

$\exists s_a^* : TimedTrace;\ \aleph_a^* : TimedRefusal;\ tst_a : TimedState \mid$
$\quad (tst_a, tst_c) \in Retr_{tst}(\{\mathcal{M}_\mathcal{R}\}) \bullet$
$\quad \exists s_a : TimedTrace;\ \aleph_a : TimedRefusal \mid$
$\quad\quad ((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\{\mathcal{M}_\mathcal{R}\}) \wedge$
$\quad\quad s_a \upharpoonright_{ttr} ORE = s_a^* \wedge \aleph_a^* = \aleph_a \cup (\mathbb{T} \times ORE) \bullet$
$\quad ((s_a^*, \aleph_a^*), tst_a) \in tfs_of_model_{INT}(\{(CSU_A, \mathcal{M}_\mathcal{A})\})$

$\vdash$ [def. $tfs_of_model_{EXT}$]

$(tst_a, tst_c) \in Retr_{tst}(\{\mathcal{M}_\mathcal{R}\}) \wedge$
$((s_a, \aleph_a), (s_c, \aleph_c)) \in Retr_{tf}(\{\mathcal{M}_\mathcal{R}\}) \wedge$
$((s_a, \aleph_a), tst_a) \in tfs_of_model_{EXT}(\{(CSU_A, \mathcal{M}_\mathcal{A})\})$

$\vdash$ [def. $Retr_{Interface/State}$]

$((s_c, \aleph_c), tst_c) \in Retr_{Interface/State}(\{RSU\})(tfs_of_model_{EXT}(\{(CSU_A, \mathcal{M}_\mathcal{A})\}))$

$\square$

Appendix D

RT-Z Syntax

The concrete syntax of RT-Z is formally defined in this appendix. Again, we follow the approach taken in the Z Standard by using the same subset of the standard ISO [1996] as the syntactic metalanguage. The operators of this syntactic metalanguage that we use in the following syntax definition are described in Table D.1, which we adopt from [ISO, 2002, p. 3].

As elaborated in Part II, the syntax of RT-Z is built on the syntax of the Z notation. For instance, the `TYPES & CONSTANTS` section of a specification unit is constituted by a Z section. Therefore, the following syntax definition refers to non-terminal symbols of the concrete syntax of the Z notation, as defined in [ISO, 2002, Section 8.2]. For convenience, we map the identifiers of these non-terminals to corresponding identifiers that clearly indicate their Z origin.[1]

For the predicate languages of both the extended timed failures model (ETFM) and the timed failures/states model (TFSM), we define the syntax of the core language (to understand the term "core language" see the remarks on p. 134).

In the following syntax definition, non-terminals with the suffix $_{-tok}$ denote corresponding terminal symbols, e.g.,

[1] `Section` and `Expression` are mapped to `ZSect` and `ZExpr`, respectively.

symbol	definition
=	defines a non-terminal on the left in terms of the syntax on the right
|	separates alternatives
,	separates notation to be concatenated
{}	delimit notation to be repeated zero or more times
[]	delimit optional notation
()	are grouping brackets
' '	delimit a terminal symbol
;	delimits a definition

Table D.1: Syntactic metalanguage.

$\{_{\texttt{-tok}} = \text{`}\{\text{`}$

Moreover, the non-terminal NL denotes a "newline."

RT-Z Specifications

```
RTZSpec = [ZSect, NL], (CSpecUnit | ASpecUnit), {NL, (CSpecUnit | ASpecUnit)};

Id = LETTER, {LETTER | DIGIT | `_`};
LETTER = `a` | ... | `z` | `A` | ... | `Z`;
DIGIT = `0` | ... | `9`;
```

Specification Units

```
CSpecUnit = `SPEC.UNIT`, Id, [[-tok , Id, {,-tok , Id}, ]-tok],
            [`EXTENDS`, Id, {,-tok , Id}],
            NL, CSpecSect, {NL, CSpecSect};

CSpecSect = ExtendsSect
          | SubunitsSect
          | TypesconstSect
          | InterfaceSect
          | EnvassumpSect
          | LocalSect
          | StateSect
          | OppredsSect
          | BehaviourSect
          ;

ASpecUnit = `SPEC.UNIT`, Id, [[-tok , Id, {,-tok , Id}, ]-tok],
            [`EXTENDS`, Id, {,-tok , Id}],
            NL, ASpecSect, {NL, ASpecSect};

ASpecSect = ExtendsSect
          | TypesconstSect
          | InterfaceSect
          | EnvassumpSect
          | IORelSect
          | StatepropSect
          | BehavpropSect
          ;

ExtendsSect = `EXTENDS`, NL, ExtendsDecl, {;-tok, NL, ExtendsDecl};

ExtendsDecl = Id, {NL, ExtendsItem, ;-tok};
```

```
ExtendsItem = ‘instantiate ’, Id, ‘ with ’, Id, {,-tok , Id, ‘ with ’, Id}
            | ‘rename ’, Id, ‘ to ’, Id, {,-tok , Id, ‘ to ’, Id}
            | ‘hide ’, Id, {,-tok , Id}
            | ‘redefine ’, Id, {,-tok , Id}
            ;

SubunitsSect = ‘SUBUNITS‘, NL, SubunitsDecl, {;-tok, NL, SubunitsDecl};

SubunitsDecl = ‘subunit‘, Id, ‘spec. unit‘, Id, {NL, SubunitsItem, ;-tok};

SubunitsItem = ‘rename ’, Id, ‘ to ’, Id, {,-tok , Id, ‘ to ’, Id}
             | ‘hide ’, Id, {,-tok , Id}
             | ‘instantiate ’, Id, ‘ with ’, Id, {,-tok , Id, ‘ with ’, Id}
             ;

TypesconstSect = ‘TYPES & CONSTANTS‘, NL, ZSect;

InterfaceSect = ‘INTERFACE‘, NL,
                ‘port‘, Id, ‘domain‘, ZExpr,
                {;-tok, NL, ‘port‘, Id, ‘domain‘, ZExpr};

EnvassumpSect = ‘ENVIRONMENTAL ASSUMPTIONS‘, NL,
                (PredDef_ETFM | EvSetDef),
                {NL, (PredDef_ETFM | EvSetDef)};

LocalSect = ‘LOCAL‘, NL,
            ‘channel‘, Id, ‘domain‘, ZExpr,
            {;-tok, NL, ‘channel‘, Id, ‘domain‘, ZExpr}
            ;

StateSect = ‘STATE‘, NL, ZSect;

OppredsSect = ‘OPS & PREDS‘, [[-tok , Id, {,-tok , Id}, ]-tok], NL, ZSect;

BehaviourSect = ‘BEHAVIOUR‘, NL, (ProcDef | EvSetDef),
                {NL, (ProcDef | EvSetDef)};

IORelSect = ‘I/O RELATIONS‘, NL, ZSect;

StatepropSect = ‘STATE PROPERTIES‘, NL, ZSect;

BehavpropSect = ‘BEHAVIOURAL PROPERTIES‘, NL, (PredDef_TFSM | EvSetDef),
                {NL, (PredDef_TFSM | EvSetDef)};
```

Process Terms

$$
\begin{array}{rl}
\mathtt{Proc} = & \mathtt{Id}, [\,(_{\mathtt{-tok}}, \mathtt{Id}, \{,_{\mathtt{-tok}}, \mathtt{Id}\},)_{\mathtt{-tok}}\,] \\
| & \text{`}\mathit{Stop}\text{`} \mid \text{`}\mathit{Skip}\text{`} \\
| & \mathtt{Proc}, ;_{\mathtt{-tok}}, \mathtt{Proc} \\
| & \mathtt{Proc}, \text{`}[\![\text{`}, \mathtt{EvSetExpr}, \text{`}]\!]\text{`}, \mathtt{Proc} \\
| & \mathtt{Proc}, \text{`}[\![\text{`}, \mathtt{EvSetExpr}, |_{\mathtt{-tok}}, \mathtt{EvSetExpr}, \text{`}]\!]\text{`}, \mathtt{Proc} \\
| & \mathtt{Proc}, \text{`}\ |||\ \text{`}, \mathtt{Proc} \\
| & \mathtt{Proc}, \text{`}\ \Box\ \text{`}, \mathtt{Proc} \mid \mathtt{Proc}, \text{`}\ \sqcap\ \text{`}, \mathtt{Proc} \\
| & \mathtt{Id}, (_{\mathtt{-tok}}, \mathtt{Proc},)_{\mathtt{-tok}} \mid \mathtt{Id}, \text{`}^{-1}\text{`}, (_{\mathtt{-tok}}, \mathtt{Proc},)_{\mathtt{-tok}} \\
| & \mathtt{Proc}, \text{`}\ \setminus\ \text{`}, \mathtt{EvSetExpr} \mid \mathtt{Proc}, \text{`}\ \triangle\ \text{`}, \mathtt{Proc} \\
| & \text{`}\ \bigsqcap\ \text{`}, \mathtt{Id}, \text{`}\ \in\ \text{`}, \mathtt{ZExpr}, \mathtt{Proc} \\
| & \text{`}\ \big|\big|\big|\ \text{`}, \mathtt{Id}, \text{`}\ \in\ \text{`}, \mathtt{ZExpr}, \mathtt{Proc} \\
| & \text{`}\ \big\|\ \text{`}, \mathtt{Id}, \text{`}\ \in\ \text{`}, \mathtt{ZExpr}, [_{\mathtt{-tok}}, \mathtt{EvSetExpr},]_{\mathtt{-tok}}, \mathtt{Proc} \\
| & \mathtt{Id}, \text{`}?\text{`}, (_{\mathtt{-tok}}, \mathtt{Id}, \text{`}\ :\ \text{`}, \mathtt{ZExpr},)_{\mathtt{-tok}}\ \text{`}\ \rightarrow\ \text{`}, \mathtt{Proc} \\
| & \mathtt{Id}, \text{`}?\text{`}, (_{\mathtt{-tok}}, \mathtt{Id}, \text{`}\ :\ \text{`}, \mathtt{ZExpr},)_{\mathtt{-tok}}, \text{`}@\text{`}, \mathtt{Id}, \text{`}\ \rightarrow\ \text{`}, \mathtt{Proc} \\
| & \mathtt{Id}, \text{`}!\text{`}, \mathtt{ZExpr}, \text{`}\ \rightarrow\ \text{`}, \mathtt{Proc} \\
| & \text{`}\ \mathit{Wait}\ \text{`}, \mathtt{TExpr} \mid \mathtt{Proc}, \text{`}\triangleright\text{`}, \{_{\mathtt{-tok}}, \mathtt{TExpr}, \}_{\mathtt{-tok}}, \mathtt{Proc} \\
| & \mathtt{Proc}, \text{`}\triangle\text{`}, \mathtt{TExpr}, \mathtt{Proc} \\
 & ;
\end{array}
$$

$$
\mathtt{ProcDef} = \mathtt{Id}, [\,(_{\mathtt{-tok}}, \mathtt{Id}, \{,_{\mathtt{-tok}}, \mathtt{Id}\},)_{\mathtt{-tok}}\,], \text{`}\ \widehat{=}\ \text{`}, \mathtt{Proc};
$$

Event Sets

$$
\mathtt{EvSetDef} = \mathtt{Id}, \text{`}\ \widehat{=}\ \text{`}, \mathtt{EvSetExpr};
$$

$$
\mathtt{EvSetExpr} = \mathtt{RefExpr} \mid \mathtt{Id};
$$

Predicates (ETFM)

$$
\begin{array}{rl}
\mathtt{Pred_ETFM} = & \mathtt{AtomicPred_ETFM} \\
| & \mathtt{Pred_ETFM}, \text{`}\ \wedge\ \text{`}, \mathtt{Pred_ETFM} \\
| & \mathtt{Pred_ETFM}, \text{`}\ \vee\ \text{`}, \mathtt{Pred_ETFM} \\
| & \text{`}\neg\ \text{`}, \mathtt{Pred_ETFM} \\
| & \mathtt{Pred_ETFM}, \text{`}\ \Rightarrow\ \text{`}, \mathtt{Pred_ETFM} \\
| & \mathtt{Pred_ETFM}, \text{`}\ \Leftrightarrow\ \text{`}, \mathtt{Pred_ETFM} \\
| & \text{`}\ \forall\ \text{`}, \mathtt{Id}, \text{`}\ :\ \text{`}, \mathtt{ZExpr}, \text{`}\ \bullet\ \text{`}, \mathtt{Pred_ETFM} \\
| & \text{`}\ \exists\ \text{`}, \mathtt{Id}, \text{`}\ :\ \text{`}, \mathtt{ZExpr}, \text{`}\ \bullet\ \text{`}, \mathtt{Pred_ETFM} \\
| & \text{`}\ \mathbf{let}\ \text{`}, \mathtt{Id}, \text{`}\ ==\ \text{`}, \mathtt{ZExpr}, \text{`}\ \bullet\ \text{`}, \mathtt{Pred_ETFM} \\
 & ;
\end{array}
$$

$$
\mathtt{PredDef_ETFM} = \mathtt{Id}, [\,(_{\mathtt{-tok}}, \mathtt{Id}, \{,_{\mathtt{-tok}}, \mathtt{Id}\},)_{\mathtt{-tok}}\,], \text{`}\ \widehat{=}\ \text{`}, \mathtt{Pred_ETFM};
$$

```
AtomicPred_ETFM = Id, [ (-tok , Id, {,-tok , Id}, )-tok ]
                | Id, (-tok , Id, ‘ ≙ ‘, ZExpr, {,-tok , Id, ‘ ≙ ‘, ZExpr}, )-tok
                | TTrExpr, ‘ in ‘, TTrExpr
                | TTrExpr, ‘ prefix ‘, TTrExpr
                | TrExpr, ‘ in ‘, TrExpr
                | TrExpr, ‘ prefix ‘, TrExpr
                | TRefExpr, ‘ ⊆ ‘, TRefExpr
                | RefExpr, ‘ ⊆ ‘, RefExpr
                | TExpr, ‘ ≤ℝ ‘, TExpr
                | TExpr, ‘ ∈ ‘, TIntExpr
                | NatExpr, ‘ ≤ ‘, NatExpr
                ;

ValExpr = ZExpr
        | ‘seq‘, TrExpr
        | ‘set‘, TrExpr
        | ‘ head ‘, TrExpr | ‘ foot ‘, TrExpr
        | ‘ first ‘, TTrExpr | ‘ last ‘, TTrExpr
        | ‘time‘, TExpr
        ;

TExpr = ‘0ℝ‘ | ‘1ℝ‘ | Id
      | TExpr, ‘ +ℝ ‘, TExpr | TExpr, ‘ *ℝ ‘, TExpr
      | ‘ begin ‘, TTrExpr | ‘ begin ‘, TRefExpr
      | ‘ begin ‘, (-tok , TTrExpr, ,-tok , TRefExpr, )-tok
      | ‘ end ‘, TTrExpr | ‘ end ‘, TRefExpr
      | ‘ end ‘, (-tok , TTrExpr, ,-tok , TRefExpr, )-tok
      ;

TIntExpr = [-tok , TExpr, ,-tok , TExpr, ]-tok
         | (-tok , TExpr, ,-tok , TExpr, )-tok
         | [-tok , TExpr, ,-tok , TExpr, )-tok
         | (-tok , TExpr, ,-tok , TExpr, ]-tok
         ;

TTrExpr = ‘s‘
        | ‘⟨‘, ‘⟩‘
        | ‘⟨‘, (-tok , TExpr, ,-tok , Id, [‘.‘, ValExpr], )-tok ,
          {,-tok , (-tok , TExpr, ,-tok , Id, [‘.‘, ValExpr], )-tok }, ‘⟩‘
        | TTrExpr, ‘ ⁀ ‘, TTrExpr
        | ‘ tail ‘, TTrExpr | ‘ init ‘, TTrExpr
        | TTrExpr, ‘ ↾ttr ‘, RefExpr | TTrExpr, ‘ \ttr ‘, RefExpr
        | TTrExpr, ‘ ↑ttr ‘, TIntExpr
        | TTrExpr, ‘ ↾ttr ‘, TExpr | TTrExpr, ‘ ‖↾ttr ‘, TExpr
        | TTrExpr, ‘ ↿ttr ‘, TExpr | TTrExpr, ‘ ↿‖ttr ‘, TExpr
        | TTrExpr, ‘ +ttr ‘, TExpr | TTrExpr, ‘ ∸ttr ‘, TExpr
        ;
```

TRefExpr = '$\aleph$'
  | '$\varnothing$'
  | {-tok , (-tok , TExpr, ,-tok , Id, ['.', ValExpr],)-tok ,
    {,-tok , (-tok , TExpr, ,-tok , Id, ['.', ValExpr],)-tok }, }-tok
  | TRefExpr, ' $\cup$ ', TRefExpr
  | {-tok , ' head ', TTrExpr, }-tok | {-tok , ' foot ', TTrExpr, }-tok
  | TRefExpr, ' $\setminus_{tref}$ ', RefExpr
  | TRefExpr, ' $\upharpoonright_{tref}$ ', TIntExpr
  | TRefExpr, ' $\|\!\upharpoonright_{tref}$ ', TExpr | TRefExpr, ' $\upharpoonleft_{tref}$ ', TExpr
  | TRefExpr, ' $+_{tref}$ ', TExpr | TRefExpr, ' $\dot{-}_{tref}$ ', TExpr
  ;

TrExpr = '$\langle$', '$\rangle$' | '$\langle$', Id, '.', ValExpr, {,-tok , Id, '.', ValExpr}, '$\rangle$'
  | TrExpr, ' $\frown$ ', TrExpr | ' strip ', TTrExpr | ' tail ', TrExpr | ' init ', TrExpr
  | TrExpr, ' $\upharpoonright_{tr}$ ', RefExpr | TrExpr, ' $\setminus_{tr}$ ', RefExpr
  ;

RefExpr = '$\{\!|$ ', Id, ' $|\!\}$'
  | '$\varnothing$' | {-tok , Id, ['.', ValExpr], {,-tok , Id, ['.', ValExpr]} }-tok
  | RefExpr, ' $\cup$ ', RefExpr
  | {-tok , ' first ', TTrExpr, }-tok | {-tok , ' last ', TTrExpr }-tok
  | 'σ', TrExpr | 'σ', TTrExpr
  ;

NatExpr = Id
  | '0' | 'succ', NatExpr
  | TTrExpr, ' $\downarrow_{ttr}$ ', RefExpr | TrExpr, ' $\downarrow_{tr}$ ', RefExpr
  ;

Predicates (TFSM)

Pred_TFSM = AtomicPred_TFSM
  | Pred_TFSM, ' $\wedge$ ', Pred_TFSM
  | Pred_TFSM, ' $\vee$ ', Pred_TFSM
  | '$\neg$ ', Pred_TFSM
  | Pred_TFSM, ' $\Rightarrow$ ', Pred_TFSM
  | Pred_TFSM, ' $\Leftrightarrow$ ', Pred_TFSM
  | ' $\forall$ ', Id, ' : ', ZExpr, ' $\bullet$ ', Pred_TFSM
  | ' $\exists$ ', Id, ' : ', ZExpr, ' $\bullet$ ', Pred_TFSM
  | ' **let** ', Id, ' == ', ZExpr, ' $\bullet$ ', Pred_TFSM
  ;

```
AtomicPred_TFSM = AtomicPred_ETFM
                | Id, ' at ', TExpr
                | Id, ' before ', TExpr
                | Id, ' invariant'
                | Id, ' during ', TIntExpr
                ;
```

Appendix E

Glossary of Timed CSP Notation

We summarise the notation of timed CSP used throughout this thesis. As a convention, s denotes a timed trace, seq a sequence, $\aleph$ a timed refusal, e an event, t a time instant, A a set of events, I a time interval and c a channel.

Σ: universal set of events.

$\checkmark$: termination event ($\checkmark \notin \Sigma$).

$\Sigma^{\checkmark}$: $\Sigma \cup \{\checkmark\}$.

τ: internal event ($\tau \notin \Sigma$).

$\mathrm{channel}(c.v)$: channel component c of compound event $c.v$.

$\mathrm{value}(c.v)$: value component v of compound event $c.v$.

$\{\!| c |\!\}$: set of events that can be communicated along channel c.

$\langle\rangle$: empty sequence (timed, untimed trace).

$\langle e_1, \ldots, e_N \rangle$: sequence (timed, untimed trace) consisting of the elements (timed, untimed events) $e_1, \ldots, e_N$.

$seq_1 \frown seq_2$: concatenation of sequences (timed/untimed traces) seq_1 and seq_2.

$\mathrm{head}\, seq$: first element of sequence seq (timed/untimed trace).

$\mathrm{tail}\, seq$: sequence seq (timed/untimed trace) without the first element.

$\mathrm{foot}\, seq$: last element of sequence seq (timed/untimed trace).

$\mathrm{init}\, seq$: sequence seq (timed/untimed trace) without the last element.

$\#seq$: length of sequence seq (timed/untimed trace).

$\sigma\, seq$: set of elements appearing in sequence seq (timed/untimed trace).

seq_1 **in** seq_2: seq_1 is a contiguous subsequence of seq_2.

seq_1 **prefix** seq_2: seq_1 is a prefix of seq_2.

$\mathrm{strip}\ s$: untimed trace corresponding to timed trace s.

$\mathrm{first}\ s$: first event appearing in timed trace s.

$\mathrm{last}\ s$: last event appearing in the finite timed trace s.

$\mathrm{begin}_{ttr}\ s$, $\mathrm{begin}_{tref}\ \aleph$, $\mathrm{begin}_{tfail}(s, \aleph)$: time of the first event in timed trace s, timed refusal $\aleph$ or timed failure $(s, \aleph)$.

$\mathrm{end}_{ttr}\ s$, $\mathrm{end}_{tref}\ \aleph$, $\mathrm{end}_{tfail}(s, \aleph)$: time of the last event in finite timed trace s, least upper bound of times appearing in timed refusal $\aleph$ or timed failure $(s, \aleph)$.

$s \upharpoonright_{ttr} A$, $\aleph \upharpoonright_{tref} A$, $tr \upharpoonright_{tr} A$: timed trace s, timed refusal $\aleph$ or untimed trace tr restricted to the set of events A.

$\sigma(s)$, $\sigma_{tref}(\aleph)$: set of events appearing in timed trace s or timed refusal $\aleph$.

$s \downarrow_{ttr} A$, $tr \downarrow_{tr} A$: number of occurrences of events from A in timed trace s or untimed trace tr.

$s \setminus_{ttr} A$, $\aleph \setminus_{tref} A$, $(s, \aleph) \setminus_{tfail} A$, $tr \setminus_{tr} A$: timed trace s, timed refusal $\aleph$, timed failure $(s, \aleph)$ or untimed trace tr without the elements of A.

$s \uparrow_{ttr} I$, $\aleph \uparrow_{tref} I$: projection of timed trace s or timed refusal $\aleph$ to time interval I.

$s \upharpoonright_{ttr} t$, $s \Vert\!\upharpoonright_{ttr} t$, $\aleph \Vert\!\upharpoonright_{tref} t$: timed trace s before time t, timed trace s strictly before time t and timed refusal $\aleph$ before time t.

$s \upharpoonleft_{ttr} t$, $s \upharpoonleft\!\Vert_{ttr} t$, $\aleph \upharpoonleft_{tref} t$: timed trace s after time t, timed trace s strictly after time t and timed refusal after time t.

$s +_{ttr} t$, $\aleph +_{tref} t$, $(s, \aleph) +_{tfail} t$: timed trace s, timed refusal $\aleph$ or timed failure $(s, \aleph)$ translated through t time units.

$s \dot{-}_{ttr} t$, $\aleph \dot{-}_{tref} t$, $(s, \aleph) \dot{-}_{tfail} t$: timed trace s, timed refusal $\aleph$ or timed failure $(s, \aleph)$ translated back through t time units.

$(s_1, \aleph_1) \preceq (s_2, \aleph_2)$: information order on timed failures. $(s_1, \aleph_1)$ contains less trace and refusal information than $(s_2, \aleph_2)$.

s interleaves (s_1, s_2): s is an interleaving of the sequences s_1 and s_2.

$s\ \mathrm{synch}_A\ (s_1, s_2)$: s synchronises s_1 and s_2 on events in $A^{\checkmark}$.

Bibliography

M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

M. Ainsworth, A. H. Cruickshank, and P. J. L. Wallis. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, 1994.

R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

K. Araki, A. Galloway, and K. Taguchi, editors. *Proceedings of the 1st International Conference on Integrated Formal Methods*, 1999. Springer-Verlag.

J. C. M. Baeten and J. A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3: 142–188, 1991.

E. A. Boiten, H. Bowman, J. Derrick, P. F. Linington, and M. W. A. Steen. Viewpoint consistency in ODP. *Computer Networks*, 34(3):503–537, Sept. 2000.

E. A. Boiten, H. Bowman, J. Derrick, and M. Steen. Issues in multiparadigm viewpoint specification. In A. Finkelstein and G. Spanoudakis, editors, *SIGSOFT '96 International Workshop on Multiple Perspectives in Software Development (Viewpoints '96)*, pages 162–166. ACM, 1996.

T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

T. Bolognesi, F. Lucidi, and S. Trigila. Converging towards a timed LOTOS standard. *Computer Standards & Interfaces*, 16:87–118, 1994.

T. Bolognesi, J. van de Lagemaat, and C. Vissers. *LOTOSphere: Software Development with LOTOS*. Kluwer Academic Publishers, 1995.

H. Bowman, E. Boiten, J. Derrick, and M. Steen. Viewpoint consistency in ODP, a general interpretation. In E. Najm and J.-B. Stefani, editors, *First IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems*, pages 189–204. Chapman & Hall, Mar. 1996.

H. Bowman, E. A. Boiten, J. Derrick, and M. W. A. Steen. Strategies for consistency checking based on unification. *Science of Computer Programming*, 33:261–298, Apr. 1999a.

H. Bowman, M. Steen, E. Boiten, and J. Derrick. A formal framework for viewpoint consistency. Computing Laboratory Technical Report 22-99, University of Kent at Canterbury, 1999b.

M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems — an introduction to FOCUS. Technical Report TUM-I9202, Institut für Informatik, Technische Universität München, 1992.

J. Bryans, J. Davies, and S. Schneider. Towards a denotational semantics for ET-LOTOS. In I. Lee and S. A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, number 962 in Lecture Notes in Computer Science, pages 269–283. Springer-Verlag, 1995.

R. Büssow, R. Geisler, W. Grieskamp, and M. Klar. The μSZ notation version 1.0. Technical Report 97–26, Technische Universität Berlin, Fachbereich Informatik, 1997.

R. Büssow and W. Grieskamp. *The Z of ZETA*. Technische Universität Berlin, December 1998. The ZETA System Documentation.

M. Butler, L. Petre, and K. Sere, editors. *Proceedings of the 3rd International Conference on Integrated Formal Methods, IFM 2002*, number 2335 in Lecture Notes in Computer Science, 2002. Springer-Verlag.

Z. Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, Dec. 1991.

C. Choppy, P. Poizat, and J.-C. Royer. A global semantics for views. In *International Conference on Algebraic Methodology and Software Technology, AMAST'2000*, number 1816 in Lecture Notes in Computer Science, pages 165–180. Springer-Verlag, 2000.

E. Clarke and J. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University, Aug. 1996.

A. Coen-Porisini, R. A. Kemmerer, and D. Mandrioli. Specification of realtime systems using ASTRAL. *IEEE Transactions on Software Engineering*, 23(9):572–598, Sept. 1997.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

J. Davies. *Specification and Proof in Real-Time CSP*. Technical monograph, Oxford University, 1993. Cambridge University Press.

J. Davies and S. Schneider. Real-time CSP. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-Time System Development*. World Scientific Publishing Company, Inc., Feb. 1995.

J. W. de Bakker, C. Huizing, W. P. de Roever, and G. Rozenberg, editors. *Real-Time: Theory in Practice*, number 600 in Lecture Notes in Computer Science, 1991. Springer-Verlag.

R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer-Verlag, 2001.

J. Derrick, E. Boiten, H. Bowman, and M. Steen. Viewpoints and consistency: translating LOTOS to Object-Z. *Computer Standards and Interfaces*, 21:251–272, Aug. 1999.

J. Derrick and G. Smith. Structural refinement in Object-Z / CSP. In Grieskamp et al. [2000].

J. S. Dong and B. Mahony. Active object in TCOZ. In J. Staples, M. Hinchey, and S. Liu, editors, *Proceedings of the 1998 IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 16–25. IEEE Computer Society Press, 1998.

H. Ehrig and M. Große-Rhode, editors. *Proceedings of INT 2002 : Second International Workshop on Integration of Specification Techniques for Applications in Engineering*, 2002. URL `http://tfs.cs.tu-berlin.de/~mgr/int02/proceedings.html`.

H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications*. Springer-Verlag, 1985.

H. B. Enderton. *Elements of set theory*. Academic Press, 1977.

ESPRESS. Engineering of safety-critical embedded systems (research project), 1998. URL `http://www.first.gmd.de/~espress/`.

C. Fidge. A comparative introduction to CSP, CCS and LOTOS. Technical Report 93-24, Software Verification Research Centre, Department of Computer Science, University of Queensland, 1994.

C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A set-theoretic model for real-time specification and reasoning. In J. Jeuring, editor, *Mathematics of Program Construction (MPC'98)*, number 1422 in Lecture Notes in Computer Science, pages 188–206. Springer-Verlag, 1998.

A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.

C. Fischer. How to combine Z with a process algebra. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM '98: The Z Formal Specification Notation*, number 1493 in Lecture Notes in Computer Science, pages 5–23. Springer-Verlag, 1998.

C. Fischer. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik, Universität Oldenburg, 2000.

C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In Araki et al. [1999], pages 315–334.

Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR2 User Manual*, 2000. URL `http://www.fsel.com/fdr2_manual.html`.

W. Grieskamp, M. Heisel, and H. Dörr. Specifying embedded systems with statecharts and Z: an agenda for cyclic software components. *Science of Computer Programming*, 40(1):31–57, 2001.

W. Grieskamp, T. Santen, and B. Stoddart, editors. *Proceedings of the 2nd International Conference on Integrated Formal Methods, IFM 2000*, number 1945 in Lecture Notes in Computer Science, 2000. Springer-Verlag.

M. Große-Rhode. Semantic integration of heterogeneous formal specifications via transformation systems. Technischer Bericht 2001/13, Technische Universität Berlin, Fakultät für Elektrotechnik und Informatik, 2001.

R. K. Gupta. Introduction to embedded systems. Course Material, 2002. URL `http://www1.ics.uci.edu/~rgupta/ics212.html`. University of California at Irvine.

A. G. Hamilton. *Numbers, sets and axioms: the apparatus of mathematics*. Cambridge University Press, 1982.

D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

A. Haxthausen and C. George. A concurrency case study using RAISE. In Woodcock and Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, number 670 in Lecture Notes in Computer Science, pages 367–387. Springer-Verlag, 1993.

M. Heisel. *Improving Software Quality with Formal Methods*. Habilitation thesis, Technische Universität Berlin, 1997.

M. Heisel and C. Sühl. Combining Z and Real-Time CSP for the development of safety-critical systems. In *Proceedings 15th International Conference on Computer Safety, Reliability and Security*. Springer-Verlag, 1996.

M. Heisel and C. Sühl. Methodological support for formally specifying safety-critical software. In *Proceedings 16th International Conference on Computer Safety, Reliability and Security*. Springer-Verlag, 1997.

C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*. John Wiley & Sons, 1996.

S. Helke and F. Kammüller. Representing hierarchical automata in interactive theorem provers. In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2001)*, number 2152 in Lecture Notes in Computer Science, pages 233–248. Springer-Verlag, 2001.

M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.

C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

J. Hoenicke and E.-R. Olderog. Combining specification techniques for processes, data and time. In Butler et al. [2002], pages 245–266.

ISO. Information Technology – Syntactic Metalanguage – Extended BNF. International Standard ISO/IEC 14977:1996, International Organization for Standardization, 1996.

ISO. Information processing systems – Open systems interconnection – Estelle – A formal description technique based on an extended state transition model. International Standard 9074, International Organization for Standardization, 1997.

ISO. Information technology – Open Distributed Processing – Reference model: Overview. International Standard 10746-1, International Organization for Standardization, 1998.

ISO. Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics. Draft International Standard ISO/IEC 13568:2002, International Organization for Standardization, 2002.

F. Jahanian and A. K. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, 1994.

C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

M. B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.

S. Kalvala. A formulation of TLA in Isabelle. In *Proc. 8th International Higher Order Theorem Proving*, pages 214–228, 1995.

Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics — 9th International Conference*, number 1125 in Lecture Notes in Computer Science, pages 283–298. Springer-Verlag, 1996.

P. Koopman. Embedded systems in the real world. Course Material: Dependable Embedded Systems, 1999. URL `http://www-2.cs.cmu.edu/~koopman/des_s99/emb_sys_intro.pdf`. Carnegie Mellon University.

P. J. Koopman. Embedded system design issues (the rest of the story). In *IEEE International Conference on Computer Design (ICCD '96)*, pages 310–319. IEEE Computer Society, 1996.

K. Kronlöf, editor. *Method Integration—Concepts and Case Studies*. Wiley Series in Software Based Systems. John Wiley & Sons, 1993.

L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

L. Lamport. Specifying concurrent systems with TLA$^+$. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, volume 173 of *NATO: Computer & Systems Sciences*. IOS Press, 2000.

L. Léonard and G. Leduc. An introduction to ET-LOTOS for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29:271–292, 1997.

N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: The semantics of TCOZ. Technical Report 97-24, Commonwealth Scientific and Industrial Research Organisation (CSIRO), 1997.

B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering*, pages 95–104. IEEE Computer Society Press, 1998a.

B. Mahony and J. S. Dong. Network topology and a case study in TCOZ. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM '98: The Z Formal Specification Notation*, number 1493 in Lecture Notes in Computer Science, pages 308–327. Springer-Verlag, 1998b.

B. Mahony and J. S. Dong. Overview of the semantics of TCOZ. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*. Springer-Verlag, 1999a.

B. Mahony and J. S. Dong. Sensors and actuators in TCOZ. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99—Formal Methods*, number 1709 in Lecture Notes in Computer Science, pages 1166–1185. Springer-Verlag, 1999b.

B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–176, 2000.

Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification*. Springer-Verlag, 1992.

A. J. R. G. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

F. Moller and C. Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, number 458 in Lecture Notes in Computer Science, pages 401–415. Springer-Verlag, 1990.

X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [1991], pages 526–548.

J. Ostroff. Verification of safety critical systems using TTM/RTTL. In de Bakker et al. [1991], pages 573–602.

D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

L. C. Paulson. *Isabelle's Object-Logics*. Computer Laboratory, University of Cambridge, 1996.

B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall International Series in Computer Science, 2nd edition, 1996.

A. P. Ravn, H. Rischel, and K. M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, Jan. 1993.

A. W. Roscoe. An alternative order for the failures model. *Journal of Logic and Computation*, 2 (5):557–577, Oct. 1992.

A. W. Roscoe. Unbounded non-determinism in CSP. *Journal of Logic and Computation*, 3(2): 131–172, Apr. 1993.

A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

A. W. Roscoe and G. Barrett. Unbounded nondeterminism in CSP. In *Mathematical Foundations of Programming Semantics*, number 442 in Lecture Notes in Computer Science, pages 160–193. Springer-Verlag, 1989.

M. Saaltink. The Z/EVES system. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *ZUM '97: Z Formal Specification Notation. 11th International Conference of Z Users*, number 1212 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, 1997.

B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1998.

S. Schneider. *Correctness and Communication in Real-time Systems*. PhD thesis, Oxford University Computing Laboratory, Programming Research Group, 1990.

S. Schneider. Abstraction and testing. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99 – Formal Methods*, number 1708 in Lecture Notes in Computer Science, pages 738–757. Springer-Verlag, 1999a.

S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, 1999b.

A. Sherif, A. Sampaio, and S. Cavalcante. An integrated approach to specification and validation of real-time systems. In J. N. Oliveira and P. Zave, editors, *Proceedings of FME 2001: Formal Methods for Increasing Software Productivity*, number 2021 in Lecture Notes in Computer Science, pages 278–299. Springer-Verlag, 2001.

G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *Proceedings of FME'97: Industrial Benefit of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 62–81. Springer-Verlag, 1997.

G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods Series. Kluwer Academic Publishers, 2000.

G. Smith. An integration of Real-Time Object-Z and CSP for specifying concurrent real-time systems. In Butler et al. [2002], pages 267–285.

G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems—an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

G. Smith and I. J. Hayes. Towards Real-Time Object-Z. In Araki et al. [1999], pages 49–65.

SoftSpez. Software specification: Integration of software specification techniques for applications in engineering. DFG Priority Programme, 2000. URL `http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html`.

I. Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1995.

J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

C. Sühl. RT-Z: An integration of Z and timed CSP. In Araki et al. [1999], pages 29–48.

C. Sühl. Applying RT-Z to develop safety-critical systems. In T. Maibaum, editor, *Proceedings of the Third International Conference on Fundamental Approaches to Software Engineering (FASE 2000)*, number 1783 in Lecture Notes in Computer Science, pages 51–65. Springer-Verlag, 2000.

C. Sühl. An overview of the integrated formalism RT-Z. *Formal Aspects of Computing*, 13(2): 94–110, 2002.

W. A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 1975.

H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 318–337. Springer-Verlag, Sept. 1997.

H. E. Treharne. *Combining Control Executives and Software Specifications*. PhD thesis, Department of Computer Science, Royal Holloway, University of London, 2000.

UK Ministry of Defence. Defence standard 00-55: Requirements for safety related software in defence equipment, Aug. 1997.

UNIFORM. Universal formal methods workbench (research project), 1998. URL `http://www.informatik.uni-bremen.de/uniform/`.

M. Weber. Systematic design of embedded control systems. GMD-Bericht 283, GMD—Forschungszentrum Informationstechnik GmbH, 1997.

J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

P. Zave and M. Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, Oct. 1993.

P. Zave and M. Jackson. Where do operations come from: A multiparadigm specification technique. *IEEE Transactions on Software Engineering*, 22(7):508–528, 1996.

Index of Terms

A

abstract data types . . . 160
abstract language constructs . . . 15, 229
abstraction levels . . . 4, 15, 223
ACP_ρ . . . 263
ACT ONE . . . 23, 182, 263
admissible . . . 259, 261
aggregation . . . **66–82**, 200
  indexed . . . **78–81**, 194, 198
  simple . . . **72**
aggregation hierarchy . . . 68, 71, 75, 105
algebraic laws . . . **24**
alternating bit protocol . . . 54, 95
architectural description language . . . 262
architecture . . . 10, 46, 183
ASTRAL . . . 54, 68

B

B . . . 54
base formalisms . . . 31, 39, 171, 176
BaseStates . . . 88
behaviour
  continuous . . . 16, 209
  discrete . . . 16
  hybrid . . . 13, 16
$Behaviour_{Ext}$ . . . 89
blocking view . . . 49, 159

C

calculus . . . 227–230, 260
CCS . . . 23, 263
channels . . . 12, **236**
classification . . . 4, 31, 223
closed system view . . . 12, 105, 151, 156
coherence . . . 33
cohesion . . . 70
complexity . . . 4, 53, 65, 225
component-oriented approaches . . . 31
compositional . . . 236, 247, 260
compositionality . . . 220, 226
computational model . . . 238
concrete language constructs . . . 15, 229
concurrent . . . 7, 53, 69–71, 78, 106–108, 205, 225
consistency . . . 27, 31
  checks . . . 27, 230
contraction mappings . . . 148, 257
control behaviour . . . 37, 46
correspondences . . . 27, 33, 34
CSP . . . 17, 21, 183, 236
CSP part . . . 39, 44, 46, 105
CSP-OZ . . . **17–19**, 33, 104, 171–176

D

data-processing tasks . . . 15
data-related characteristics . . . 37, 39, 45
decomposition style . . . 29, 32
delayed reference expansion . . . 77
development process . . . 4, 9
distributed . . . 7, 205
distribution . . . 78
double event approach . . . 48
duality . . . 249
Duration Calculus . . . 264
dynamic behaviour . . . 37, 39, 60, 62

E

embedded systems . . . 3, **7**, 37
environment . . . 11, 236
environmental assumptions 13, 206, 224
ET-LOTOS . . . 263
event transitions . . . 238
events . . . 11, 12, 236
  operation-related . . . **124**
  plain . . . **124**
evolution transitions . . . 238

execution . . . **245**
maximal . . . 245
extended timed failures model . . 109, 122
extension . . . **83–94**

F

failures . . . **124**, 127
failures–divergences model . 22, 104, 158
fairness . . . 26
finite variability . . . 125, **247**
fixed points . . . 148, 256
least . . . 148
formal description technique . . . 23
formal methods . . . 3, 24, 226, 229, 230
formal semantics . . . 15
formal specification . . . 224

G

gas burner . . . 207
global definitions . . . 53, 184
global invariants . . . **68–71**, 75, 106, 225
GlobalInv . . . 68

H

hiding . . . **94**
hierarchical decomposition . . . 53
history . . . **110**, 117
history model . . . 109, **117–122**

I

I/O relations . . . 40
immediate reference expansion . . . 76
incremental . . . 66, 83
information hiding . . . 14
$Init_{Ext}$ . . . 88
integrated formalism . . . 3
integration
conserving . 4, 21, 23, **33**, 37, 39, 103, 223
monolithic . . . 17, 18, 21, **33**
interface . . . 12, 236
intermediate semantic models . . . **109**
IO refinement . . . 152, 160

L

library . . . 65, 84
lifting . . . 75
links . . . 39, 46, 99, 111
LOTOS . . . **23–24**, 34, 54, 180–182

M

maximal parallelism . . . 237, 240
maximal progress . . . 237, 242, 245
methodological support . . . 25, 229
metric space . . . 256, **257**
complete . . . 148, 256
Modechart/RTL . . . 264
model checker . . . 229
models (Z) . . . **110**, 234
multi-lift system . . . 183
multi-viewpoint specification . . . 31
mutual recursion . . . 244

N

non-interference conditions . . . **106**
nondeterminism . . . 104
unbounded . . . 148, 258

O

Object-Z . . . 17, 18, 19, 20, 21, 54, 66, 71, 183, 225
open system view . . . 12, 105, 151, 157
operation
execution . . . 48
invocation . . . 47, 116
termination . . . 47, 116
operation-related events . . . 77

P

parallel . . . *see* concurrent
partial operation events . . . 116
partial order
complete . . . 148
partial specifications . . . 31
port . . . **58**
pre-history . . . **121**
predicate language . . . 133, 134, 141, 224
process algebras . . . 236
process control . . . 3, 7
process term language . . . 46, 60, 238
process variable . . . 243
processes . . . 236
promotion . . . 75
schema . . . 75, 80

R

reactive . . . 7, 15, 262

real numbers 122
real-time 3, 183, 235
  constraints 7, 15, 37, 39
reduction process . 66, 69, 72, 78, 86, 162
refinement 4, 5, 21, **151–167**, 226
  order 255
  techniques 158, 223
refusals **124**
reliability 3, 8, 15
renaming **94**
retrieve relations 152
reuse 4, 21, 33, 83, 176, 224
RSL **24–25**, 33

S

safety 13, 207, 225, 259
  constraints 12, 207
safety-critical 3, 7, 15, 214, 225
schema calculus 235
section
  `BEHAVIOUR` 60
  `BEHAVIOURAL PROPERTIES` 62
  `ENVIRONMENTAL ASSUMPTIONS` . 58
  `INTERFACE` 57
  `I/O RELATIONS` 61
  `LOCAL` 58
  `OPS & PREDS` 59
  `STATE` 58
  `STATE PROPERTIES` 62
  `SUBUNITS` 71
  `TYPES & CONSTANTS` 55
sections (RT-Z) 54
sections (Z) 233
semantic integration 144
semantic metalanguage 112
semantics 4
  denotational 5, 19, 25, **103–146**, 223, 224, 229, 234
  operational **24**, 229, 238
separation of concerns 53, 70, 223
simulation 16
  downward 158
  upward 158
single-event approach 48
software design 11, 45
  specification 11, 14
software engineering 53
software requirements
  elicitation 10
  specification 11, 13
specification macros 43, 141, 259
specification units 5, **53**
  abstract **61–62**
  concrete **54–55**
  parametrised 95
  retrieve **152**
spin 246
state guards 21
state-based techniques 158
state-oriented 46, 111, 234
state-transition systems 112, 158
$State_{Ext}$ 88
static structure 37, 39, 45
stimulus–response behaviour . 37, 46, 60, 197
structuring operators 5, 20, 224, 225
SubunitStates 74
syntax 4, 303
system **11**
system design 10, 45
  specification 10
system model 4, 11, 23, 29, 172
system requirements
  analysis 9
  specification 9
systems engineering 53

T

TCCS 263
TCOZ **20–21**, 33, 103, 176–180, 183
temporal logic 30
temporal ordering 39, 62, 247
theorem prover 230
time additivity 244
time closure 244
time determinism 244
time domain 122
time interpolation 244
time-guarded **247**
time-variant state properties . . **43**, 62, 92, 141
timed CSP 3, 20, **235–263**
timed events **124**, 247
timed failures **125**, 128, 247, 249

timed failures model . 124, 128, **247**, **249**
  finite (FTF) . . . 256
timed failures semantics . . . 126
timed failures/states model . 63, 105, 109, 111, 141
timed observations . . . 40, 103, 108
timed refusals . . . 21, 40, 103, **125**, 248
timed specifications . . . 258
timed states . . . 40, 103, 108, **141**
timed traces . . . 21, 40, 103, **124**, 247
timestop . . . 246
TLA$^+$ . . . **25–26**, 54
tool support . . . 230, 233
TPL . . . 264
traces . . . **124**, 127
transformation rules . . . 66, 69, 82, 86
TTM/RTTL . . . 264

U

urgency . . . 242, 245

V

value domain . . . 44, 55, 112
verification . . . 21, 260
  techniques . . . 220, 226
viewpoint-oriented
  approaches . . . 31
  combinations . . . **31**
viewpoints
  formal setting . . . **27–28**
  in Z . . . **28–29**

W

well-defined . . . 106–108
well-formed . . . 108, 117–121
well-formedness condition . . . 117–118
well-timed . . . **247**, 256
wide-spectrum language . . . 24

Z

Z notation . . . 3, 18, 233–235
Z part . . . 39, 44, 45, 105
Z Standard . . . 73, 109, 233
Zeno . . . 246
Zermelo–Fraenkel . . . 109, 234

Index of Semantic Definitions

Symbols

$[\![-]\!]^{\mathcal{P}}_{ETFM}$ **135**
$[\![-]\!]^{\epsilon}_{\alpha}$ **141**
$[\![-]\!]^{\epsilon}_{Ref}$ **140**
$[\![-]\!]^{\epsilon}_{TRef}$ **139**
$[\![-]\!]^{\epsilon}_{TTr}$ **139**
$[\![-]\!]^{\epsilon}_{Zval}$ **138**
$[\![-]\!]^{\mathcal{P}}_{TFSM}$ **142**

A

ASpecUnit **146**
AssocSubspace **115**

B

Binding **114**

C

ChannelValue **124**
chan_of **124**
ChanSgn **112**
ConcOps **119**
CSpecUnit **113**

E

Event **124**
events_of **117**
ev_of **125**
ev_of_op **127**
Exec **116**

F

Failure **124**
$failures_Z$ **128**
FinRefusalTokenSet **125**

H

HalfOpenInterval **125**
Histories **121**
History **117**

I

InitModel **114**
Inputs **114**
Invoc **116**
Invocations **117**
InvTermComb **120**

M

MaxWellFormed **119**

O

OP_APP **114**
OpModel **115**
op_name **114**
op_params **114**
OpSeqEffects **116**
OpSetNonInterference **116**
ops_of **117**
ORC **124**
Outputs **114**

P

PartialOpSeq **117**
PlainOf **114**
POP_APP **116**
pop_name **116**
pop_params **116**
PreHist **121**
Primed **114**
PrimedOf **114**
ProcEnv **112**

R

Refusal **124**
RefusalToken **125**

S

simultan_of **117**
StateModel **114**
states_of **117**
StateVars **114**

T

$\mathbb{T}$ **123**
$\mathbb{T}^+$ **123**
Term **116**
$tfs_of_model_{EXT}$ **145**
$tfs_of_model_{INT}$ **145**
TimedEvent **125**
TimedFailure **125**
$timed\,failures_{ETCSP}\,[\![-]\!]_{_}$ **129**
$timed\,failures_{ETFM}\,[\![-]\!]_{_}$ **133**
TimedFailuresModel **126**
$timed\,failures\,states_A\,[\![-]\!]$ **147**
$timed\,failures\,states_C\,[\![-]\!]$ **146**
$timed\,failures_Z$ **128**
TimedRefusal **125**
TimedState **141**
$timed\,states_Z$ **144**
timedst_of_stseq **144**
TimedTrace **125**
time_of **125**
Trace **124**
$traces_Z$ **127**

V

val_of **124**

W

$WellFormed_1$ **118**
$WellFormed_2$ **118**
$WellFormed_3$ **119**