A Constraint Oriented Proof Methodology based on Modal Transition Systems

Kim G. Larsen[1]
Bernhard Steffen[2]
Carsten Weise[3]

ABSTRACT We present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems allowing loose state-based specifications, which can be refined by successively adding constraints. Key concepts of our method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*. Central to the method is the use of *Parametrized* Modal Transition Systems. The method easily transfers to real-time systems, where the main problem are parameters in timing constraints.

1 Introduction

The use of formal methods and in particular formal verification of concurrent systems, interactive or fully automatic, is still limited to very specific problem classes. For state-based methods this is mainly due to the state explosion problem: the state graph of a concurrent systems grows exponentially with the number of its parallel components – and with the number of clocks in the real-time case –, leading to an unmanageable size for most practically relevant systems. Consequently, several techniques have been developed to tackle this problem. Here we focus on the four main streams and do not discuss the flood of very specific heuristics. Most elegant and ambitious are *compositional* methods (e.g. [ASW94, CLM89, GS90][4]), which due to the nature of parallel compositions are unfortunately rarely applicable. *Partial order* methods try to avoid the state explosion problem by

---

[1]University of Aalborg, BRICS, kgl@iesd.auc.dk
[2]University of Passau, steffen@fmi.uni-passau.de
[3]University of Technology Aachen, carsten@informatik.rwth-aachen.de
[4]In contrast to the first reference, the subsequent two papers address compositional reduction of systems rather than compositional verification.

suppressing unnecessary interleavings of actions [GW91, Val93, GP93]. Although extremely successful in special cases, these methods do not work in general. In practice, *Binary Decision Diagram*-based codings of the state graph are successfully applied to an interesting class of systems, see e.g. [Br86, BCMDH90, EFT91]. These codings of the state graph do not explode directly, but they may explode during verification, and it is not yet fully clear when this happens. All these techniques can be accompanied by *abstraction*: depending on the particular property under investigation, systems may be dramatically reduced by suppressing details that are irrelevant for verification, see e.g. [CC77, CGL92, GL93]. Summarizing, all these methods cover very specific cases, and there is no hope for a uniform approach. Thus more application specific approaches are required, extending the practicality of formal methods.

We present a constraint-oriented state-based proof methodology for concurrent software systems which exploits compositionality and abstraction for the reduction of the verification problem under investigation. Formal basis for this methodology are Modal Transition Systems (MTS) [LT88] allowing loose state-based specifications, which can be refined by successively adding constraints. In particular, this allows extremely fine-granular specifications, which are characteristic for our approach: each aspect of a system component is specified by a number of independent constraints, one for each parameter configuration. This leads to a usually infinite number of extremely simple constraints which must all be satisfied by a corresponding component implementation. Beside exploiting compositionality in the standard (vertical) fashion, this extreme component decomposition also supports a horizontally compositional approach, which does not only separate proof obligations for subcomponents or subproperties but also for the various parameter instantiations. This is the key for the success of the following three step reduction, which may reduce even a verification problem for infinite state systems to a small number of automatically verifiable problems about finite state systems:

- *Separating the Proof Obligations.* Sections 4 and 5 present a proof principle justifying the separation and specialization of the various proof obligations, which prepare the ground for the subsequent reduction steps.

- *Skolemization.* The separation of the first step leaves us with problems smaller in size but larger in number. Due to the nature of their origin, these problems often fall into a small number of equivalence classes requiring only one prototypical proof each.

- *Abstraction.* After the first two reduction steps there may still be problems with infinite state graphs. However, the extreme specialization of the problem supports the power of abstract interpretation, which finally may reduce all the proof obligations to finite ones.

Our proof methodology is not complete, i.e., there is neither a guarantee for the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation for the success of the succeeding steps or the adequate abstraction for the final verification. Still, as should be clear from the examples in the paper, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Of course, the more complex the system structure the more involved will be the required search of appropriate granularity and abstraction.

Whereas complex data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, this extension does not affect the possibility of a finite state reduction. For the real-time case, the basis are Timed Modal Transition Systems (TMS) [CGL93], where (weak) refinement is decidable. The TMS tool Epsilon (see again [CGL93]) can be used to find the refinements on demand.

However, in this paper *parametrized* timed modal transition systems are used. Parameters may appear either in actions (so-called *parametrized actions*) or in timing constraints. Due to infinite parameter sets, specifications may in general have an infinite number of actions. Our method however aims at reducing this set of actions to a (small) finite one, such that automatic analysis of the transition systems is possible. The method does not apply to timing parameters, although we will demonstrate how to reduce them in our particular examples. The main problem with timing parameters is that existing tools cannot deal with both, parameters and refinement.

We demonstrate our methodology by two examples: an extremely simple problem of pipelined buffers, and a specification and verification problem of a Remote Procedure Call (RPC) posed by Broy and Lamport ([BL93]). The method is explained step by step by applying it first to the simple example and afterwards to the RPC problem in order to indicate that the methods scales up. Both problems have untimed and timed versions including even parameters in the timing constraints. The specific constellation, however, allows us to capture these parameters.

The next section recalls the basic theory of Modal Transition Systems, which we use for system specification. Thereafter we describe the RPC problem. The following sections explain our method in detail. Section 4 presents our notion of projective views and discusses the first reduction step. The subsequent two sections are devoted to the second and third reduction step, while Section 7 shows how to extend our method to real time systems over a dense time domain. Finally, Section 8 summarizes our conclusion and directions to future work.

2 Modal Transition Systems

In this section we give a brief introduction to the existing theory of modal transition systems. We assume familiarity with CCS. For more elaborate introductions and proofs we refer the reader to [LT88, HL89, Lar90].

When specifying reactive systems by traditional Process Algebras like e.g. CCS [Mil89], one defines the set of action transitions that can be performed (or observed) in a given system state. In this approach, any valid implementation *must* be able to perform the specified actions, which often constrains the set of possible implementations unnecessarily. One way of improving this situation within the framework of operational specification is to allow specifications where one can explicitly distinguish between transitions that are *admissible* (or allowed) and those that are *required*. This distinction allows a much more flexible specification and a much more generous notion of implementation, and therefore improves the practicality of the operational approach. Technically, this is made precise through the following notion of *modal transition systems*:

Definition 2.1. *A modal transition system is a structure* $\mathcal{S} = (\Sigma, A, \rightarrow_{\Box}, \rightarrow_{\Diamond})$*, where* Σ *is a set of states,* A *is a set of actions and* $\rightarrow_{\Box}, \rightarrow_{\Diamond} \subseteq \Sigma \times A \times \Sigma$ *are transition relations, satisfying the consistency condition* $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$. □

Intuitively, the requirement $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$ expresses that anything which is required should also be allowed hence ensuring the consistency of modal specifications. When the relations $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ coincide, the above definition reduces to the traditional notion of labelled transition systems.

Syntactically, we represent modal transition systems by means of a slightly extended version of CCS. The only change in the syntax is the introduction of two prefix constructs $a_{\Box}.P$ and $a_{\Diamond}.P$ with the following semantics: $a_{\Diamond}.P \xrightarrow{a}_{\Diamond} P$, $a_{\Box}.P \xrightarrow{a}_{\Box} P$ and $a_{\Box}.P \xrightarrow{a}_{\Diamond} P$. The semantics for the other constructs follow the lines of CCS in the sense that each rule has a version for $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ respectively. We will call this version of CCS *modal* CCS.

As usual, we consider a design process as a sequence of *refinement steps* reducing the number of possible implementations. Intuitively, our notion of when a specification S refines another (weaker) specification T is based on the following simple observation. Any behavioural aspect *allowed* by S should also be allowed by T; and dually, any behavioural aspect which is already guaranteed by the weaker specification T must also be guaranteed by S. Using the derivation relations $\rightarrow_{\Box}$ and $\rightarrow_{\Diamond}$ this may be formalized by the following notion of *refinement*:

Definition 2.2. *A refinement* $\mathcal{R}$ *is a binary relation on* Σ *such that whenever* $S\,\mathcal{R}\,T$ *and* $a \in A$ *then the following holds:*

1. *Whenever* $S \xrightarrow{a}_{\Diamond} S'$*, then* $T \xrightarrow{a}_{\Diamond} T'$ *for some* T' *with* $S'\,\mathcal{R}\,T'$*,*

2. *Whenever* $T \xrightarrow{a}_{\Box} T'$, *then* $S \xrightarrow{a}_{\Box} S'$ *for some* S' *with* $S' \mathcal{R} T'$.

S is said to be a refinement of T in case (S, T) is contained in some refinement $\mathcal{R}$. We write $S \triangleleft T$ in this case. $\Box$

Note that when applied to traditional labelled transition systems (where $\rightarrow = \rightarrow_{\Box} = \rightarrow_{\Diamond}$) this defines the well–known bisimulation equivalence [Par81, Mil89]. – Using standard techniques, one straightforwardly establishes that $\triangleleft$ is a preorder preserving all modal CCS operators.

$\triangleleft$ allows *loose* specifications. This important property can be best explained by looking at the 'weakest' specification $\mathcal{U}$ constantly allowing any action, but never requiring anything to happen. Operationally, $\mathcal{U}$ is completely defined by $\mathcal{U} \xrightarrow{a}_{\Diamond} \mathcal{U}$ for all actions a. It is easily verified that $S \triangleleft \mathcal{U}$ for any modal specification S.

Intuitively, S and T are *independent* if they are not contradictory, i.e. any action required by one is not constraint by the other. The following formal definition is due to the fact that for S and T to be *independent* all 'simultaneously' reachable processes S' and T' must be independent too:

Definition 2.3. *An independence relation $\mathcal{R}$ is a binary relation on Σ such that whenever $S \mathcal{R} T$ and $a \in A$ then the following holds:*

1. *Whenever* $S \xrightarrow{a}_{\Box} S'$, *there is a unique* T' *such that* $T \xrightarrow{a}_{\Diamond} T'$ *and* $S' \mathcal{R} T'$,
2. *Whenever* $T \xrightarrow{a}_{\Box} T'$, *there is a unique* S' *such that* $S \xrightarrow{a}_{\Diamond} S'$ *and* $S' \mathcal{R} T'$,
3. *Whenever* $S \xrightarrow{a}_{\Diamond} S'$ *and* $T \xrightarrow{a}_{\Diamond} T'$ *then* $S' \mathcal{R} T'$.

S and T are said to be independent *in case (S, T) is contained in some independence relation $\mathcal{R}$.* $\Box$

Note in particular that two specifications are independent if none of them requires any actions. Independence is important, as it allows to define conjunction on modal transition systems by:

$$\frac{S \xrightarrow{a}_{\Box} S' \quad T \xrightarrow{a}_{\Diamond} T'}{S \wedge T \xrightarrow{a}_{\Box} S' \wedge T'} \qquad \frac{S \xrightarrow{a}_{\Diamond} S' \quad T \xrightarrow{a}_{\Box} T'}{S \wedge T \xrightarrow{a}_{\Box} S' \wedge T'}$$

$$\frac{S \xrightarrow{a}_{\Diamond} S' \quad T \xrightarrow{a}_{\Diamond} T'}{S \wedge T \xrightarrow{a}_{\Diamond} S' \wedge T'}$$

Of course, $S \wedge T$ is always a well-defined modal specifications (i.e. any required transition is also allowed), and in fact, for independent arguments S and T it defines their *logical* conjunction:

Theorem 2.4. Let S and T be independent modal specifications. Then $S \wedge T \lhd S$ and $S \wedge T \lhd T$. Moreover, if $R \lhd S$ and $R \lhd T$ then $R \lhd S \wedge T$.

In order to compare specifications at different levels of abstraction, it is important to abstract from transitions resulting from internal communication.

This can be done as usual: For a given modal transition system $\mathcal{S} = (\Sigma, A \cup \{\tau\}, \rightarrow_\Box, \rightarrow_\Diamond)$ we derive the modal transition system $\mathcal{S}_\varepsilon = (\Sigma, A \cup \{\varepsilon\}, \Rightarrow_\Box, \Rightarrow_\Diamond)$, where $\stackrel{\varepsilon}{\Rightarrow}_\Box$ is the reflexive and transitive closure of $\stackrel{\tau}{\rightarrow}_\Box$, and where $T \stackrel{a}{\Rightarrow}_\Box T', a \neq \varepsilon$, means that there exist T'', T''' such that

$$T \stackrel{\varepsilon}{\Rightarrow}_\Box T'' \stackrel{a}{\rightarrow}_\Box T''' \stackrel{\varepsilon}{\Rightarrow}_\Box T'$$

The relation $\Rightarrow_\Diamond$ is defined in a similar manner.

The notion of *weak refinement* can now be introduced as follows: S weakly refines T in $\mathcal{S}$, $S \unlhd T$, iff there exists a refinement relation on $\mathcal{S}_\varepsilon$ containing S and T.

Weak refinement $\unlhd$ essentially enjoys the same pleasant properties as $\lhd$: it is a preorder preserved by all modal CCS operators except $+$ [HL89] (including restriction, relabelling and hiding). Moreover, for ordinary labelled transition systems weak refinement reduces to the usual notion of weak bisimulation ($\approx$).

In our examples, we will deal with weak refinement and (in general) infinite action sets. In the context of weak refinement, forbidding internal τ-actions in a constraint is a severe and unnatural restriction. We therefore consider only *saturated* versions of specifications, which *always* allow τ-steps by having τ-may-loops at each of their states. Note that each process S can easily be saturated by adding τ-loops. Moreover, a process S and its saturated version S^+ are mutual weak refinements of each other:

$$S \unlhd S^+ \quad \text{and} \quad S^+ \unlhd S$$

Thus they are substitutive in the context of parallel composition and hiding. The restriction to saturated specifications, therefore, does not cause any limitation in our setting.

The use of saturated transition systems has a major technical advantage: the definitions of conjunction and independence work for weak refinement in the same way as before for strong refinement. This is not true in the general case, which requires tedious adaptations.

Thus let us assume in the following that all transitions systems are saturated. This guarantees the validity of some important rules:

Proposition 2.5. *Assume a (possible infinite) index set I, a subset $J \subseteq I$, a set L of actions, two families of modal transition systems $S_i, T_i (i \in I)$ and a modal transition system T. Let the families S_i, T_i be pairwise independent, as well as the processes $(S_i \mid T)$. Then the following laws for conjunctions hold:*

1. *Adding constraints refines a specification:*

$$\bigwedge_{i \in I} S_i \unlhd \bigwedge_{j \in J} S_j$$

2. *Conjunction is preserved by refinement:*

$$\forall i \in I.(S_i \unlhd T_i) \quad \textit{implies} \quad \bigwedge_{i \in I} S_i \unlhd \bigwedge_{i \in I} T_i$$

3. *Conjunction distributes over parallel composition:*

$$(\bigwedge_{i \in I} S_i) \| T \unlhd \bigwedge_{i \in I} (S_i \| T) \qquad \textit{and} \qquad \bigwedge_{i \in I} (S_i \| T) \unlhd (\bigwedge_{i \in I} S_i) \| T$$

4. *Conjunction distributes over restriction:*

$$(\bigwedge_{i \in I} S_i) \setminus L \unlhd \bigwedge_{i \in I} (S_i \setminus L) \qquad \textit{and} \qquad \bigwedge_{i \in I} (S_i \setminus L) \unlhd (\bigwedge_{i \in I} S_i) \setminus L$$

The proofs for all these claims are straightforward. As an example, we give a proof for the left hand side of the third part.

Starting from $(\bigwedge_{i \in I} S_i) \mid T$ it is immediate for any $j \in I$ that

$$(\bigwedge_{i \in I} S_i) \mid T \lhd S_j \mid T$$

holds. As this is independent of j, we directly find that $(\bigwedge_{i \in I} S_i) \mid T$ is a refinement of the conjunction $\bigwedge_{i \in I} (S_i \mid T)$. $\square$

In our examples, certain patterns of modal transition systems will be found frequently. Assuming an action set `Act` and subsets α, β and γ, Fig. 1 depicts two of these patterns, which will be used in our examples. We use the following "abbreviations" for these transition systems:

$$\mathbf{AG}_{\beta} \neg \alpha \tag{1.1}$$

for the left hand side transition system and

$$\mathbf{AG}_{\mathtt{Act}} ([\, \alpha \,]\, \mathbf{AG}_{\gamma} \neg \beta \,) \tag{1.2}$$

for the right hand side system.

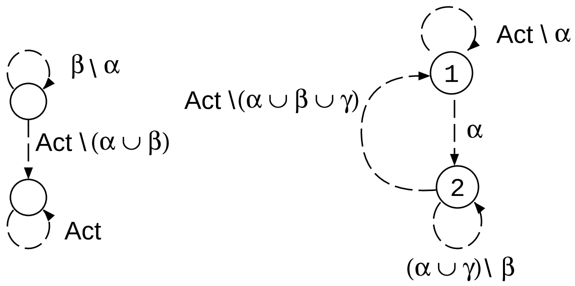


FIGURE 1. Typical Patterns of Modal Transition Systems

The intuition behind these transition system is "as long as only actions from β are taken, no actions from α may be allowed"[5] and "after an action from α has been taken, no actions from β are allowed as long as we only traverse actions from γ". The given "abbreviations" are in fact formulae of a parameterized version of CTL. As we cannot discuss the relationship between CTL and modal transition systems here, the interested reader is referred to [CES83] for standard CTL and to [Ste93] to learn about an extension of CTL which is powerful enough to capture the considered modal transition systems.

3 The Remote Procedure Call Problem

We demonstrate our method by applying it to a specification problem given by Broy and Lamport. Due to space limitations we can only present part of the problem.

The original problem consists of a *memory component* and an *RPC mechanism.* The memory component accepts read and writes from several processes, and returns the requested values (none in case of write) or raises an exception. The only exception here is *memory failure*, i.e. the memory could not read from/write to the hardware. A component in which exceptions do never occur is called a *reliable memory.*

The processes are connected to the memory component via an RPC (Remote Procedure Call) mechanism. The RPC mechanism simply forwards calls from the processes to the memory, and returns from the memory to the processes. The RPC should be transparent to the user, i.e. the composition of the memory component and the RPC should be an implementation of the memory. This is what we will call the *untimed RPC problem.*

In the real-time case, the time to forward calls and returns by the RPC should be no more than δ. Further an exception should be raised if a call to the RPC does not return within $2\delta + \varepsilon$ seconds. We will prove that if all

[5]Actions outside β can be regarded as ways to *escape* the 'universal' proof obligation.

calls to a reliable memory return within ε seconds, then the composition of the RPC and the reliable memory is an implementation of the reliable memory. This is the *timed RPC problem.*

The following is an informal specification of the memory component M, concentrating on write calls only. We assume sets `procId` of process identifiers, `memLocs` of memory locations and `memVals` of memory values, with typical elements `id`, `loc` and `val` resp. We will often use Z as an abbreviation for the product of the three sets, i.e. $Z := \texttt{procId} \times \texttt{memLocs} \times \texttt{memVals}$, with typical element $z \in Z$.

The events occurring in the memory component are described by *parameterized actions*, taking arguments from `procId`, `memLocs` and `memVals`. The actions of M are:

$\texttt{mWr}(\texttt{id}, \texttt{loc}, \texttt{val})$:	write-call from process `id` of value `val` to location `loc`
$\texttt{write}(\texttt{id}, \texttt{loc}, \texttt{val})$:	atomic write of value `val` to location `loc` initiated by process `id`
$\overline{\texttt{mRetWr}}(\texttt{id})$:	send return from a write-request to process `id`
$\overline{\texttt{mFail}}(\texttt{id})$:	signal memory failure to process `id`

The I/O-behaviour of the memory component M is given in Fig. 2.

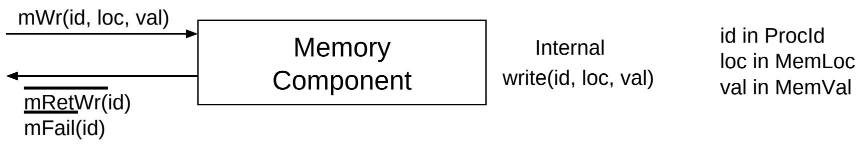


FIGURE 2. I/O-Behaviour of Memory Component

The specification of the (reliable) memory component is a conjunction of the following properties:

$P0$ The memory component engages in actions only when it is called

$P1$ Each write operation (successful or not) performs a sequence of zero or more atomic writes of the correct value to the correct location at some time between the call and return. For a successful write operation, there must be at least one atomic write.

$P2$ A memory failure is never raised.

Clearly, the memory component M is specified by the conjunction of $P0$ and $P1$, while the reliable memory M_R is the conjunction of the M and $P2$. Note that for fixed `id` the last property can be easily specified by

$$\mathbf{AG}_{\mathtt{Act}} \neg \{ \overline{\mathtt{mFail}(\mathtt{id})} \}$$

The RPC R simply hands calls and returns (including the memory failure exception) through. These are the actions of the RPC:

$\mathtt{rWr}(\mathtt{id}, \mathtt{loc}, \mathtt{val})$	:	remote write of value `val` to location `loc` issued by process `id`
$\overline{\mathtt{rRetWr}}(\mathtt{id})$	:	return from remote write issued by process `id`
$\overline{\mathtt{rFail}}(\mathtt{id})$	:	RPC returns an exception from a call issued by process `id`
$\overline{\mathtt{mWr}}(\mathtt{id}, \mathtt{loc}, \mathtt{val})$	:	send a write of value `val` to location `loc` initiated by process `id`
$\mathtt{mRetWr}(\mathtt{id})$	:	return from a write initiated by process `id`
$\mathtt{mFail}(\mathtt{id})$	:	memory component raised a memory failure

The I/O-behaviour of the combined components can be depicted as in Fig. 3:

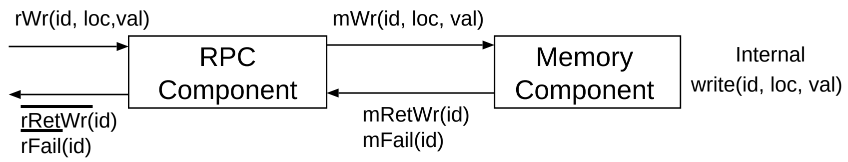


FIGURE 3. Combination of RPC and Memory

In the next sections, we will explain our method directly using a much simpler example. At the end of each section we show how our method transfers to the RPC problem. We start with the untimed case.

4 Projective Views

In the following, we present, motivate and clarify our proof methodology by means of a minimal example, which is just sufficient to explain the various phenomena.

Consider the parallel system in Fig. 4. Here two parameterized, disposable component media (supposed to transmit natural numbers) A and B are composed in parallel yielding a pipeline. Informally, the component A is supposed to input a natural number on port a, then output this number on port b after which it will terminate. The behaviour of B is similar. Using modal transition systems, the parallel system may be expressed as follows:

$$\Big(\underbrace{a_\Box x . \overline{b_\Box} x}_{A} \mid \underbrace{b_\Box x . \overline{c_\Box} x}_{B} \Big) \backslash \{b\}$$

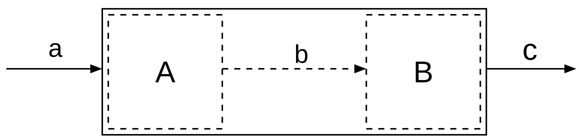


FIGURE 4. A Pipe Line of Two Disposable Media

The behaviour of A and B are given by the two infinite–width transition systems of Fig. 5. However, rather than using these direct specifications of A

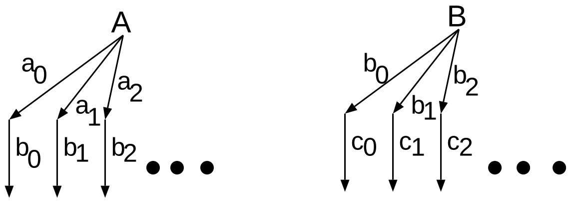


FIGURE 5. Behaviour of A and B.

and B we specify the two components behaviour using projective views A_n and B_n; one view for each possible natural number n. The projective view A_n specifies the *constraints* on the behaviour of the component A when focusing on transmission of the value n; this constraint can be expressed as the modal transition system A_n given in Fig. 6(where we use solid lines for must- and dotted lines for may-transition).

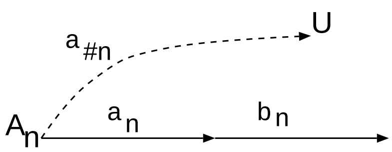


FIGURE 6. Projective View A_n

Here $a_{\neq n}$ denotes all labels of the form a_m where $m \neq n$; also $\mathcal{U}$ denotes the universal modal transition system constantly allowing all actions. Note that this 'n-th view' imposes no constraint on the behaviour of A when transporting values different from n. The complete specification of the component A is the conjunction of all projective views[6] A_n. In fact it is easy to establish the following facts:

$$A \trianglelefteq \bigwedge_n A_n \quad \text{and} \quad \bigwedge_n A_n \trianglelefteq A \tag{1.3}$$

[6]Note that all the projective views of A are pairwise independent.

where A refers to the (infinite) transition system of Fig. 5. Obviously, we may obtain similar projective views B_n for component B.

Let us now consider the problem of verifying that the overall system $\Big(A \,|\, B\Big)\backslash\{b\}$ is observationally equivalent to the system $C = a_\square x.\overline{c_\square} x$ (i.e. a slightly different disposable media). As A, B and C are standard transitions systems, i.e., everything allowed is also required, this problem is equivalent to showing

$$\Big(A \,|\, B\Big)\backslash\{b\} \ \trianglelefteq \ C$$

Thus (1.3), together with the observation that also C may be expressed as a conjunction of an infinite number of constraints C_n, leaves us with the following refinement problem:

$$\Big(\bigwedge_n A_n \,|\, \bigwedge_n B_n\Big)\backslash\{b\} \ \trianglelefteq \ \bigwedge_n C_n \tag{1.4}$$

4.1 Application to the RPC problem

We give modal transition systems for the specification of properties $P0, P1$ and $P2$ of the memory component. Therefore we split $P1$ into two properties $P1a, P1b$ meaning

$P1a$ A write-call from process `id` cannot return unless an atomic write is performed.

$P1b$ As long as a write-call from process `id` has not returned, no atomic write to a wrong location or of a false value occurs

The labels in the following specifications are sets of actions (called *abstracted actions*). A single action is a shorthand for the set containing this and only this action. For the other sets, we use the usual set-theoretic connectives, and a dot-notation, where a parametrized action with dots as parameters means "the set of all actions where the dotted position is replaced by all legal values for the parameter", e.g. for a fixed $\mathtt{id} \in \mathtt{procId}$, $\mathtt{mWr}(\mathtt{id}, ., .)$ is the set $\{\mathtt{mWr}(\mathtt{id}, \mathtt{loc}, \mathtt{val}) \,|\, \mathtt{loc} \in \mathtt{memLocs}, \mathtt{val} \in \mathtt{memVals}\}$.

The properties $P1a$ and $P1b$ are easily expressed by the following abbreviations of modal transition systems:

$$\mathbf{AG}_{\mathtt{Act}} \Big([\,\mathtt{mWr}(\mathtt{id}, ., .)\,]\, \mathbf{AG}_{\mathtt{Act}\backslash\mathtt{write}(\mathtt{id},.,.)} \neg \{\, \overline{\mathtt{mRetWr}(\mathtt{id})} \,\} \Big)$$
$$\mathbf{AG}_{\mathtt{Act}} \Big([\,\mathtt{mWr}(\mathtt{id}, \mathtt{loc}, \mathtt{val})\,]\, \mathbf{AG}_{\mathtt{Act}\backslash\mathtt{write}(\mathtt{id},\mathtt{loc},\mathtt{val})} \neg \{\, \overline{\mathtt{mRetWr}(\mathtt{id})} \,\} \Big)$$

Our specification assumes that calls from different processes are handled concurrently. As calls from different processes do not interfere, no actions parametrized with an identifier other than `id` is constrained in the specifications of calls from process `id`. This is modelled by allowing all actions with an identifier different from the fixed `id` in any state. Instead of adding to each state a loop where all these actions are allowed, we draw boxes meaning "a state with a loop for all non-`id` actions". By this the conjunction of the specifications for all processes is the same as their parallel composition.

The modal transition systems which specify the properties for a fixed value `id` are given in Fig. 7.

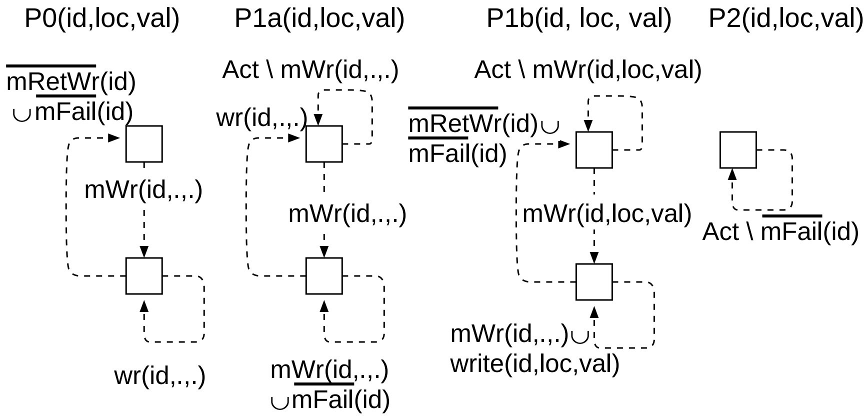
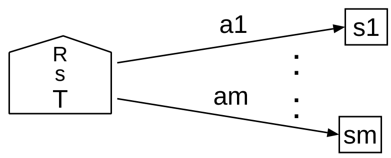


FIGURE 7. MTS for properties $P0$, $P1a$, $P1b$ and $P2$

The transition systems for $P1a, P1b$ and $P2$ are the expansions of the "abbreviated" transition systems (cf. Fig. 1), while the transition system for $P0$ was defined directly. Note that only $P1b$ really depends on `loc` and `val`, and that the properties $P0, P1a, P1b$ and $P2$ are the conjunctions of the above modal specifications over all $z \in Z$.

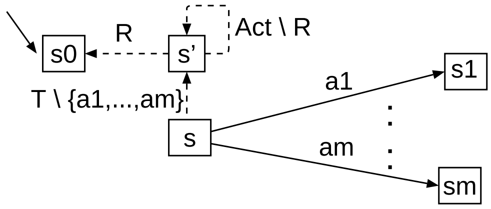
Let $M(z)$ be the conjunction $P0(z) \wedge P1a(z) \wedge P1b(z)$, and $M_R(z) = M(z) \wedge P2(z)$. The memory component M is the conjunction of $M(z)$ over all $z \in Z$.

Let `Act` be the set of all actions. For two sets $R \subseteq$ `Act` (*return set*) and $T \subseteq$ `Act` (*tolerance set*), a state s and actions $a_1, \ldots, a_m \in$ `Act`. Then we use the following *macro state* for the specification of the RPC:



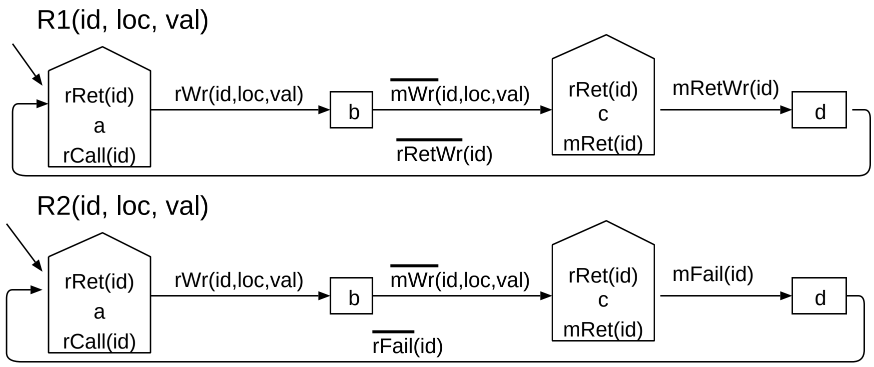
Here the edges leaving the "macro state" can be either may- or must-transition.

For a given transition system with start state s_0 and an auxiliary state s' not already in the transition system, this is meant to expand to



i.e. state s tolerates any action from T. If the behaviour of a tolerated action is already specified by an outgoing edge, nothing new happens. Otherwise, the system goes to the auxiliary state s', where it accepts any action until a return action (from R) occurs. Return actions take the system back to the start state.

There are two main projective views of the RPC. In the first view, a write is handed through and a return received from the memory. In the second view, instead of a return a memory failure is received. These two views $R_1(\mathtt{id}, \mathtt{loc}, \mathtt{val})$ and $R_2(\mathtt{id}, \mathtt{loc}, \mathtt{val})$ are given in the following picture:



The sets in the macro states are defined as follows:

$$
\begin{array}{rcl}
\mathtt{rCall(id)} & := & \mathtt{rWr}(\mathtt{id}, ., .) \\
\mathtt{rRet(id)} & := & \overline{\mathtt{rRetWr}}(\mathtt{id}) \cup \overline{\mathtt{rFail}}(\mathtt{id}) \\
\mathtt{mRet(id)} & := & \mathtt{mRetWr}(\mathtt{id}) \cup \mathtt{mFail}(\mathtt{id})
\end{array}
$$

While it is natural to use must-transitions in this specification, the lack of must-transitions in the memory component allows us to weaken these must transitions to may transitions without affecting the correctness of a successful proof. This guarantees the well-definedness of conjunction, as all our specifications are now independent.

Let $R(z) := R_1(z) \wedge R_2(z)$. The untimed specification of the RPC R is the conjunction of $R(z)$ over all z.

Let f be a relabelling mapping all actions of the RPC to the appropriate actions of the memory component, and $A := \texttt{rWr}(.,.,.) \cup \texttt{rRetWr}(.) \cup \texttt{rFail}(.)$ and $H := \texttt{write}(.,.,.)$. Then the untimed verification problem is

$$\Big(R \,|\, M/H\Big)\backslash A[f] \;\trianglelefteq\; M/H \tag{1.5}$$

where the internal actions of the memory (i.e. the atomic writes) are hidden.

5 Sufficient Proof Condition

As a conjunction is a refinement of every of its components (cf. Prop. 2.5), the proof of (1.4) can be reduced to the verification of

$$\Big(\bigwedge_{i\in N} A_i \,|\, \bigwedge_{i\in N} B_i\Big)\backslash\{b\} \;\trianglelefteq\; C_j$$

for each natural j. Note that this is even a necessary condition for our claim.

This reduction alone would not gain much. Here however it turns out that it is sufficient to verify

$$\forall j \in N.\; \Big(A_j \,|\, B_j\Big)\backslash\{b\} \;\trianglelefteq\; C_j \tag{1.6}$$

which is intuitively clear as transmitting j through the pipeline only depends on transmitting j through its components.

The fact that (1.6) is sufficient follows from a general proof principle behind the reduction. The idea is that there is a typical pattern of refinement we need to establish. This pattern consists of a large conjunction $\bigwedge C_j$ on the right side, and a parallel composition of large conjunctions on the left side (with possible restriction). To establish such a weak refinement, it is sufficient to establish the refinement for each conjunct C_j. However, concentrating on a specific component C_j, a lot of the details of the implementation on the left side can (hopefully) be disregarded, thus it will be sufficient to restrict the proof to subsets of the conjuncts in the parallel components of the left hand side. These subsets will generally depend on j.

This is formalized by the following *sufficient proof condition*:

Theorem 5.1. *Assume index sets $I_1, \dots I_k, I$, and modal transition systems $A_i^\ell, C_j (\ell \in \{1, \dots, k\}, i \in I_\ell, j \in I)$. If there are subsets $I_{\ell,j} \subseteq I_\ell$ for each $\ell \in \{1, \dots, k\}$ and $j \in I$, such that*

$$\forall j \in I.\; \Big(\bigwedge_{i\in I_{1,j}} A_i^1 \,|\, \dots \,|\, \bigwedge_{i\in I_{k,j}} A_i^k\Big)\backslash L \;\trianglelefteq\; C_j \tag{1.7}$$

then

$$\Big(\bigwedge_{i \in I_1} A_i^1 \mid \ldots \mid \bigwedge_{i \in I_k} A_i^k \Big) \backslash L \;\trianglelefteq\; \bigwedge_{j \in I} C_j \tag{1.8}$$

holds as well.

Proof. Starting from the assumption (1.7) for an arbitrary j, we can shift all conjunctions from the inside of the formula out by using distributivity of conjunction over parallel composition and restriction:

$$\begin{aligned} \bigwedge_{i_1 \in I_{1,j}, \ldots, i_k \in I_{k,j}} (A_{i_1}^1 \mid \ldots \mid A_{i_k}^k) \backslash L \;&\trianglelefteq\; \Big(\bigwedge_{i \in I_{1,j}} A_i^1 \mid \ldots \mid \bigwedge_{i \in I_{k,j}} A_i^k \Big) \backslash L \\ &\trianglelefteq\; C_j \end{aligned} \tag{1.9}$$

Conjuncting (1.9) over all $j \in I$ gives us

$$\bigwedge_{i_1 \in I'_1, \ldots, i_k \in I'_k} (A_{i_1}^1 \mid \ldots \mid A_{i_k}^k) \backslash L \;\trianglelefteq\; \bigwedge_{j \in I} C_j$$

for subsets $I'_\ell \subseteq I_\ell$. As adding constraints refines a specification, the following is a refinement of the left hand side:

$$\bigwedge_{i_1 \in I_1, \ldots, i_k \in I_k} (A_{i_1}^1 \mid \ldots \mid A_{i_k}^k) \backslash L$$

Using the distributivity of conjunction over parallel composition and restriction once more, this can further be refined to

$$\Big(\bigwedge_{i \in I_1} A_i^1 \mid \ldots \mid \bigwedge_{i \in I_k} A_i^k \Big) \backslash L$$

Finally, the transitivity of $\trianglelefteq$ allows us to combine the last three lines in order to establish our claim. □

Of course, in general the power of this proof principle strongly depends on a good choice of the $I_{\ell,j}$, which was trivial in our example.

5.1 Application to the RPC Problem

With the same argumentation, to prove (1.5) it is sufficient to show

$$\forall z \in Z. \; \Big(R(z) \mid M(z)/H \Big) \backslash A[f] \;\trianglelefteq\; M(z)/H \tag{1.10}$$

6 Skolemization and Abstraction

So far we have reduced the overall verification problem of (1.4) to that of (1.6). At first sight this doesn't seem much of a reduction as (1.6) requires a refinement proof to be established for each natural number. Fortunately, these proofs are not really sensitive to the actual value of the natural number n. Letting k be an arbitrary natural number (or a Skolem constant) it suffices to prove:

$$\Big(A_k \,|\, B_k\Big)\backslash\{b\} \ \trianglelefteq\ C_k \tag{1.11}$$

in order to infer (1.6). Thus we are now left with the problem of establishing a *single* refinement. But still, though finite *state* the specifications A_k and B_k both have infinitely many *transitions* (as $a_{\neq k}$ is an inifinite label set).

However we can find an equivalence relation on the actions of the components which is of finite index, but still fine enough to establish the proof goal. Replacing a system with a new one gained by collapsing w.r.t. an equivalence relation is called *abstraction*.

In the following, $[s]^{\equiv}$ is the equivalence class of s under $\equiv$.

If the equivalence relation is understood from the context, we write $[s]$.

In general, an equivalence relation on states and transitions is needed, but for the examples here an equivalence relation on transitions suffices:

Definition 6.1. *Let P be a TMS over an alphabet* `Act` *with transition relations $\longrightarrow_\Box, \longrightarrow_\Diamond$. Each equivalence relation $\equiv$ on* `Act` *induces a* collapsed TMS $P^{\equiv}$ *over the alphabet* $\mathtt{Act}_{\equiv} := \{[a] \,|\, a \in \mathtt{Act}\}$ *and transition relations $\longrightarrow'_\Box, \longrightarrow'_\Diamond$ defined by*

$$\frac{p \stackrel{a}{\longrightarrow}_\Box p'}{p \stackrel{[a]}{\longrightarrow}{}'_\Box\, p'} \qquad\qquad \frac{p \stackrel{a}{\longrightarrow}_\Diamond p'}{p \stackrel{[a]}{\longrightarrow}{}'_\Diamond\, p'}$$

An equivalence relation $\equiv$ on `Act` *is* compatible *with P iff for all $a' \in [a]$ and all reachable states p, p' of P:*

$$p \stackrel{a}{\longrightarrow}_\Box p' \quad \textit{iff} \quad p \stackrel{a'}{\longrightarrow}_\Box p' \qquad \textit{and} \qquad p \stackrel{a}{\longrightarrow}_\Diamond p' \quad \textit{iff} \quad p \stackrel{a'}{\longrightarrow}_\Diamond p'$$

Compatible equivalence relations satisfy the following three properties:

Proposition 6.2. *Let P and Q be two TMS's and $\equiv$ an equivalance relation on their common alphabet compatible with P and Q. Then the following holds:*

1. *$P^{\equiv} \trianglelefteq Q^{\equiv}$ implies $P \trianglelefteq Q$,*
2. *if $[\tau] = \{\tau\}$ then $\equiv$ is compatible with $P \,|\, Q$,*

3. *if* $[\tau] = \{\tau\}$ *and for* $L \subseteq \mathtt{Act}$ *and every* $a \in \mathtt{Act}$ *either* $[a] \cap L = [a]$ *or* $[a] \cap L = \emptyset$[7], *then* $\equiv$ *is compatible with* $P \setminus L$.

Proof. 1. $P^{\equiv} \trianglelefteq Q^{\equiv}$ implies the existence of a weak refinement relation between the states of $P^{\equiv}$ and $Q^{\equiv}$. As no states are collapsed, we can use the same relation to establish $P \trianglelefteq Q$ exploiting its compatitiblity:

If Q requires an a-step, then $Q^{\equiv}$ requires an $[a]$-step by definition. As $P^{\equiv}$ is a weak refinement of $Q^{\equiv}$, it requires an $[a]$-step as well. Thus by definition P requires an a'-step for some $a' \in [a]$. Compatibility now guarantees an a'-step for every $a' \in [a]$, in particular for a itself.

The part for may-transitions follows analogously.

2. Assume $P \mid Q \xrightarrow{a}_{\Box} P' \mid Q'$. Then we must show that for all $a' \in [a]$ we have $P \mid Q \xrightarrow{a'}_{\Box} P' \mid Q'$ as well.

If $a = \tau$, then $[a] = \{a\}$, so $a' = a$. Thus the proposition is true.

If $a \neq \tau$, then w.l.o.g. $P \xrightarrow{a}_{\Box} P'$ and $Q = Q'$, and the compatibility of $\equiv$ with P guarantees $P \xrightarrow{a'}_{\Box} P'$, and therefore $P \mid Q \xrightarrow{a'}_{\Box} P' \mid Q'$.

The proof for $\xrightarrow{a}_{\Diamond}$ follows the same lines.

3. Assuming $P \setminus L \xrightarrow{a}_{\Box} P' \setminus L$, it suffices to show $P \setminus L \xrightarrow{a'}_{\Box} P' \setminus L$ for all $a' \in [a]$.

If $a = \tau$, then by the same argument as above $a' = a$, and the proposition holds.

If $a \neq \tau$, then $a \notin L$ and $P \xrightarrow{a}_{\Box} P'$. Thus the compatibility yields $P \xrightarrow{a'}_{\Box} P'$, and therefore $P \setminus L \xrightarrow{a'}_{\Box} P' \setminus L$, as the condition in 3. guarantees $a' \notin L$.

The part for $\xrightarrow{a}_{\Diamond}$ follows along the same lines. ☐

This Proposition allows us to reduce verification problems for infinite systems to problems for finite systems, as soon as an appropriate equivalence relation can be found.

For our example, let us consider the equivalence relation $\equiv$ defined by $x_k \equiv x_k$ and $x_i \equiv x_j$ whenever $i, j \neq k$, where x ranges over $\{a, b, c\}$. Further τ builds an equivalence class of its own.

Obviously, $\equiv$ is compatible with A_k, B_k and C_k. As further all conditions of Prop. 6.2 are met, $\equiv$ is also compatible with $\left(A_k{}^{\equiv} \mid B_k{}^{\equiv}\right) \setminus \{b\}$. Thus the verification of (1.4) can further be reduced to the refinement proof between the finite $\equiv$–abstracted versions of A_k, B_k and C_k

$$\left(A_k{}^{\equiv} \mid B_k{}^{\equiv}\right) \setminus \{b\} \ \trianglelefteq \ C_k{}^{\equiv} \tag{1.12}$$

which can easily be done by means of the automatic verification tool Epsilon.

[7] i.e. L is union of some equivalence classes

6.1 Application to the RPC Problem

Instead of proving (1.10) for all z, a proof for a prototypical z is sufficient here. Most of the abstraction is already carried out by using abstracted actions. Note however that the abstracted actions are in general *not* the required equivalence classes. For the RPC problem e.g. $\texttt{write}(z)$ is an equivalence class of its own, and the set $\texttt{write}(\texttt{id}, ., .) \setminus \texttt{write}(z)$ is another equivalence classes. This specific partitioning of the atomic write actions reflects the fact that we must distinguish between a write of the correct value to the correct location and all other writes from the same process.

Looking at the diagrams of Sect. 4.1 easily reveals that the resulting transition systems are small and easily in the range of the EPSILON tool.

7 Specifications with Time

The above examples can be extended to deal with real time. For the specification we use Wang Yi's Timed CCS (see [Yi91]) together with modal specifications. For details on these so called *Timed Modal Specifications* see [CGL93]. This method can be used with any totally ordered time domain, while in the following we will assume the positive real numbers.

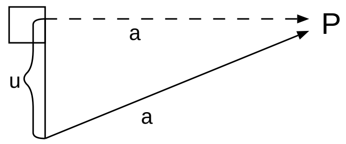
The passing of time is modelled by a delay action $\varepsilon(d)$, where d is a positive real number. The intuitive meaning of such a delay is that a time amount of d passes until the end of this action. Normal actions are enabled immediately, and can be taken at any time. As an example, the process $a_\Box x.\varepsilon(2).\overline{b_\Box} x$ can execute $a_\Box x$ at any time. Thereafter it must delay for at least two time units before it can engage in $b_\Box x$.

Further we assume *maximal progress*, i.e. a communication must be performed as soon as possible. Putting $a_\Box x.\varepsilon(2).\overline{b_\Box} x$ in parallel with $\overline{a_\Box} x.\varepsilon(3).b_\Box x$ would force the communication via channel a to take place immediately, and the communication via channel b to happen after exactly three time units.

For our specification, the macro $a[l, u]$ is convenient, where a is an action and l, u are real numbers with $l < u$. The intuition is that a process $a[l, u].P$ *may* enable a after l time units and *must* enable a after u time units. In other words, communication via a may be possible after at least l time units, and will be possible at any time after u time units.[8] This macro is defined as $a[l, u].P = (\varepsilon(l).a_\Diamond + \varepsilon(u).a_\Box).P$.

In our examples, the lower bound is always zero. The graphical presentation we use for $a[0, u].P$ is:

[8]Note that u is not a *time-out*, but a switching point between a may and a must reuirement!



Let d be a fixed real number. Then we specify a timed process $A(d)$, which reads port a and subsequently outputs its input onto port b within d time units, by $a_\square x.\overline{b}x[0,d]$. Note that this is a timed version of process A. The same construction gives timed versions $B(d)$ and $C(d)$ of B and C.

We are now going to establish that a 'pipeline' with two components with delay d should not be slower than one component with delay $2d$, i.e.

$$\Big(A(d) \mid B(d)\Big) \setminus \{b\} \;\trianglelefteq\; C(2d)$$

The same method as in the untimed case reduces the situation to

$$\Big({A_k}^{\equiv}(d) \mid {B_k}^{\equiv}(d)\Big) \setminus \{b\} \;\trianglelefteq\; {C_k}^{\equiv}(2d)$$

for a Skolem constant k and the equivalence relation of the previous section. Now, given a specific value for d this proof can be carried out using the Epsilon tool, which treats real valued timer domains by means of the clock region automaton technique (see [AD94] for details). This technique relies on integer values for all explicit timer constants in the specification, which can be achieved by multiplication with an appropriate constant in most applications. As all timer constants are multiplied by the same constant, this does not affect the principle behaviour of the system. In our example, the obvious choice for this constant is $1/d$, leaving us with the following refinement problem

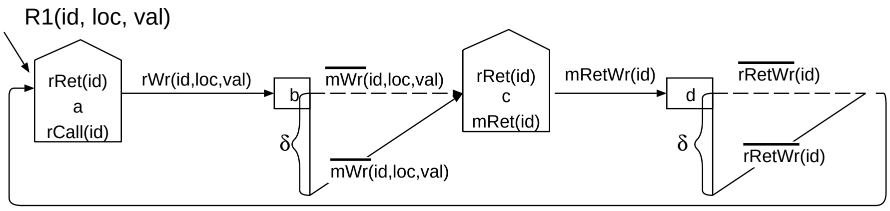
$$\Big({A_k}^{\equiv}(1) \mid {B_k}^{\equiv}(1)\Big) \setminus \{b\} \;\trianglelefteq\; {C_k}^{\equiv}(2)$$

which can be solved using Epsilon.

Note that this proof indeed covers the statement for any d. Thus even in the presence of real time, the original verification problem is reduced to a very simple, automatically solvable problem.

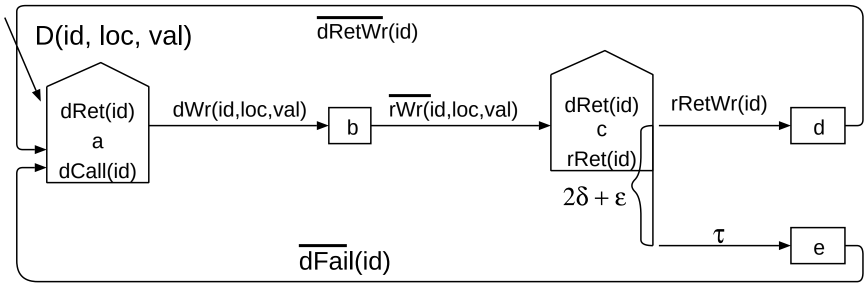
7.1 Application to the RPC Problem

The following is a timed version of R_1, where passing through the calls and returns takes not more than δ seconds:

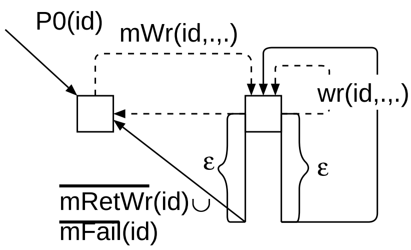


Note that actions without a timing constraint are enabled at any time. The timed version of R_2 is defined analogously (although unnecessary for the reliable memory). Call the timed RPC R^δ.

In the same way as the RPC we specify a demon which signals a failure if a call to the RPC does not return within $2\delta + \varepsilon$ seconds. The actions of the demon are the same as those of the RPC, only the prefix `r` is replaced by a `d`. Timeout is modelled by a τ-transition. The specification of the demon $D_1(z)$ is



To define a timed reliable memory, we only need to alter property $P0$ by requiring the return to occur within ε time. This is done by the following:



We call the resulting timed specification of the reliable memory M_R^ε. The timed verification problem then is

$$\left(D^{2\delta+\varepsilon} \mid R^\delta \mid M_R^\varepsilon / H \right) \backslash A[f] \;\trianglelefteq\; M_R / H$$

Note that the memory on the right hand side is the "untimed" M_R, where we interprete all actions to be enabled all the time. Further the set A

and the relabelling f have to be adjusted. This problem can once again be reduced by our method to a problem concerning transition systems of small size, as we only need to look at a prototypical z.

However, having two parameters δ and ε in the timing constraints, the standard multiplication trick is not sufficient to produce a parameterless situation. Luckily, this particular example is equivalent to a one parameter problem: computing $R^\delta \mid M_R^\varepsilon / H$ by hand one finds a transiton system, which can be regarded as parameterized in $2\delta + \varepsilon$ only. Now the previously used multiplication trick is applicable opening the problem to automatic verification by means of EPSILON.

8 Conclusion and Future Work

We have introduced a new constraint-oriented method for the (automated) verification of concurrent systems. Key concepts of our 'divide and conquer' method are *projective views*, *separation of proof obligations*, *Skolemization* and *abstraction*, which together support a drastic reduction of the complexity of the relevant subproblems. Of course, our proof methodology does neither guarantee the possibility of a finite state reduction nor a straightforward method for finding the right amount of separation or the adequate abstraction. Still, there is a large class of problems and systems, where the method can be applied quite straightforwardly. Typical examples are systems with limited data dependence. Whereas involved data dependencies may exclude any possibility of 'horizontal' decomposition, our approach elegantly extends to real time systems, even over a dense time domain. In fact, the resulting finite state problems can be automatically verified using the EPSILON verification system. All this has been illustrated using a simple example of pipelined buffers. Our experience indicates that our method scales up to practically relevant problems, as demonstrated by the problem of the transparent RPC.

Beside further case studies and the search for good heuristics for proof obligation separation and abstraction, we are investigating the limits of tool support during the construction of constraint based specifications and the application of the three reduction steps. Whereas support by graphical interfaces and interactive editors is obvious and partly implemented in META-Frame, a management system for synthesis, analysis and verification currently developed at the university of Passau, the limits of consistency checking and tool supported search for adequate separation and abstraction are still an interesting open research topic.

As pointed out, one major problem are parameters in the timing constraints. We are currently investigating methods – similar to the approach presented for parametrized timed automata in [AHV93] – for checking bisimulation and (weak) refinement for *parametrized modal transition systems.*

9 References

[ASW94] H. Andersen, C. Stirling, G. Winskel. *A Compositional Proof System for the Modal Mu-Calculus.* in: Proc. LICS 1994.

[AD94] R. Alur, D.L. Dill. *A Theory of Timed Automata.* in: Theoretical Computer Science Vol. 126, No. 2, April 1994, pp. 183-236.

[AHV93] R. Alur, T.A. Henzinger, M.Y. Vardi. *Parametric real-time reasoning.* Proc. 25th STOC, ACM Press 1993, pp. 592–601.

[BL93] M. Broy, L. Lamport. *Specification Problem.* Case study for the Dagstuhl Seminar 9439, 1994.

[Br86] R. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* in: IEEE Transactions on Computation, 35 (8). 1986.

[BCMDH90] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang. *Symbolic Model Checking:* 10^{20} *States and Beyond.* in: Proc. LICS'90.

[BS90] J. Bradfield, C.Stirling. *Local Model Checking for Finite State Spaces.* LFCS Report Series ECS-LFCS-90-115, June 1990

[CES83] E. Clarke, E.A. Emerson, A.P. Sistla. *Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications: A Practical Approach.* In Proc. 10th POPL'83

[CGL93] K. Čerāns, J.C. Godesken, K.G. Larsen. *Timed Modal Specification - Theory and Tools.* in: C. Courcoubetis (Ed.), Proc. 5th CAV, 1993. LNCS 697, Springer Berlin 1993, pp. 253–267.

[CGL92] E. Clarke, O. Grumber, D. Long. *Model Checking and Abstraction.* in: Proc. XIX POPL'92.

[CLM89] E. Clarke, D. Long, K. McMillan. *Compositional Model Checking.* in: Proc. LICS'89.

[CC77] P. Cousot, R. Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* in: Proc. POPL'77.

[EFT91] R. Enders, T. Filkorn, D. Taubner. *Generating BDDs for Symbolic Model Checking in CCS.* in: Proceedings CAV'91, LNCS 575, 1991, pp. 203–213

[EL86] E. Emerson, J. Lei. *Efficient model checking in fragments of the propositional mu-calculus.* In Proc. LICS'86, pp. 267–278.

[GW91] P. Godefroid, P. Wolper. *Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties.* in: Proc. CAV'91, LNCS 575, pp. 332–342.

[GP93] P. Godefroid, D. Pirottin. *Refining Dependencies Improves Partial-Order Verification Methods.* in: Proceedings CAV'93, LNCS 697, 1991, pp. 438–449.

[GL93] S. Graf, C. Loiseaux. *Program Verification using Compositional Abstraction.* in: Proceedings FASE/TAPSOFT'93.

[GS90] S. Graf, B. Steffen. *Using Interface Specifications for Compositional Minimization of Finite State Systems.* in: Proc. CAV'90.

[Koz83] D. Kozen. *Results on the Propositional mu-Calculus.* TCS 27, 333-354, 1983

[HL89] H. Hüttel and K. Larsen. *The use of static constructs in a modal process logic.* Proceedings of Logic at Botik'89. LNCS 363, 1989.

[Lar90] K.G. Larsen. *Modal specifications.* In: Automatic Verification Methods for Finite State Systems LNCS 407, 1990.

[LT88] K. Larsen and B. Thomsen. *A modal process logic.* In: Proceedings LICS'88, 1988.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Par81] D. Park. *Concurrency and automata on infinite sequences.* In P. Deussen (ed.), LNCS 104, pp. 167–183, 1981.

[Ste89] B. Steffen. *Characteristic Formulae.* In Proc. ICALP'89, LNCS 372, 1989

[Ste93] B. Steffen. *Generating data flow analysis algorithms from modal specifications.* in: Science of Computer Programming 21, (1993), 115 - 139.

[Val93] A. Valmari. *On-The-Fly Verification with Stubborn Sets.* in: C. Courcoubetis (Ed.), Proc. 5th CAV, 1993. LNCS 697, pp. 397–408.

[Yi91] W. Yi. *CCS + Time = an Interleaving Model for Real-Time Systems,* Proc.18th Int. Coll. on Automata, Languages and Programming (ICALP), Madrid, July 1991. LNCS 510, Springer New York 1991, pp. 217-228.