Technical Report MIP-9313, Universität Passau, Fakultät für Mathematik und Informatik, December 1993

Module Configuration by Minimal Model Construction

Bernhard Steffen* Tiziana Margaria* Burkhard Freitag†

Abstract

We present a framework for the *automatic configuration* of large systems from a library of reusable software components. Core of the framework is a *modal logic* that uniformly and elegantly captures *type descriptions*, *module specifications* and *relative time*. Whereas the first two 'dimensions' are treated similarly by means of a simple logic over a taxonomy of types and modules respectively, time is expressed by means of modalities. Besides allowing an elegant and transparent specification of module configurations, our framework also provides a *minimal model generator* that automatically generates minimal solutions to a specification problem. All this is illustrated for a practically relevant application: the automatic configuration of heterogeneous analysis systems from loose specifications.

1 Introduction and Motivation

The configuration of complex systems from reusable software components requires the search for possible implementations as well as the adaptation of these implementations within the program context according to interface constraints. Considering *interfacing modules* that, in the case of an interface conflict, serve for the right conversion, transformation or parameter configuration, it is possible to achieve a high degree of software reuse (cf. e.g. [FrMS94]). However, the approach presented there, which is used in practice [FA94], is rather restrictive: it only provides the required interfacing modules for type-incorrect program fragments. From a different point of view, it constructs a module configuration, i.e. a sequence of modules that closes the gap between two mismatching types, or equivalently, it solves an input/output specification given in terms of types. However, the real problems one wants to specify also concern *semantic properties*, perhaps expressed by some predicates restricting the kind of modules used in the solution, or even *timing constraints*, like "the declaration module should precede the using modules".

In this paper we present a framework for the *automatic configuration* of large systems from a library of reusable software components, which is based on a *modal logic* that uniformly and elegantly captures *type descriptions*, *module specifications* and *relative time*. Whereas the first two 'dimensions' are treated similarly by means of a simple logic over a taxonomy of types and modules respectively, time is expressed by means of modalities. More technically, the static aspects (type constraints) form the *atomic propositions*, the dynamic aspects (module specifications) the *actions*, and the timing aspects are modelled by means of the *modalities* of our (linear time) modal logic (cf. [Pnue77]). This allows us to give a coherent and uniform (trace) semantics of our specification language relative to a *configuration*

*Lehrstuhl für Informatik, Universität Passau, Innstr. 33, D-94032 Passau (Germany), tel: +49 851 509.345, fax: +49 851 509.346, {tiziana, steffen}@fmi.uni-passau.de

†Technische Universität München, Orléansstr. 34, D-81667 München (Germany), tel: +49 89 48095.175, fax: +49 89 48095.170, freitag@informatik.tu-muenchen.de

universe, which is the conceptual transition system-like model of the database of reusable software components: formulas are interpreted with respect to type correct module configurations, which are given as paths within the configuration universe.

Besides allowing an elegant and transparent specification of module configurations, our framework also provides a *minimal model generator* that automatically generates minimal implementations to satisfiable specifications. This generator, which like Manna and Wolper's synthesis algorithm (cf. [MaWo84]), is based on a tableau method, is modular in the following sense:

- *local constraints*, i.e. type and module constraints, are solved by database techniques. We will not go into details here, and rather consider these techniques as parameters of our approach. In fact, they can easily be exchanged, whenever a better implementation is available.
- *global constraints* give raise to refinement steps during the model construction leading to an exponential worst-case complexity, which is not surprising (cf. e.g. [SiCl85]). However, typical specifications consist of a number of well-defined subgoals, which must be met successively. These subgoals can be treated independently, resulting in an acceptable performance.

The framework is illustrated by means of a practically relevant application: the automatic configuration of heterogeneous analysis systems from loose specifications.

The paper is organized as follows: After defining the notion of configuration universe and module configuration in Section 2, we present our specification language in Section 3. Subsequently, we describe the algorithm for model construction that solves our specification problem and discuss its correctness, completeness and efficiency in Section 4. Finally, we illustrate our approach by means of a case study of practical relevance in Section 5 and draw our conclusions in Section 6.

2 Models for Configurations

First we introduce the conceptual model of the database of reusable software components.

Definition 2.1 *The* configuration universe CU *is defined as a triple* $(\mathcal{T}, \mathcal{M}, Trans)$ *where*

- $\mathcal{T}$ *is a set of types*
- $\mathcal{M}$ *is a set of software modules*
- $Trans = \{(t, m, t')\}$ *is a set of transitions where* $t, t' \in \mathcal{T}$ *and* $m \in \mathcal{M}$.

The reason for choosing this transition system-like conceptual structure is that it

- can be *locally* constructed, by simply considering the module names together with their input and output types and
- inherently embodies *global* configuration constraints: two modules can be (type correctly) connected iff the corresponding transitions are neighbours, i.e., iff the output type of the first coincides with the input type of the second.

The second feature directly motivates our notion of module configuration:

Definition 2.2 (Module configuration)

A module configuration *(or implementation) is a finite path within the configuration universe* CU, *i.e. a finite sequence of module names* $(m_1, m_2, \ldots, m_k)$, $m_i \in \mathcal{M}, 1 \leq i \leq k$, *satisfying* $\forall\, 2 \leq i \leq k.\ t'(m_{i-1}) = t(m_i)$. *The set of all module configurations is denoted by* $\mathcal{MC}$.

Thus module configurations are simply *type-correct* sequential compositions of (elementary) modules. As will be illustrated in Section 5, this simple approach already allows quite some interesting applications. It is possible to extend our approach to other module connectives, whose clean integration is a topic of current research (cf. Section 6).

3 Specification Language

As already mentioned, our specification language, which can be seen as an 'interpreted version' of Propositional Linear Temporal Logic (PLTL, see [Emer90, SiCl85]) or a linear time variant of the temporal Logic CTL (see [ClES86]), uniformly and elegantly comprises three dimensions:

- **Type or static constraints:** Here we provide a simple logic over a taxonomy of *types*, i.e. a hierarchical classification of elementary types together with some logical connectives. The type constraints form the *atomic propositions* of our temporal (modal) logic.

- **Module or dynamic constraints:** Here we provide a simple logic over a taxonomy of *modules*, which is similar to the logic of type constraints. The intuition behind this taxonomy is to express some semantic aspects of the modules. Module constraints form the *actions* of our temporal (modal) logic. In contrast to the Kripke structure used for CTL, transitions are labelled in our case, requiring action-labelled modalities in the specifications.

- **Time constraints:** These constraints, which for example specify the order in which modules can be combined, are expressed by means of the *modal structure* of our logic.

Semantically, our specification language is a modal logic interpreted over module configurations, i.e. over paths of the configuration universe. Its three-dimensional structure uniformly combines *static*, *dynamic*, and *timing* constraints to a specification language with a coherent semantics.

The following two subsections are devoted to the *local* constraints, i.e. the static and dynamic parts, and to the *global* constraints, i.e. the timing aspects.

3.1 Local Constraints: Taxonomic Expressions

Types and modules are hierarchically organised by means of a taxonomy, which, in our case, is a subset of the powerset of types (resp. modules), together with an ordering given by the usual set inclusion. Formally, the elements of a taxonomy are unary predicates that are intended to express certain semantic characterizations. Thus negation, conjunction and disjunction are well-defined, leading to the following definition:

Definition 3.1 (Taxonomy expressions)

Let TAX be a taxonomy over some set. Then we can construct the corresponding set of taxonomy expressions by:

$$TE \quad ::= \quad A \mid \neg\, TE \mid TE \,\wedge\, TE \mid TE \,\vee\, TE$$

where $A \in TAX$ are taken as atomic propositions.

Now the language for static and dynamic constraints is simply given by means of the corresponding set of taxonomic expressions, which are assumed here to contain the singleton sets $\{m\}$ for every module $m \in \mathcal{M}$ as atomic predicates[1]:

[1] This is necessary to allow users to specify their configurations as precisely as possible.

- Static or type constraints are given by taxonomic expressions t_c over the set of types. We will denote this set of *static taxonomy expressions* by STE.

  In this paper we do not consider type construction explicitely: even constructed types are atomic on this level. Handling type constructors is an orthogonal but straightforward extension.

- Dynamic or module constraints are given by taxonomic expressions m_c over the set of modules. We will denote this set of *dynamic taxonomy expressions* by DTE.

  Also in this case we do not consider substructure, since this paper concentrates on the configuration of complex systems according to *global* constraints.

The semantics of these expressions is straightforward, and simply considers taxonomy expressions as derived predicates: a type (or module) satisfies a type (or module) constraint if it satisfies the corresponding predicate.

Remark: The reason for this choice of local constraints is the following:

Normally, it is not desirable to consider every module of the base set $\mathcal{M}$ that satisfies the corresponding type constraints (i.e. a static taxonomy expression) as a possible next candidate that can extend an already established partial configuration. In practice, often much more is known about the candidate modules, like e.g. problem class, field of application or even the accurate module name. The dynamic taxonomy expressions of our framework allow us to formally express this knowledge in a way accessible to the minimal model generator. Thus it is automatically taken care of during the module configuration process.

Currently, the search for appropriate modules is realized by means of the OTTER theorem prover [McCu90]. Moreover, an implementation of module selection from a database using methods of (deductive) database systems [FSSBG93, FSS91] is under way.

3.2 Global Constraints: The Temporal Aspect

Using a modal logic as a frame, we are able to combine the two local kinds of constraints to obtain an elegant and uniform specification language for global constraints. The syntax of this language (SLTL, for Semantic Linear Time Logic) is defined as follows:

Definition 3.2 (SLTL)

The syntactic structure of SLTL is given by:

$$\Phi \ ::= \ tt \mid t_c \mid \neg\Phi \mid \Phi \wedge \Phi \mid \ < m_c > \Phi \mid \mathbf{G}\ \Phi \mid \Phi\ \mathbf{U}\ \Psi$$

where $t_c \in STE$ and $m_c \in DTE$ represent type constraints and module constraints, respectively.

Modal logic formulas Φ are interpreted over the set of module configurations, i.e. over paths in the configuration universe, leading to a **linear time logic** and therefore to the semantics given in Figure 1.

Intuitively, tt is satisfied by every module configuration and t_c by every module configuration whose first component has an input interface satisfying t_c. Negation and disjunction are interpreted in the usual fashion. The construct $< . >$ is a *modal operator* corresponding to the (parameterized) *nexttime* operator of Linear Time Logic. Accordingly, $< m_c > \Phi$ is satisfied by a module configuration, if its first component satisfies m_c , and the *continuation* of the module configuration[3] satisfies Φ. Further,

[3]I.e., the module configuration where the first component is skipped.

$$
\begin{array}{rcl}
[\![tt]\!] & = & \mathcal{MC} \\
[\![t_c]\!] & = & \{\, (m_1, \cdots, m_k) \in \mathcal{MC}^2 \mid \text{The input interface of } m_1 \text{ satisfies } t_c \,\} \\
[\![\neg\Phi]\!] & = & \{\, p \in \mathcal{MC} \mid p \notin [\![\Phi]\!] \,\} \\
[\![\Phi_1 \wedge \Phi_2]\!] & = & [\![\Phi_1]\!] \cap [\![\Phi_2]\!] \\
[\![<m_c>\Phi]\!] & = & \{\, (m_1, m_2, \ldots, m_k) \in \mathcal{MC} \mid m_1 \in [\![m_c]\!] \wedge (m_2, \ldots, m_k) \in [\![\Phi]\!] \,\} \\
[\![\mathbf{G}\Phi]\!] & = & \{\, (m_1, \ldots, m_k) \in \mathcal{MC} \mid \forall j \leq k.\ (m_j, \cdots, m_k) \in [\![\Phi]\!] \,\} \\
[\![\Phi\mathbf{U}\Psi]\!] & = & \{\, (m_1, \ldots, m_k) \in \mathcal{MC} \mid \exists i \leq k.\ (m_i, \cdots, m_k) \in [\![\Psi]\!] \wedge\ (\forall j < i.(\, m_j, \cdots, m_k) \in [\![\Phi]\!]) \,\}
\end{array}
$$

Figure 1: The semantics of SLTL formulas.

$\mathbf{G}\Phi$ requires that Φ is satisfied generally, i.e. for every postfix[4] of the module configuration.[5] Finally, $\Phi\mathbf{U}\Psi$ (strong until) expresses that the property Φ holds for all modules of the module configuration, until we reach a position where the corresponding continuation satisfies the property Ψ. Note that $\Phi\mathbf{U}\Psi$ guarantees that the property Ψ holds eventually!

Convention: Even though module configurations are written as sequences of module names, they are paths in the configuration universe, i.e. they consist of alternating sequences of type and module names. As the types are implied by the module names, we can stick to our convention for the ease of representation. It should be kept in mind, however, that the reference points for the interpretation of a configuration are the nodes corresponding to the input type of the first module.

It is convenient to define the following derived operators:

$$
\begin{array}{lccl}
\textit{Disjunction}: & \Phi \vee \Psi & =_{df} & \neg\,(\neg\Phi \wedge \neg\Psi) \\
\textit{Eventually}: & \mathbf{F}\ \Phi & =_{df} & tt\ \mathbf{U}\ \Phi
\end{array}
$$

These two operators are standard. The following operators are particularly suitable when configuring modules:

$$
\begin{array}{lcl}
\Phi\ \textbf{before}\ \Psi & =_{df} & (\neg\Psi)\ \mathbf{U}\ (\Phi \wedge \neg\Psi)\ \wedge\ \mathbf{F}\ \Psi \\
\Phi\ \textbf{nlt}\ \Psi & =_{df} & (\neg\Psi)\ \mathbf{U}\ \Phi\ \wedge\ \mathbf{F}\ \Psi \\
\Phi\ \textbf{then}\ \Psi & =_{df} & \mathbf{F}\ (\Phi\ \wedge\ \mathbf{F}\ \Psi) \\
\Psi\ \textbf{after}\ \Phi & =_{df} & \mathbf{F}\ (\Phi \wedge\ <.>\ \mathbf{F}\Psi)
\end{array}
$$

Intuitively, Φ **before** Ψ and Ψ **after** Φ guarantee that Ψ will become valid strictly after Φ has been satisfied, with the distinction that Φ **before** Ψ additionally forbids Ψ to hold before Φ is satisfied. In contrast, Φ **nlt** Ψ (*not later than* operator) and Φ **then** Ψ do not require this 'strict' sequentialization. They are also satisfied, if $\Phi \wedge \Psi$ hold eventually (with a similar distinction as above).

Minimality Requirement: A system for automatically generating solutions to configuration problems should provide the simplest solutions, which may then successively be refined in order to satisfy additional constraints. As we will see in Section 5, this can easily and modularly be achieved within our framework. This observation leads to the following requirement:

[4]'Postfix' is a straightforward generalization of continuation.

[5]Note that, as we are dealing with finite paths only, certain properties of this kind will never hold. We will mainly use this operator to express global type constraints.

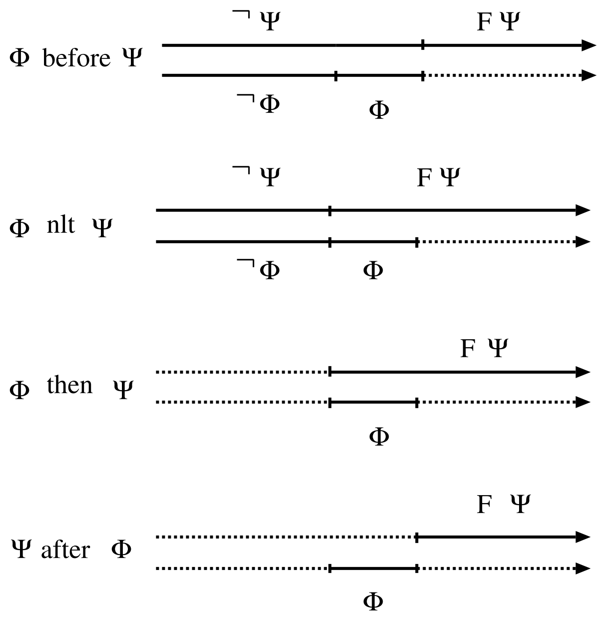

Figure 2: Illustration of the derived operators.

> Admissible *module configurations are additionally required to be* minimal*: no shorter module configuration satisfies the specification.*[6]

We close this section with some simple examples. A detailed treatment of a complex example is given in Section 5.

Examples

1. The logic SLTL includes as a special case the whole language for local constraints. This is clear for the type constraints. Modules (i.e., module configurations of length one) satisfying a taxonomy expression $m_c \in DTE$ can be explicitely specified by $< m_c > tt$ in SLTL. Note that the minimality requirement enters here. Without this requirement, each extension of these 'single module solutions' would be a perfect solution.

   Of course this principle can be easily extended to specify explicit sequences of modules by[7]:

   $$\Phi_{seq} \quad = \quad < m_1 >< m_2 > \ldots < m_i > tt$$

   The trace $(m_1, m_2, \ldots, m_i)$ of the configuration universe CU is the unique solution to that problem.

2. Looser specifications, i.e., specifications that abstract from the actual number and name of the modules in the desired configuration, are expressed through more abstract properties of the traces. As an example, the following property

   A module declaration must precede the module's use

[6]Module configurations are paths of the configuration universe. Thus their length is defined.
[7]Remember, we assume the set of taxonomy expressions to contain the singleton sets as atomic predicates.

can easily be expressed by: *decl(m)* **before** *use(m)*.

Similarly, one can formulate runtime properties like

Produce an output before termination.

4 Module Configuration by Minimal Model Construction

Our algorithm is based on four functions, which embody the semantics of our logic and guide the search for an initial model. Given a module configuration specification Φ, these functions are defined as follows:

Computation of Admissible Initial Modules:

$$adm(\Phi) \;=\; \begin{cases} \{\, m \mid (m) \in [\![\Phi]\!] \,\} & \text{if } \Phi \in STE \\ \{\, m \mid (m) \notin adm(\Phi_1) \,\} & \text{if } \Phi = \neg\Phi_1 \\ adm(\Phi_1) \;\cap\; adm(\Phi_2) & \text{if } \Phi = \Phi_1 \wedge \Phi_2 \\ [\![m_c]\!] & \text{if } \Phi = < m_c > \Phi_1 \\ adm(\Phi_1) & \text{if } \Phi = \mathbf{G}\,\Phi_1 \\ adm(\Phi_1) \cup adm(\Psi) & \text{if } \Phi = \Phi_1\,\mathbf{U}\,\Psi \end{cases}$$

Whereas the intuitive meaning of *adm* is clear, the idea behind *Split* is to realize the case distinction inherent in the until operator: in essence, the formula $\Phi_1\ \mathbf{U}\ \Psi$ is decomposed into the two conjuncts Ψ and $\Phi_1 \wedge < adm(\Phi_1) > \Phi$, which express the two situations we must distinguish: either Ψ is valid immediately, or the 'recursion' must be unfolded leading to the second conjunct.[8] Thus the set $Split(\Phi)$ yields the full case distinction for an exhaustive search.

The Split Operator:

$$Split(\Phi) \;=\; \begin{cases} \{\, \neg\Phi' \mid \Phi' \in Split(\Phi_1) \,\} & \text{if } \Phi = \neg\Phi_1 \\ \{\, \Phi' \wedge \Phi'' \mid \Phi' \in Split(\Phi_1) \text{ and } \Phi'' \in Split(\Phi_2) \,\} & \text{if } \Phi = \Phi_1 \wedge \Phi_2 \\ \{\, \mathbf{G}\Phi' \mid \Phi' \in Split(\Phi_1) \,\} & \text{if } \Phi = \mathbf{G}\,\Phi_1 \\ Split(\Psi) \cup \{\, \Phi_1 \wedge < adm(\Phi_1) > \Phi \,\} & \text{if } \Phi = \Phi_1\,\mathbf{U}\,\Psi \\ \Phi & \text{otherwise} \end{cases}$$

where $adm(\Phi)$ is used as an abbreviation for the dynamic taxonomy expression $\bigvee \{\, m \mid m \in adm(\Phi) \,\}$.

The application of *Split* eliminates all *unguarded* occurrences[9] of the until operator. This observation simplifies the understanding of the definition of *Red*, which reduces a formula according to a transition step: Φ is valid for a module configuration $(m_1, \cdots, m_k)$, if and only if the next step is consistent with Φ, i.e. $m_1 \in adm(\Phi)$, and $Red(\Phi)$ is satisfied by $(m_2, \cdots, m_k)$. Note that the 'until case' in this definition is unnecessary for our application.

[8] In fact, one could strengthen the second formula to $\neg\Psi \wedge \Phi_1 \wedge < adm(\Phi_1) > \Phi$ in order to make the cases disjoint. However, this strengthening does not seem to improve the situation in practice.

[9] An occurrence is *guarded*, if it appears within the range of a modality operator.

The Reduction Operator:

$$Red(\Phi) \quad = \quad \begin{cases} tt & \text{if } \Phi \in \{tt, t_c\} \\ \neg Red(\Phi_1) & \text{if } \Phi = \neg\Phi_1 \\ Red(\Phi_1) \wedge Red(\Phi_2) & \text{if } \Phi = \Phi_1 \wedge \Phi_2 \\ \Phi_1 & \text{if } \Phi = < m_c > \Phi_1 \\ \Phi \wedge Red(\Phi_1) & \text{if } \Phi = \mathbf{G}\ \Phi_1 \\ \Phi \wedge Red(\Phi_1) & \text{if } \Phi = \Phi_1\ \mathbf{U}\ \Psi \end{cases}$$

The last operator is important for the termination of our algorithm. Its underlying intuition will be explained in the context where the operator is used.

The G-Eliminator:

$$\mathbf{G}_{elim}(\Phi) \quad = \quad \begin{cases} \neg\mathbf{G}_{elim}(\Phi_1) & \text{if } \Phi = \neg\Phi_1 \\ \mathbf{G}_{elim}(\Phi_1) \wedge \mathbf{G}_{elim}(\Phi_2) & \text{if } \Phi = \Phi_1 \wedge \Phi_2 \\ \mathbf{G}_{elim}(\Phi_1) & \text{if } \Phi = \mathbf{G}\ \Phi_1 \\ \Phi & \text{otherwise} \end{cases}$$

4.1 The Algorithm

In order to construct a minimal model for a module configuration specification Φ, our algorithm sucessively treats subproblems of the form (t, Φ'), where t is a type of $\mathcal{T}$ and Φ' is a module configuration specification. In the elementary version of the algorithm, where no optimizations are considered, this global treatment is organized as follows:

1. Determine all types t of $\mathcal{T}$ satisfying

   $$adm(t) \cap adm(\Phi) \neq \emptyset$$

   and construct the set of first level *configurations* $(PS, (t, \Phi))$ consisting of a subproblem or *continuation constraint* (t, Φ) and its corresponding *partial solution* PS, i.e. a minimal path of the configuration universe, which reduces the original problem to the considered subproblem[10]. Obviously, initially this partial solution is just the empty module configuration.

2. Intuitively, the *termination* condition is given by situations, where the type of the current subproblem implies the formula that remains to be verified. Using finite paths as models for the formulas, the role of $\mathbf{G}\Phi$ is not immediate. A moment of thought, however, establishes that checking that a type implies $\mathbf{G}_{elim}(\Phi)$ rather than Φ itself just provides the required termination condition in our context. Consequently, if one of the subproblems, e.g. (t', Φ'), of the currently investigated level is trivial, i.e. the atomic proposition t' implies $\mathbf{G}_{elim}(\Phi')$, the corresponding partial solution is returned and the algorithm **terminates**. If this happens already at the first level, t' is returned as the trivial solution to the problem.

   Otherwise, the step function NEXTSTEP defined below is applied to each of the current-level configurations, and the resulting sets of next-level configurations are collected. This results in a (global) set of next-level configurations (PS, CC), where

[10] If there are several such paths, we simply choose non-deterministically.

- PS is a path of the configuration universe, whose length coincides with the level of the problems currently being investigated, and
- CC is the corresponding 'next-level' subproblem, which must be investigated further.

3. If the set of 'next-level' problems is empty, the algorithm **terminates** with the answer that the specification is not satisfiable in the configuration universe.

   Otherwise, the algorithm proceeds by treating the new level according to the second step.

The procedure NEXTSTEP $(PS, (t, \Phi))$ realizes the following three substeps:

1. If $adm(t) \cap adm(\Phi) = \emptyset$, the considered subproblem is not satisfiable. Thus NEXTSTEP (t, Φ) **terminates** with the empty set.

   Otherwise, $Split(\Phi)$ is computed, leading to a new set of subproblems

$$\{ (t, \Phi') \mid \Phi' \in Split(\Phi) \}$$

2. For each of these new subproblems (t, Φ') and each transition (t, m, t') with $m \in adm(t) \cap adm(\Phi')$, we construct the corresponding next step subproblem $(t', Red(\Phi'))$. If this subproblem is new, i.e. it did not appear before, the configuration $(PS', (t', Red(\Phi'))$ is collected, where PS' is the partial solution PS extended by the module m.

3. NEXTSTEP (t, Φ) **terminates** with a set of new next-level configurations.

A stepwise execution of the algorithm is presented in Section 5 on a practically relevant case study.

4.2 Correctness, Completeness and Efficiency

It is easy to see that the number of formulas generated from Φ by applications of *Red* and *Split* is finite. Thus the finiteness of the configuration universe implies the finiteness of the number of generated subproblems and the termination of the local computations like e.g. for *adm*, *Red* and *Split*, which guarantees the *termination* of the complete program. Moreover, considering the stepwise construction of the subproblems, which describe the remaining constraints for their corresponding partial solutions, yields the *correctness* of the algorithm. The *completeness* of the algorithm is a consequence of the exhaustiveness of the search as organized by our 'tableau'-like approach. Summarising, we therefore have:

Theorem 4.1 (Correctness and Completeness)

The algorithm always terminates and, in case of a satisfiable specification, it delivers a corresponding minimal module configuration.

Global constraints give raise to refinement steps during the model construction leading to an exponential worst-case complexity, which is not surprising for a global search problem (cf. e.g. [SiCl85]). However, typical specifications consist of a number of well-defined subgoals which must be met successively. These subgoals can be treated independently, leading to an acceptable performance even for the basic version of our algorithm. Moreover, the algorithm leaves plenty of room for optimizations, which will be the topic of a future paper.

5 Case Study: Configuration of Heterogeneous Analysis Systems

The idea that different verification tools should be combined to a single tool in order to allow a flexible verification dates back to the Edinburgh Concurrency Workbench ([ClPS89, ClPS93]) which included a model checker and various equivalence and preorder checkers. Practical experience showed that this flexibility is important in order to deal with 'real-life' verification problems. Thus in the meantime almost all modern tools and systems, such as e.g. Aldébaran [Fern88], AUTO [BoSV88, RoSi91], and CESAR [RRSV87], provide several methods for verification. However, the combination of the various methods is not supported there.

This is the motivation for our environment for the development of special purpose heterogeneous analysis and verification tools, which constitutes a framework for the development of application specific heterogeneous tools and provides facilities for the automation of the synthesis process along the lines described in the previous Section. We will therefore illustrate our minimal model generator within the fragment of this setting which is described in Section 5.1. In order to demonstrate the flexibilty of our approach, we will

- first, generate a minimal module configuration for an intuitive problem, which can be specified essentially in terms of an type-inconsistent program, in Section 5.2
- second, strengthen the requirement for the solution by adding a global type constraint, which in this case will lead to an unsatisfiable specification, in Section 5.3, and
- third, eliminate some overspecification of the initial specification, which finally yields the desired implementation, in Section 5.4.

5.1 The Configuration Setup

Let us consider the fragment of the database of available components shown in Tab. 1 and taxonomically classified in Figure 3. It includes analysis and transformation components (*part-s-bisim*, *part-w-bisim*, *collapse*, *char-form*), semantic interpretation functions (*op-sem*), modules for minimization by rewriting (*CCS-rewr*), and some primitives to support a graphic display of the results (*mkgraph*, *showgraph*). Relevant basic types are: graph types *Graph* and *S-Graph* [Hims93], logic formulas *Form*, *CCS* [Miln89] expressions, transition systems *TS*, and partitions *Part* on the state space of transition systems.

5.2 The Initial Specification

Our discussion is based on the following synthesis problem:

Minimize a CCS term ccs up to bisimulation and display the result graphically.

A user of our system may try the following straightforward, but type inconsistent and semantically incomplete formalization of the verbal specification:

$$showgraph(\mathsf{minimize}(\mathsf{bisim}(ccs))) \qquad (*)$$

We will now illustrate the synthesis algorithm of our system by showing how it automatically transforms this intuitive specification 'skeleton' into a complete implementation by means of the modules listed in the Table 1. This part of our system is realized on top of the OTTER theorem prover, which serves as deduction engine for constraint solving and for search in the database of elementary modules.

Library fragment			
module name	input type	output type	function description
part-s-bisim	*TS*	*Part*	partitions into equiv. classes wrt. strong bisim.
part-w-bisim	*TS*	*Part*	partitions into equiv. classes wrt. weak bisim
collapse	*Part*	*TS*	factorizes wrt. a partition
char-form	*TS*	*Form*	conv. into a characteristic formula
op-sem	*CCS*	*TS*	constr. the model of a CCS term
CCS-rewr	*CCS*	*CCS*	reduces a CCS term wrt. bisim.
mkgraph	*TS*	*Graph*	converts the model into a graph
showgraph	*Graph*	*S-Graph*	displays a graph

Table 1: Fragment of the library of components

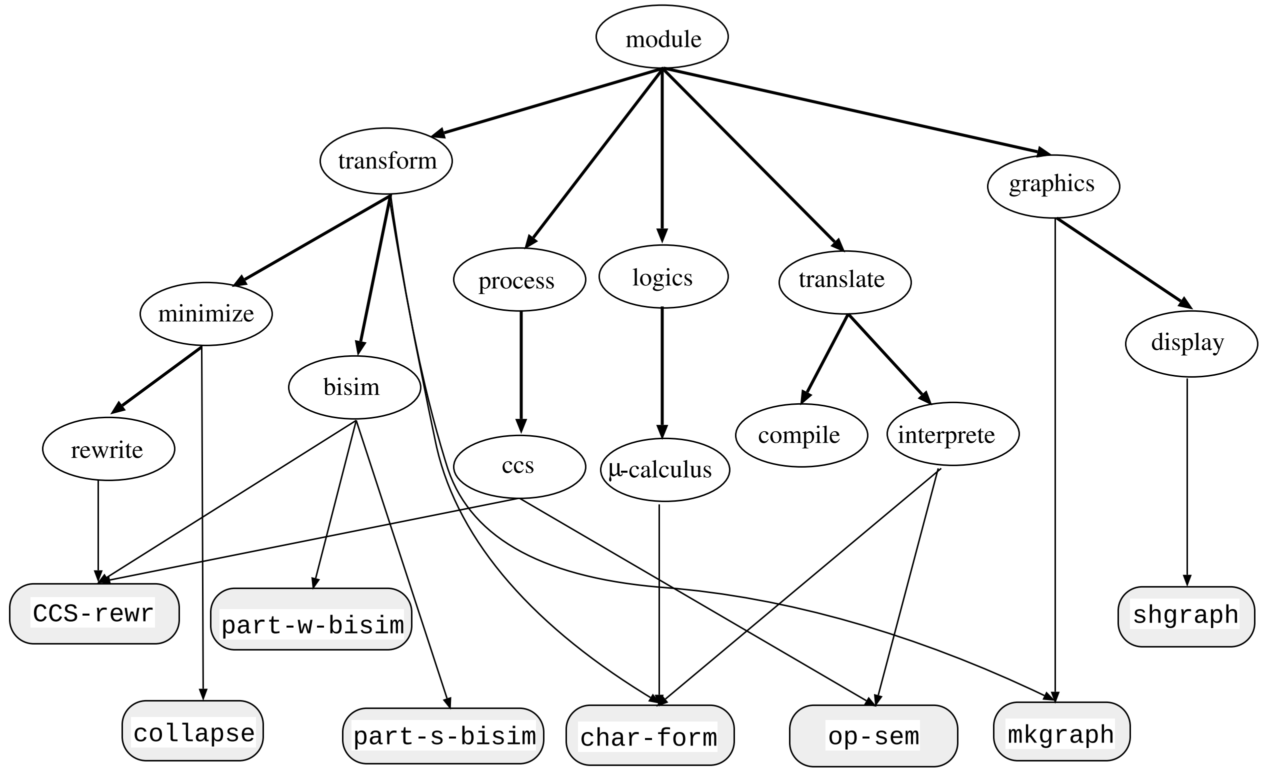

Figure 3: Taxonomy over the library fragment

The 'informal' specification above corresponds to the modal formula:

$$< \mathsf{bisim} > \ \textbf{before} \ < \mathsf{minimize} > \ \wedge \ \mathit{Graph}\ \textbf{after} \ < \mathsf{minimize} >$$

together with the local type constraint, requiring that the synthesis must start with type *ccs*. Using macro expansion, this specification can automatically be translated into the continuation constraint consisting of the type CCS and the formula:

$$\Phi^0 \ = < \neg\mathsf{minimize} > \mathbf{U} < \mathsf{bisim} \wedge \neg\mathsf{minimize} > \ \wedge \ tt\,\mathbf{U}(< \mathsf{minimize} > \wedge < . > (tt\,\mathbf{U}\,\mathit{Graph})\,)$$

Note that this transformation realizes a simple optimization which is not mentioned in the description of our algorithm: final requirements consisting of a module application are transformed in a type requirement, namely the requirement of reaching the corresponding input type. This shortens the search path, and the final solution can simply be obtained by adding the module name to the automatically generated solution. In this case the module *showgraph* must be added to the synthesized solution.

Figure 4 summarizes the synthesis process relative to the definitions of formulas given in Figure 5. Most of the synthesis graph is self-explanatory, except for

- the 'flashes', indicating the local inconsistency of a formula,
- the ε arrows, indicating that the set of corresponding admissible modules is empty,
- the huge 'hook' at the bottom, indicating *success*, i.e. all requirements of the specifications are met, and
- the huge curley braces 'cutting' unsuccessful branches. In these cases, the branches either lead to failure, i.e. to an unsatisfiable subproblem, or they generate subproblems which already appeared on an earlier level.

Summarizing, the synthesis process therefore successfully terminates and delivers the following implementation, where we non-deterministically chose *part-s-bisim* rather than *part-w-bisim*, which also reflects the default choice in our environment:

$$\mathit{showgraph} \circ \mathit{mkgraph} \circ \mathit{collapse} \circ \mathit{part\text{-}s\text{-}bisim} \circ \mathit{op\text{-}sem}$$

5.3 Adding a Global Type Constraint

From the pratical point of view, the synthesized solution has one major flaw: it generates the full transition system for the CCS term *ccs* before the minimization is applied. This restricts the applicability of the synthesized solution because of the *state explosion* problem: the transition system for a parallel combination of processes is exponential in the number of parallel components. In order to aviod such "bad" solutions, we can simply add a global type constraint to the specification, which results in the continuation constraint:

$$< \mathsf{bisim} > \ \mathbf{before} \ < \mathsf{minimize} > \ \wedge \ \mathit{Graph}\ \mathbf{after} \ < \mathsf{minimize} > \ \wedge \neg \mathit{TS}\ \mathbf{U} \ < \mathsf{minimize} >$$

or equivalently:

$$\Psi^0 \ = \ \Phi^0 \ \wedge \ (\neg \mathit{TS}\, \mathbf{U} \ < \mathsf{minimize} >)$$

together with the local type constraint, requiring that the synthesis must start with type *ccs*. The new conjunct, $\neg \mathit{TS}\ \mathbf{U}\, \mathsf{minimize}$, intuitively expresses that no transition system should be generated before a minimization is applied. To our knoweledge, there does not exist another framework for the synthesis of module configurations that allows global constraints of this kind. Even our own approach presented in [FrMS94], which is unique in capturing the initial specification, fails to express global type constraints.

Applying our algorithm to the new specification leads to an unsuccessful result (see Figures 6,7). Thus there does not exist a module configuration satisfying all our desires. As we will see in the next section, this is a consequence of an overspecification in the initial specification Φ^0.

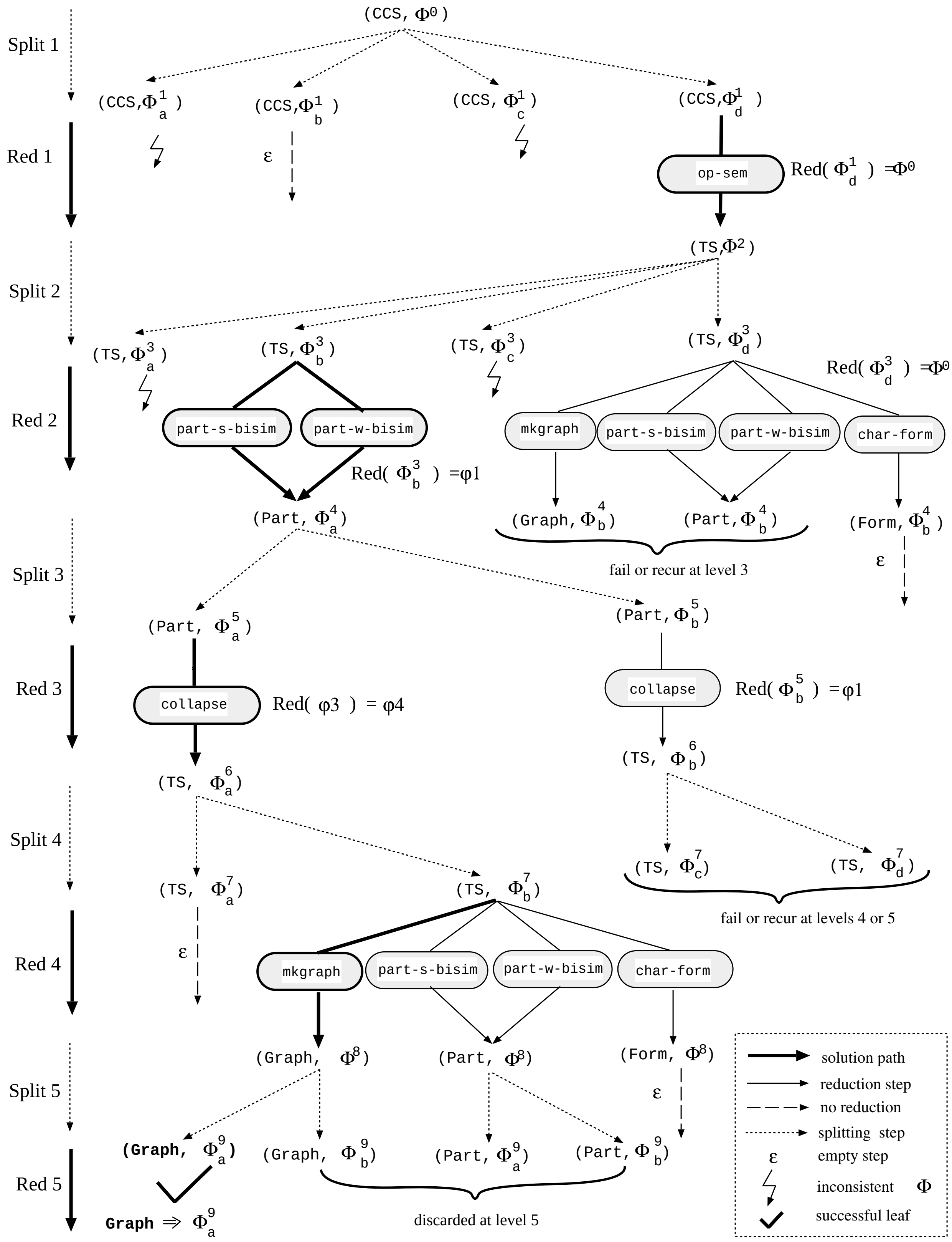

Figure 4: Solving the initial specification

$$
\begin{array}{lcl}
\Phi^0 & = & \overbrace{<\neg\mathsf{minimize}>\mathbf{U}\underbrace{<\mathsf{bisim}\wedge\neg\mathsf{minimize}>}_{\varphi_2}}^{\varphi_0}\ \wedge\ \overbrace{tt\,\mathbf{U}\underbrace{(<\mathsf{minimize}>\wedge<.>\underbrace{(tt\,\mathbf{U}\,Graph)}_{\varphi_4})}_{\varphi_3}}^{\varphi_1} \\
\\
\Phi^1_a & = & <\mathsf{bisim}\wedge\underline{\neg\mathsf{minimize}}>\ \wedge\ <\underline{\mathsf{minimize}}>\wedge<.>(tt\,\mathbf{U}\,Graph) \\
 & = & \varphi_2\ \wedge\ \varphi_3 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{inconsistent} \\
\Phi^1_b & = & <\mathsf{bisim}\wedge\neg\mathsf{minimize}>\ \wedge(tt\wedge<adm(tt)>tt\,\mathbf{U}(<\mathsf{minimize}>\wedge<.>(tt\,\mathbf{U}\,Graph)) \\
 & = & \varphi_2\wedge\ <.>\varphi_1 \\
\Phi^1_c & = & <\underline{\neg\mathsf{minimize}}>\wedge<adm(\neg\mathsf{minimize})><\neg\mathsf{minimize}>\mathbf{U}<\mathsf{bisim}\wedge\neg\mathsf{minimize}>\ \wedge \\
 & & <\underline{\mathsf{minimize}}>\wedge<.>(tt\,\mathbf{U}\,Graph) \\
 & = & <\neg\mathsf{minimize}>\wedge<adm(\neg\mathsf{minimize})>\varphi_0\ \wedge\ \varphi_3 \qquad\qquad\qquad\qquad\qquad\qquad \mathit{inconsistent} \\
\Phi^1_d & = & <\neg\mathsf{minimize}>\wedge<adm(\neg\mathsf{minimize})><\neg\mathsf{minimize}>\mathbf{U}<\mathsf{bisim}\wedge\neg\mathsf{minimize}>\ \wedge \\
 & & (tt\wedge<adm(tt)>tt\,\mathbf{U}(<\mathsf{minimize}>\wedge<.>(tt\,\mathbf{U}\,Graph)) \\
 & = & <\neg\mathsf{minimize}>\wedge<adm(\neg\mathsf{minimize})>\varphi_0\wedge<.>\varphi_1 \\
\\
\Phi^2 & = & \Phi^0 \\
\\
\Phi^3_i & = & \Phi^1_i, \qquad \mathit{for}\ i\ =\ a,\ldots,d \\
\\
\Phi^4_a & = & \varphi_1 \\
\Phi^4_b & = & \Phi_2\ =\ \Phi_0 \\
\\
\Phi^5_a & = & \varphi_3 \\
\Phi^5_b & = & tt\wedge<adm(tt)>\varphi_1 \\
\\
\Phi^6_a & = & \varphi_4 \\
\Phi^6_b & = & \Phi^5_a\ =\ \varphi_3 \\
\\
\Phi^7_a & = & Graph \\
\Phi^7_b & = & tt\wedge<adm(tt)>\varphi_4 \\
\Phi^7_c & = & \Phi^6_b\ =\ \varphi_3 \\
\Phi^7_d & = & tt\wedge<adm(tt)>\varphi_1\ =\ \Phi^5_b \\
\\
\Phi^8 & = & \Phi^6_a\ =\ \varphi_4 \\
\\
\Phi^9_a & = & \Phi^7_a\ =\ Graph \\
\Phi^9_b & = & tt\wedge<adm(tt)>\varphi_4\ =\ \Phi^7_b
\end{array}
$$

Figure 5: Continuation formulas for the initial specification

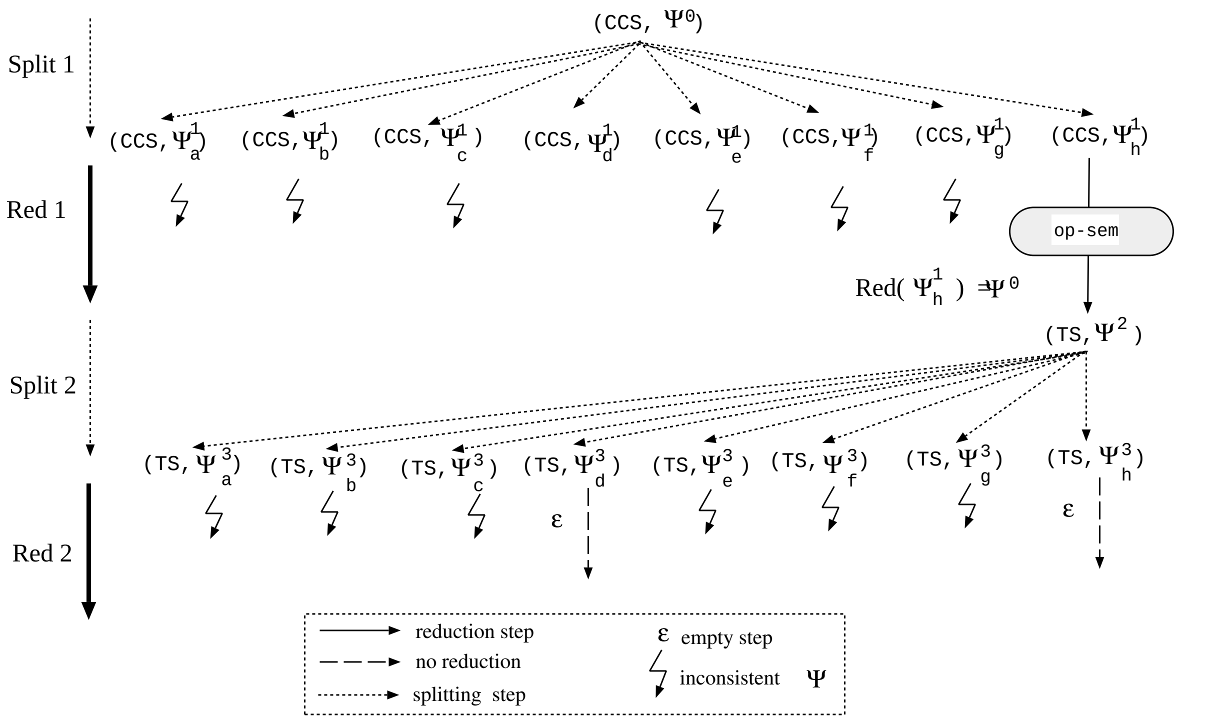

Figure 6: Attempt with the additional global type constraint

$$
\begin{array}{lcll}
\Psi^0 & = & \Phi^0 \wedge (\neg TS\ \mathbf{U}\ < \mathsf{minimize} >) & \\
\\
\Psi^1_{a,b} & = & \underline{\Phi^1_a}\ \wedge\ \mathit{Split}(\neg TS\ \mathbf{U}\ < \mathsf{minimize} >) & \mathit{inconsistent} \\
\Psi^1_c & = & \underline{\Phi^1_b} \wedge\ \underline{< \mathsf{minimize} >} & \mathit{inconsistent} \\
\Psi^1_d & = & \Phi^1_b \wedge\ \neg TS \wedge\ < adm(\neg TS) > (\neg TS\ \mathbf{U}\ < \mathsf{minimize} >) & \\
\Psi^1_{e,f} & = & \underline{\Phi^1_c}\ \wedge\ \mathit{Split}(\neg TS\ \mathbf{U}\ < \mathsf{minimize} >) & \mathit{inconsistent} \\
\Psi^1_g & = & \underline{\Phi^1_d} \wedge\ \underline{< \mathsf{minimize} >} & \mathit{inconsistent} \\
\Psi^1_h & = & \Phi^1_d \wedge\ \neg TS \wedge\ < adm(\neg TS) > (\neg TS\ \mathbf{U}\ < \mathsf{minimize} >) & \\
\\
\Psi^2 & = & \Psi^0 & \\
\\
\Psi^3_i & = & \Psi^1_i, \qquad \mathit{for}\ i\ =\ a, \ldots, h &
\end{array}
$$

Figure 7: Continuation formulas for the second attempt

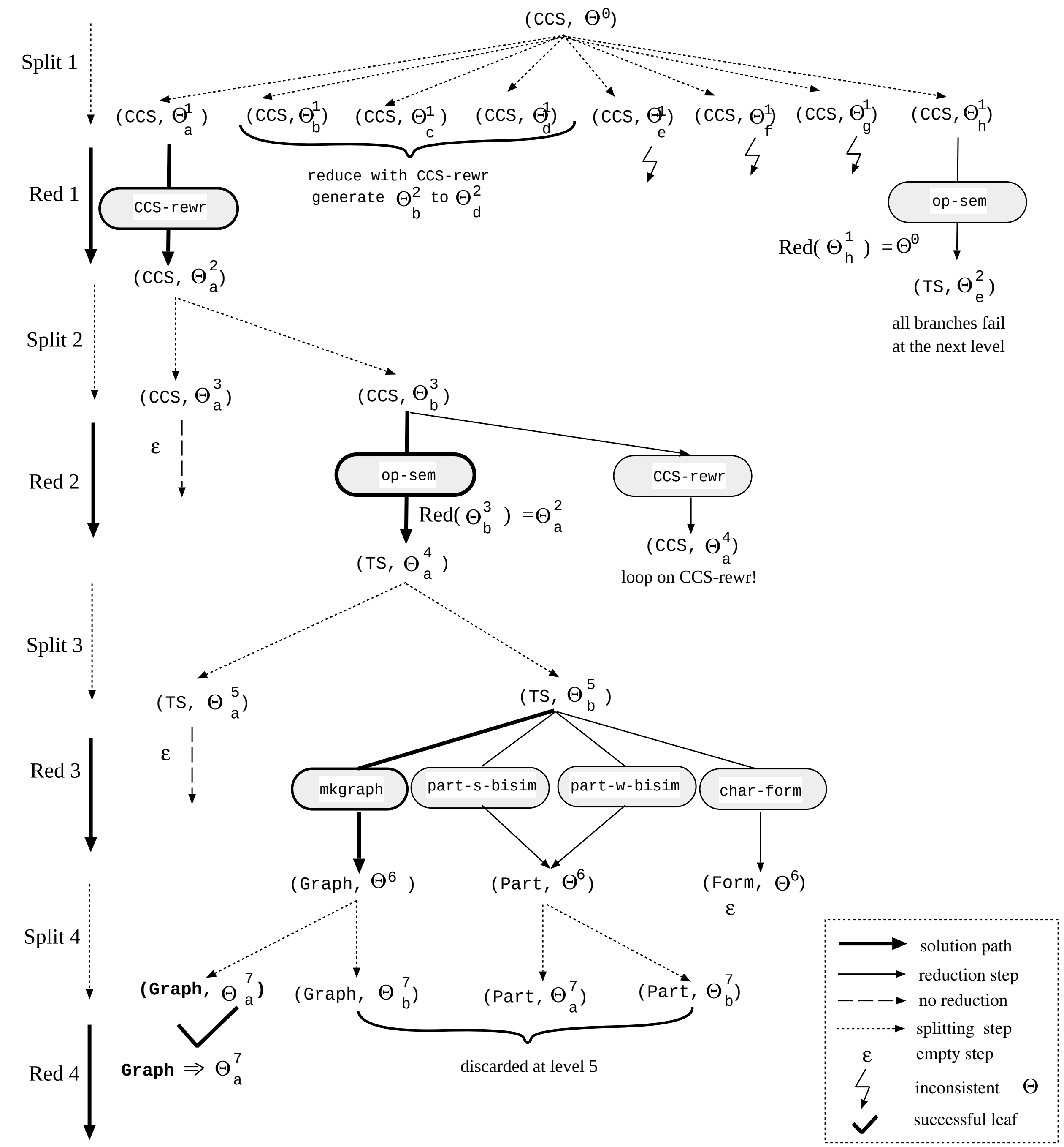

Figure 8: Solving the final specification

5.4 Eliminating the Overspecification

The harmful overspecification is a consequence of the format of the initial specification (∗), which requires a strict sequentialization of bisim and minimize. This strictness can be easily eliminated in SLTL leading to the following formula:

$$\Theta^0 \;=\; <\mathsf{bisim}>\ \mathbf{nlt}\ <\mathsf{minimize}>\ \wedge\ <\mathsf{minimize}>\ \mathbf{then}\ \mathit{Graph}\ \wedge\ \neg \mathit{TS}\ \mathbf{U}\ <\mathsf{minimize}>$$

$$
\begin{aligned}
\Theta^0 &= \overbrace{<\neg\mathsf{minimize}> \mathbf{U} \underbrace{<\mathsf{bisim}>}_{\theta_2}}^{\theta_0} \wedge \overbrace{tt\ \mathbf{U}\, \overbrace{(<\mathsf{minimize}> \wedge <.> \underbrace{(tt\ \mathbf{U}\, Graph)}_{\theta_4})}^{\theta_3}}^{\theta_1} \wedge \\
&\quad \overbrace{\neg TS\ U <\mathsf{minimize}>}^{\theta_5}
\end{aligned}
$$

$$
\begin{aligned}
\Theta^1_a &= <\mathsf{bisim}> \wedge \theta_3 \wedge <\mathsf{minimize}> \\
\Theta^1_b &= <\mathsf{bisim}> \wedge \theta_3 \wedge \neg TS \wedge <adm(\neg TS)> \theta_5 \\
\Theta^1_c &= <\mathsf{bisim}> \wedge (tt \wedge <adm(tt)> \theta_1) \wedge <\mathsf{minimize}> \\
\Theta^1_d &= <\mathsf{bisim}> \wedge (tt \wedge <adm(tt)> \theta_1) \wedge \neg TS \wedge <adm(\neg TS)> \theta_5 \\
\Theta^1_e &= <\underline{\neg\mathsf{minimize}}> \wedge <adm(\neg\mathsf{minimize})> \theta_0 \wedge \\
&\quad <\underline{\mathsf{minimize}}> \wedge <.> \theta_4 \wedge <\underline{\mathsf{minimize}}> \qquad \mathit{inconsistent} \\
\Theta^1_f &= <\underline{\neg\mathsf{minimize}}> \wedge <adm(\neg\mathsf{minimize})> \theta_0 \wedge \\
&\quad <\underline{\mathsf{minimize}}> \wedge <.> \theta_4 \wedge \neg TS \wedge <adm(\neg TS)> \theta_5 \qquad \mathit{inconsistent} \\
\Theta^1_g &= <\underline{\neg\mathsf{minimize}}> \wedge <adm(\neg\mathsf{minimize})> \theta_0 \wedge \\
&\quad (tt \wedge <adm(tt)> \theta_1) \wedge <\underline{\mathsf{minimize}}> \qquad \mathit{inconsistent} \\
\Theta^1_h &= <\neg\mathsf{minimize}> \wedge <adm(\neg\mathsf{minimize})> \theta_0 \wedge (tt \wedge <adm(tt)> \theta_1 \wedge \\
&\quad \neg TS \wedge <adm(\neg TS)> \theta_5
\end{aligned}
$$

$$
\begin{aligned}
\Theta^2_a &= \Theta^4_a = \Theta^6 = \theta_4 \\
\Theta^2_b &= \theta_4 \wedge \theta_5 \\
\Theta^2_c &= \theta_1 \\
\Theta^2_d &= \theta_1 \wedge \theta_5 \\
\Theta^2_d &= \Theta^0
\end{aligned}
$$

$$
\begin{aligned}
\Theta^3_a &= \Theta^5_a = \Theta^7_a = Graph \\
\Theta^3_b &= \Theta^5_b = \Theta^7_b = tt \wedge <adm(tt)> \theta_4
\end{aligned}
$$

Figure 9: Continuation formulas for the final specification

In fact, this formula better reflects the original verbal formulation of the problem (which also does not require any sequencing of bisimulation and minimization). The application of our algorithm to this specification yields the desired result:

$$showgraph \circ mkgraph \circ op\text{-}sem \circ CCS\text{-}rewr$$

The corresponding stepwise execution is given in Figures 8 and 9.

6 Conclusions

In this paper we have presented a framework for the *automatic configuration* of large systems from a library of reusable software components. Core of the framework is a *modal logic* that uniformly and elegantly captures *type descriptions*, *module specifications*, and *relative time*. Besides allowing an elegant and transparent specification of module configurations, our framework also provides a *minimal* *model generator* that automatically generates minimal solutions to a specification problem. All this has been illustrated for a practically relevant application: the automatic configuration of heterogeneous analysis systems from loose specifications.

Currently, we are only considering sequential module configurations, which are sufficient for our applications. However, it is also possible to extend the approach to other module connectives, whose clean integration is a topic of current research. Of particular interest are situations where modules are configured in parallel. In the simplest case, this can be modelled synchronously by considering cartesian products for types and modules. A more general treatment requires an extension of the model structure as such (cf. [EmCl82, MaWo84]).

We are also considering optimizations of our algorithm, like on the fly reductions of the modal formulas generated by the routines *Split* and *Red*, or heuristics to cut the search space. Another interesting question we want to address concerns specification patterns that allow a polynomial time minimal model construction.

References

[BoSV88] G. Boudol, R. de Simone, D. Vergamini: "*Experiment with Auto and Autograph on a Simple Case Sliding Window Protocol*", INRIA Report 870, July 1988.

[ClES86] E. Clarke, E. Emerson, A. Sistla: "*Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications*," ACM TOPLAS Vol.8, N.2, 1986, pp.244–263.

[ClPS89] R. Cleaveland, J. Parrow, B. Steffen: "*The Concurrency Workbench*," *Proc. CAV'89*, LNCS 407, Springer Verlag, 1989.

[ClPS93] R. Cleaveland, J. Parrow, B. Steffen: "*The Concurrency Workbench: A semantics-based verification tool for finite state systems*," ACM TOPLAS Vol.15, N.1, 1993, pp.36-72.

[EmCl82] E.A. Emerson, E.M. Clarke: "*Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons*," Science of Computer Programming, 2 (1982), pp.241-266.

[Emer90] E.A. Emerson: "*Temporal and Modal Logic*," In Handbook of Theoretical Computer Science, Ed. Jan van Leeuwen, Elsevier, 1990, pp.995-1072.

[Fern88] J.-C. Fernandez: "*Aldébaran: Une Système de Vérification par Réduction de Processus Communicants*," Ph.D. Thesis, Université de Grenoble, 1988.

[FA94] B. Freitag, K. Avini: "*A Hypertext-Based Tool for Large Scale Software Reuse*", submitted to CAiSE*94, available as Techn. Report, Technische Universität München, München (Germany).

[FrMS94] B. Freitag, T. Margaria, B. Steffen: "*A Pragmatic Approach to Software Synthesis*", to be presented at the ACM SIGPLAN Worksh. on Interface Definition Languages, Portland, Oregon, Januar 1994, - to appear on ACM SIGPLAN Notices.

[FSSBG93] B. Freitag, H. Schütz, G. Specht, R. Bayer, and U. Güntzer. *LOLA – a deductive database system with integrated SQL-database access*, Techn. Report, Technische Universität München, München (Germany), 1993.

[FSS91] B. Freitag, H. Schütz, and G. Specht. *LOLA — a logic language for deductive databases and its implementation*, In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91)*, pages 216 – 225, Tokyo, 1991.

[Hims93] M. Himsolt: "*Sgraph User Manual*," Techn. Report FMI, Universität Passau, Passau (Germany), 1993, in preparation.

[MaWo84] Z. Manna, P. Wolper: "*Synthesis of Communicating Processes from Temporal Logic Specifications*," ACM TOPLAS Vol.6, N.1, Jan. 1984, pp.68-93.

[McCu90] W.W. McCune: "OTTER *2.0 Users Guide*," ANL-90/9 Mathematics and Computer Science Division, Argonne Nat. Labs., Argonne, IL (USA), 1990.

[Miln89] R. Milner: "*Communication and Concurrency*," Prentice Hall, 1989.

[Pnue77] A. Pnueli: "*The temporal logics of programs*," Proc. 18th FOCS, Providence (RI) - USA, IEEE, New York 1977, pp.46-57.

[RRSV87] J. Richier, C. Rodriguez, J. Sifakis, J. Voiron: "*Verification in XESAR of the Sliding Window Protocol*" Proc. of the 7th IFIP Symposium on Protocol Specification, Testing, and Verification, North-Holland, Amsterdam, 1987.

[RoSi91] V. Roy, R. de Simone: "*Auto/Autograph*," In Proc.Computer-Aided Verification '90, American Mathematical Society, Providence, 1991.

[SiCl85] A.P. Sistla, E.M. Clarke: "*The Complexity of the Propositional Linear Temporal Logics*," Journal of the ACM, Vol.32, N.3, July 1985, pp.733-749.