UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Heterogeneous Analysis and Verification for Distributed Systems

Bernhard Steffen, Tiziana Margaria, Andreas Claßen

Lehrstuhl für Informatik
Universität Passau
Innstr. 33
D-94032 Passau (Germany)
tel: +49 851 509.3090
fax: +49 851 509.3092
{steffen,tiziana,classen}@fmi.uni-passau.de

MIP-9509
Juli 1995

Abstract

In this paper we present an environment for the development of special purpose heterogeneous analysis and verification tools, which is unique in 1) constituting a framework for the development of application specific heterogeneous tools and 2) providing facilities for the automation of the synthesis process. Based on a specification language that uniformly combines taxonomic component specifications, interface conditions, and ordering constraints, our method adds a global view to conventional single component retrieval. Following a user session, we illustrate the interactive synthesis process, which supports the inclusion of a satisfactory new software component into the repository by proposing an appropriately precomputed default taxonomic classification. This guarantees convenient retrieval for later reuse.

Contents

1 Motivation

To master the complexity and variety of system-level distributed system designs flexible environments are required, which efficiently support automation, 'design in the large' strategies and teamwork. In particular, several, drastically different methodologies must be available in order to allow an application specific treatment of problems. Unfortunately, existing systems provide only few tools and alternatives, and the inclusion of new tools into the system is quite a hard task due to the limited support for application specific tool selection or development. Particularly bound to a rich variety of approaches is the successful applicability of analysis and verification techniques to attack complex problems of practical relevance. Analysis and verification tools like the ones presented e.g. in [ClPS93, ScDa93, Fern88, BoSV88, RoSi91, RRSV87, MSGS88] therefore provide each quite a number of different methods, like e.g. abstract interpretations, bisimulation checking, theorem proving and model checking, but can only comfortably be used by experts. Moreover, their data and tool management is still quite primitive or completely missing.

Already such a large number of automatic and interactive techniques is currently available, that the oversight over their main characteristics is becoming impossible, not to mention their detailed application profile. Adequacy and possibility of combinations can be investigated case by case according to diverse criteria.

In this paper we present an environment for Computer-Aided Tool Synthesis (CATS), which is designed for handling this diversity by offering flexible tool integration, hypertext support for 1) the investigation of tools, 2) the current state of the tool repository and 3) the classification scheme for tools characterizing their application profile, as well as a powerful retrieval mechanism that searches for tool compositions satisfying a designer's query. In particular the last feature makes our environment unique in its support for the construction of heterogeneous tools from a repository of basic transformation and analysis components.

CATS is based on META-Frame [StMC95], which is a general purpose environment for the application-specific generation of complex, heterogeneous tools from a repository of basic transformation and analysis components. In a system-level verification setting for distributed systems, as the one considered here, the adequacy of this approach comes to its best illustration, since it offers designers a simple and effective means for generating a 'best possible' analyzer/verifier from a repository of already existing tools, algorithms and transformations on the basis of a possibly loose description of the purposes it should meet. This will be demonstrated following a user session, where an optimized tool for the heterogeneous verification of a specification/implementation relation is developed.

Our approach exactly meets the demands recently expressed by Goguen and Luqi in [GoLu95] for the emerging paradigm of *Domain Specific Formal Methods*:

1. *a narrow, well defined, well understood problem domain is addressed, for which a successful library may already exist,*
2. *there is a community of users who understand the domain,*

3. *the tool has a graphical user interface that is intuitive to the user community, embodying their own language and conventions,*
4. *the tool takes a* large grain *approach; rather than synthesizing procedures out of statements, it may use a library of components and synthesize code for putting them together,*
5. *inside the tool is a powerful engine that encapsulates formal methods concepts and/or algorithms; it may be a theorem prover or a code generator; users do not have to know how it works, or even that it is there.*

The essence of Goguen's proposal is to use formal methods on a large or huge grain level rather than on elementary statements, thus to support the programming with whole subroutines and modules as the elementary building blocks. This is precisely what META-Frame is designed for.

The next section will give an overview over META-Frame, together with the current application areas, Section 3 defines our specification language SLTL, Section 4 presents the synthesis component, Section 5 describes the repository for the current application, and Sections 6 and 7 illustrate the computer aided tool synthesis process along two concrete user sessions. Related work is discussed in Section 8, and the power and flexibility of our approach in Section 9.

2 The META-Frame Environment

META-Frame is a *meta-level framework* designed to offer a sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components. Figure 1 shows its overall organization. Special care has been taken in the design of an adequate, almost natural-language specification language, of a user-friendly graphical interface, of a hypertext based navigation tool, and in the automation of the synthesis process and of the repository management. This application-independent core has a firm theoretical basis: the synthesis is conceptually based on a model construction algorithm for a linear time temporal logic and supports an advanced form of Meta-level Intelligent Software Synthesis (see Section 4 and [SFCM94]).

The application-independent core is complemented by application-specific libraries of components, which constitute the objects of the synthesis. The principle of separating the component implementation from its description is systematically enforced: for each application we have a distinct Meta-Data repository containing a logic view of the components. The tools themselves and their documentation are available in a different repository. This organization offers a maximum of flexibility since the synthesis core is independent of the direct physical availability of the tools (except for the execution, which is a different matter).[1]

[1]Additional features like e.g. the advanced data and project management features (already offered by

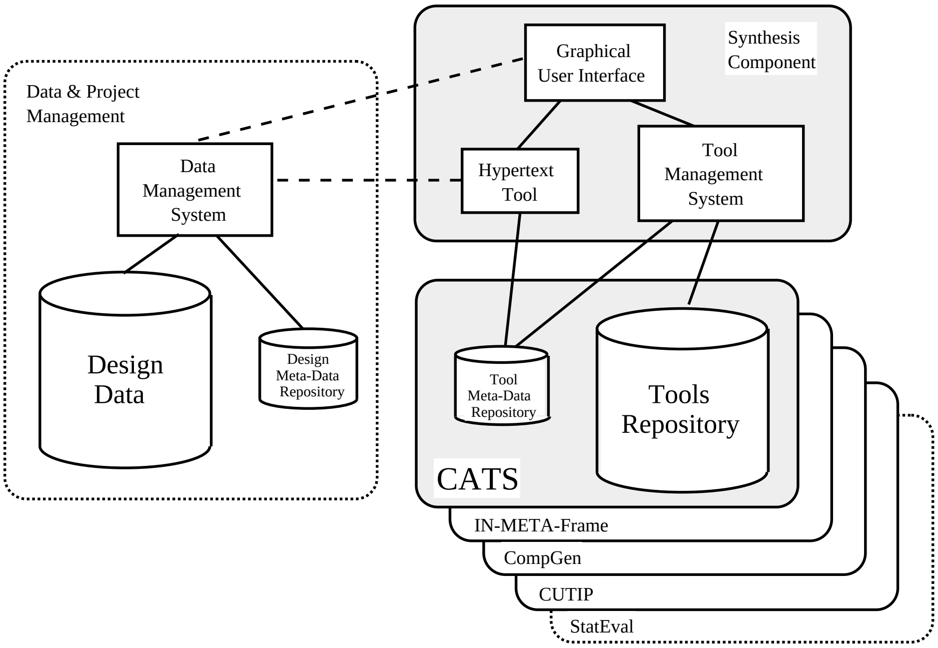

Figure 1: The META-Frame Global Architecture

Summarizing, META-Frame allows a designer the rapid and reliable realization of efficient application specific complex tools without a sophisticated user interaction, and is therefore an ideal means for a systematic investigation and construction of adequately designed tools for specific problem structures.

The META-Frame approach abstracts from implementational details by allowing designers a *high-level-development* of the tools:

- **Specifications** express constraints in a *temporal logic*, SLTL, that uniformly and elegantly captures an abstract view of the repository (see Section 3).
- **Implementations** are in a *high-level language* (HLL), tailored to express the combination of reusable analysis, verification and transformation components stored in the repository, which are considered as atomic on this level.
- **Synthesized tools** are hierarchically structured. Since only the meta-structure, combining such algorithms, is automatically generated from the specifications, the efficiency of the resulting tool depends on the efficiency of the single components. In particular, different components may be written in different application languages (C, ML, C++) of different programming paradigms (imperative, functional, object-oriented).

object oriented CAD frameworks as NELSIS [WoBD90]), could be integrated with little effort.

The current experience has shown the practicability and flexibility of the META-Frame approach in several areas of application:

- CATS: automatic generation of complex tools from a repository of components, which is the central application in this paper,
- IN-META-Frame: semi-automatic programming of advanced (new generation) Intelligent Network Services (joint project with Siemens Nixdorf, Munich)
- CompGen: determination of the correct ordering of the individual analyses and optimizing transformations in a data flow analysis/program optimization setting [KnRS95],
- CUTIP: automatic composition of complex UNIX piping sequences for text and image processing [SFCM94] (DaCapo project, in cooperation with FORWISS[2]),
- StatEval: automatic generation of tools for the flexible evaluation of statistic data (in preparation).

META-Frame is currently available for SUN SparcStations under UNIX, for PCs under LINUX, and for Siemens RM Machines under SINIX. Its graphical interface as well as the hypertext browser are built on top of the Tcl/Tk graphics library [Oust94]. Target language is a high-level PASCAL-like language, which is tailored to the combination of complex programs written in different programming languages. It is implemented in C++.

3 The Metalevel Specification Language

The synthesis component of META-Frame determines the set of solutions to specifications given in terms of global and local constraints. These constraints can uniformly and elegantly be formulated within the temporal logic SLTL (Semantic Linear Time Logic), which combines *type*, *module*, and *ordering* constraints to a specification language with a coherent semantics (see [StMF93] for details). Thus, given an SLTL specification, all legal program compositions satisfying these constraints are synthesized.

The following two paragraphs present our languages for expressing *local* constraints, capturing interfacing and module specifications, and *global* constraints, restricting the ordering between modules.

Local Constraints: The Static Aspect. They are the selection criteria for single components. Currently they are based on a taxonomic classification of the repository components whose design is the crucial part of the META-Frame instantiation for a specific application. This choice does not impose any limitation, as our approach is parameterized in the local constraint mechanisms. In particular, the taxonomic classification of components described by Prieto-Diaz [Prie91, PrFr87], which can be found in most

[2]Bavarian Research Institute for Knowledge Based Systems

software library systems, could be easily integrated in terms of local constraints in our system.

- *Interface* or *Type constraints* describe the 'neighbourhood' of programs. In our CATS application we consider type compatibility as the only criterion: Types are organized in taxonomies according to their hierarchical classification.
- *Module constraints* restrict the set of legal elementary programs which may appear in the solution. Like types, also the modules are organized in taxonomic hierarchies.

  Figure 3(low) shows an excerpt of the module taxonomy used in the following Sections as presented by the hypertext system. This taxonomy is a directed acyclic graph, where the sinks represent concrete modules (here labelled with the concrete module names), which are atomic entities in the taxonomy, and where intermediate nodes represent groups, i.e., sets of modules satisfying some basic property (expressed by predicates).

Type and module constraints are formulated as simple propositional logic formulas over the respective taxonomies, which are regarded as definitions of sets of basic predicates (atoms).[3] As an example, in the module taxonomy of Figure 3(low) the module constraint

`minimize and not bisim`

individuates the set of modules {`ES_rewr, collapse`}, which are minimization algorithms that are not based on bisimulation equivalence.

Global Constraints: The Temporal Aspect. *Ordering constraints* allow users to specify a relative ordering between (some of) the components that form a solution. Using a temporal logic as a framework, we are able to extend the two local kinds of specifications to an elegant and uniform specification language for global constraints, which is defined as follows:

Definition 3.1 (SLTL) *The syntax of Semantic Linear Time Logic (SLTL) is given by:*

$$\Phi ::= \; tt \mid \mathsf{type}(t_c) \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{<}m_c\mathsf{>}\ \Phi \mid \mathbf{G}\Phi \mid \Phi\mathbf{U}\Psi$$

where t_c and m_c represent type and module constraints, respectively.

SLTL formulas are interpreted over the set of *legal program compositions*, i.e. sequences of modules, where the output type (signature) of a module coincides with the input type (signature) of its successor. Intuitively,

- tt is satisfied by every program composition.

[3] In general the full power of the deductive database language can be used to express local constraints. It is therefore possible to incorporate any sophisticated classification scheme and taxonomic retrieval into our system.

- $\mathsf{type}(t_c)$ is satisfied by every program composition whose first component satisfies the type constraint t_c.
- Negation and conjunction are interpreted in the usual fashion.
- $\texttt{<}m_c\texttt{>}\ \Phi$ is satisfied by program compositions whose first component satisfies m_c and whose *continuation*[4] satisfies Φ. In particular, $\texttt{<>}\ \Phi$ is satisfied by every program composition whose continuation satisfies Φ.
- $\mathbf{G}\Phi$ requires that Φ is satisfied for every postfix[5] of the program composition. As we are dealing with finite paths only, certain properties of this kind will never hold. Still, this operator is useful to express global type constraints.
- $\Phi \mathbf{U} \Psi$ expresses that the property Φ holds for all components of the program composition, 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 (strong until).

The on-line introduction of **derived operators** allows a modular and intuitive formulation of complex properties. In fact, as can be seen in Section 6, our built-in macro facility allows natural language-like specifications. As examples we introduce the dual operators

$$\begin{array}{lll}
\textit{Disjunction}: & \Phi \vee \Psi & =_{df} \ \neg\,(\neg\Phi \wedge \neg\Psi) \\
\textit{Box}: & [m_c]\Phi & =_{df} \ \neg < m_c > (\neg\Phi) \\
\textit{Eventually}: & \mathbf{F}\ \Phi & =_{df} \ \neg\mathbf{G}(\neg\Phi) \\
\textit{Weak Until}: & \Phi\ \mathbf{WU}\ \Psi & =_{df} \ \neg\,(\neg\Psi\ \mathbf{U}\ \neg\,(\Phi\ \vee\ \Psi))
\end{array}$$

and convenient derived operators which occur in the example:

$$\begin{array}{ll}
\mathbf{use}\ \ \Phi & =_{df} \ \wedge\ \mathbf{F}\Phi \\
\Phi\ \mathbf{before}\ \Psi & =_{df} \ (\neg\Psi)\ \mathbf{U}\ (\Phi \wedge \neg\Psi)\ \wedge\ \mathbf{F}\ \Psi \\
\Phi\ \mathbf{nlt}\ \Psi & =_{df} \ (\neg\Psi)\ \mathbf{U}\ \Phi\ \wedge\ \mathbf{F}\ \Psi \\
\Phi\ \mathbf{then}\ \Psi & =_{df} \ \mathbf{F}\ (\Phi\ \wedge\ \mathbf{F}\ \Psi)
\end{array}$$

4 The Synthesis Component

Figure 2 illustrates the interplay of the META-Frame components during the synthesis process. Given a specification (in a user-friendly, almost natural-language form) our system provides a graph representing the set of all satisfying tool-compositions which can be built on the bais of the underlying repository. These solutions may differ in the verification techniques they use, their application profiles, and, because of the potential looseness of specifications, in their overall behaviour. CATS is designed to manage this diversity by providing users with appropriate decision support via graphical representations and the hypertext system.

[4]This continuation is simply the program composition starting from the second module.
[5]'Postfix' is a straightforward generalization of continuation.

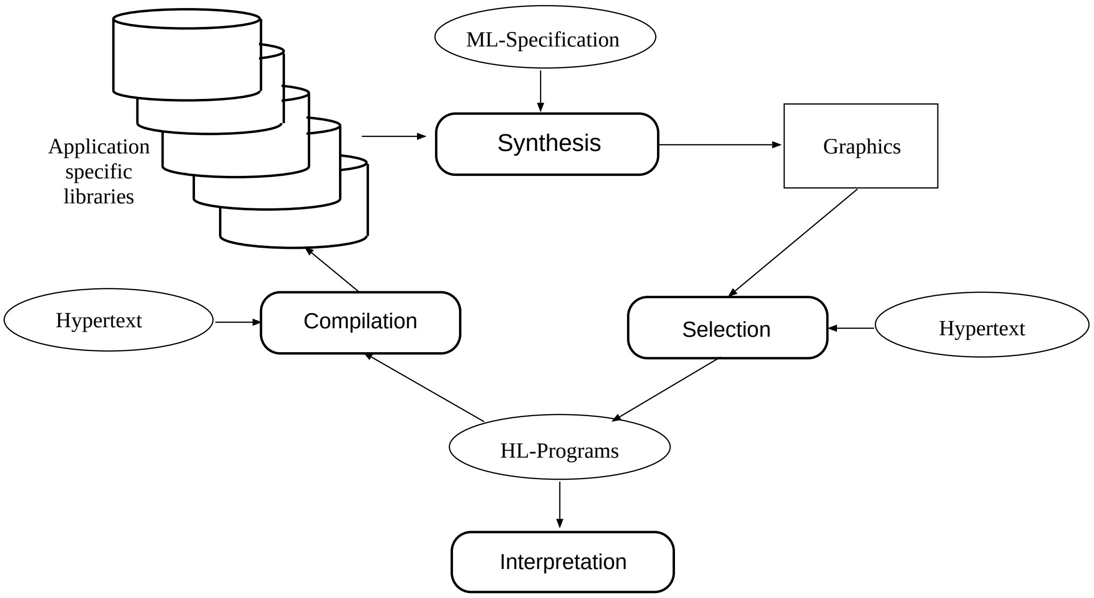

Figure 2: The Synthesis Process

During the orientation phase, where the user may not yet have a clear picture of a possible solution or even of the tools available in the repository, the flexibility of the specification language supports an efficient search: starting with an initial specification, he can successively strengthen or loosen the constraints in order to navigate through the universe of possible solutions offered by CATS.

Advanced users, perhaps with some particular solution pattern in mind, are often confronted with a larger set of solutions than expected. This widens the horizon and often reveals better alternatives.

After the search is completed, the CATS interpreter allows a direct execution of the chosen tool composition, in order to check whether all desires are met. Satisfactory tool compositions can be compiled and permanently stored in the repository for later reuse, supporting a hierarchical tool design.

Our synthesis component works by model construction roughly along the lines of [MaWo84]. In general *global constraints* lead to an exponential worst-case complexity. This 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, some of which are already realized in our implementation.

In practice, the set of all solutions is often very large. Thus it is convenient to define some general constraints keeping the set of delivered solutions manageable. In our system, *minimality* of the solutions is implemented as an option: no shorter module composition satisfies the specification. This choice reflects the desire for simple solutions and it does not affect the solvability of the specification. Our experience confirms the claim that a system for the automatic solution of composition problems should always provide the simplest solution, which may then successively be refined in order to satisfy additional constraints. As we will see in Section 6, this can easily and modularly be achieved within our framework.

5 The Repository of Analysis and Verification Components

Already such a large number of automatic and interactive analysis and verification techniques for distributed systems is currently available, that the oversight over their main characteristics is becoming impossible, not to mention their detailed applicability profile in terms of tool semantics (e.g., which problem they solve, which method they use, which are their strengths, which their limitations), of technical information (e.g., input/output formats, configurable options, platform and operating system constraints), and tool availability.

Limiting the panorama to software produced in academia, a reasonably up-to-date general purpose verification setting should include tools like

- **equivalence checkers** for the most common kinds of equivalence (strong, weak, branching bisimulation, testing, failure equivalence, input/output and trace equivalence) and the corresponding preorder checkers (for specification/implementation checks),

- **model checkers**, diversified according to the kind of modal or temporal logic (linear, interval, branching time temporal logic), its versions (Hennessy-Milner Logic, alternation-free mu-calculus, full mu-calculus, CTL, CTL*, ACTL etc.), the method (global/local), the structure (transition systems, Kripke structures, Petri Nets, I/O automata, Büchi automata), the used encodings (e.g., BDD-based methods, with their variations),

- **theorem provers**, classified according to the underlying logic (propositional logic, first, second, higher order logic, constructive logic, Martin-Löf type theory, etc.), the degree of proof automation, the reasoning method (rewriting, induction, Knuth-Bendix completion, resolution, connection method, etc.),

- **decision procedures** tuned for very specific and costly problems,

which actually all come in many variants coping with infinite state-systems, values, time (discrete or dense), priorities, probabilities and combinations thereof.

The applicability of the tools can be investigated case by case according to criteria like diagnostic information in case the verification fails, existence of transformations from the

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*	*ES*	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.
ES-rewr	*ES*	*ES*	reduces an eq. system
mkgraph	*TS*	*Graph*	converts the model into a graph
shgraph	*Graph*	*S-Graph*	displays a graph

Table 1: Fragment of the Library of Components

syntax of the actual system's description into the tool's input format, graphic routines, format converters for the communication among different tools, etc.

The library we are working with handles a varied selection of basic tools, plus the necessary infrastructure modules. The working repository is then enriched with the synthesized tools, which quickly outnumber the elementary tools. Our environment can manage repositories of some hundreds of tools. Currently, a great deal of such components are taken from other existing systems, some are optimized versions of algorithms taken from the Concurrency Workbench [ClPS93], others come from a compiler project [Stef93] or are imported from the OTTER [McCu90] resolution-based and the TIP [Frau93] induction-based theorem provers. Of course, the design, implementation and integration of new elementary algorithms still goes on.

The small fragment of our repository of reusable components which will be used in the case studies of the next Sections is shown in Tab. 1. It includes analysis and transformation components (*part-s-bisim*, *part-w-bisim*, *collapse*, *char-form*), semantic interpretation functions (*op-sem*), some modules for minimization by rewriting (*CCS-rewr*, *ES-rewr*), and graphic modules (*mkgraph, shgraph*). Relevant basic types are: modal logic equational systems *ES*, *CCS* [Miln89] expressions, transition systems *TS*, partitions *Part* on the state space of transition systems, and internal and displayable representations of graphs (*Graph, S-Graph*).

The taxonomic classification which defines groups over the atomic modules is shown in Figure 3 (middle) as it is displayed by our hypertext component, together with the cards for the group minimize (left) and for the atom part_w_bisim. The cards contain a textual description (expandable to full documentation via hypertext links) and information about the collocation in the taxonomy. For the atoms they also contain their input and output types, which in particular serve as links to the type taxonomy.

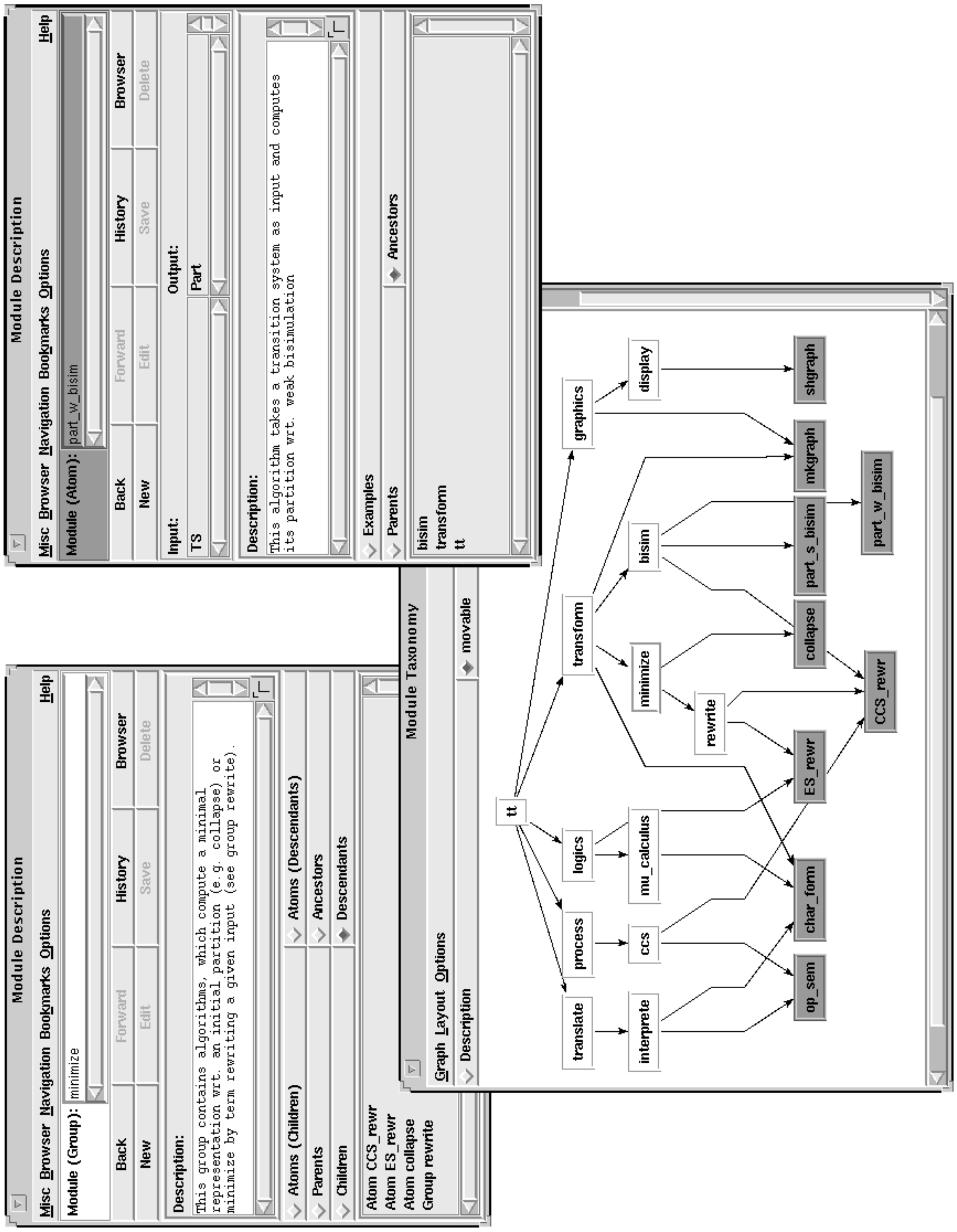

Figure 3: Hypertext: the Taxonomy over the Library Fragment

The following two sections illustrate the central features of CATS following a user session for the development of an optimized tool for the heterogeneous verification of a specification/implementation relation.

6 Working with the Synthesis Component

To illustrate the 'pure' synthesis process sketched in Figure 2, we follow a user session attacking the following loosely specified task

(∗) *Minimize a CCS term* ccs *up to bisimulation and display the result graphically.*

on the basis of the fragment of the repository of basic commands introduced in Section 5. This problem requires the synthesis of an efficient algorithm for the minimization wrt. bisimulation equivalence of distributed systems described in the process algebra CCS [Miln89].

The Specification. Already the following still natural-language like reformulation of the informal specification is understood by the system

```
type ccs,
     use bisim before minimize,
         and finally module display
```

Abbreviating $< m > tt$ by $< m >$, the informal specification above corresponds to the modal formula:

$$\Phi_0 \;=\; ccs \;\wedge\; (< \mathsf{bisim} > \;\textbf{before}\; < \mathsf{minimize} >)\;\textbf{before}\; < \mathsf{display} >$$

which is expanded in terms of primary SLTL operators and rewritten to the input format of the synthesis algorithm.

The Solution Space. Given this formula, the system returns two minimal length alternative program compositions in graph representation: every path from the start to the success node in the graph of Figure 4 (right) is a possible shortest solution to our synthesis problem.

In the solution graph, nodes represent modules (atoms) labelled by their name, and edges are labelled by type information.

Solutions can be investigated by means of the hypertext browser, which provides information about single modules and types as well as about the type and module taxonomies. The hypertext system offers via a mouse click a card for each node or edge of the graph.

In the current example, both solutions take a term of type *CCS*, generate the corresponding labelled transition system (type *TS*) by computing its operational semantics via the module *op_sem*, and compute partitions of the state space into equivalence classes,

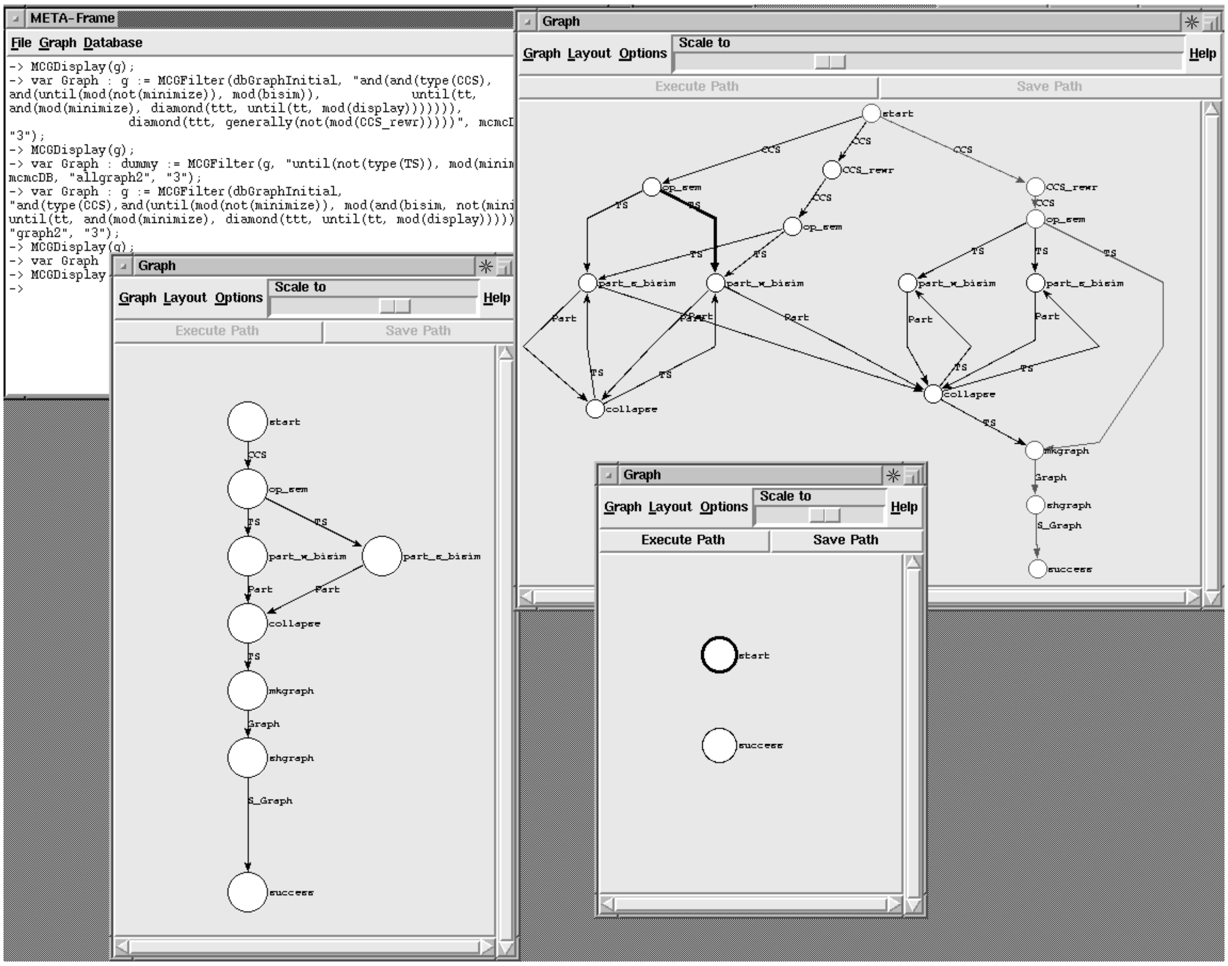

Figure 4: Synthesizing an Efficient Minimizer

either according to weak or to strong bisimulation equivalence (modules *part_w_bisim* or *part_s_bisim*). – Both alternatives are correct, since the problem formulation left the choice of bisimulation criteria unspecified. – These partitions (type *Part*) are reduced to single states by the *collapse* algorithm, which delivers a new, minimized, transition system of type *TS*, which is then converted to a *Graph* format via *mkgraph* and transformed to a displayable graph containing layout information via *shgraph* (type *S_Graph*). Note that these solutions contain (necessary) modules which are not mentioned in the specification.

The Execution. All the proposed solutions are directly executable. In fact, each path in the graph corresponds to a target program in our high level programming language, which is implemented in C++. In order to invoke a solution, its corresponding path must be selected by means of the mouse. The system offers the option for execution, as soon as the selection is unambiguous. Both solutions require an interactive selection of a input CCS term, which can be typed in a pop-up window or loaded from a file.

Modifying the Query. In the *prototyping* phase, the user approaches the desired solution by successively refining or relaxing an initial (usually loose) specification on the basis of some hypertext investigation or experiments with automatically generated prototypes. In fact, in our case these experiments reveal a major flaw of the proposed solutions: the full transition system for the CCS term *ccs* is computed before the minimization is applied. This is unsatisfactory, since it restricts the applicability of the synthesized solutions to terms where the *state explosion* problem does not strike. In general, the transition system for a parallel combination of processes may in fact grow exponentially in the number of parallel components leading to unmanageable state sizes. In order to limit such "bad" behaviour, one could require that the construction of the transition system must be delayed until a minimization is applied. This requirement can be made precise in SLTL by means of ordering constraints:

```
not type ts until module minimize
```

Adding this formula as a conjunct to the previous query leads to

$$\Phi^1 \;=\; \Phi^0 \;\wedge\; (\neg\mathsf{type}\,(TS)\;\mathbf{U}\;<\mathsf{minimize}>)$$

To our knowledge, META-Frame is the only framework for the synthesis of module compositions that allows such a flexible treatment using global constraints.

Figure 4 (middle) shows the solution space corresponding to the modified query. The graph is empty! Thus the modified query is unsolvable. The strict sequentialization of bisim and minimize in Φ^0 is not compatible with the additional efficiency constraint in the considered fragment of the module library.

Eliminating the Overspecification. The original verbal formulation of the problem (*) does not require any sequencing of bisimulation and minimization. Thus we can improve our formal specification simply by replacing the **before** operator by the weaker **nlt** operator:

```
type ccs,
     use bisim nlt minimize,
          finally module display
              and not type ts until module minimize
```

The fact that the module *CCS-rewr* satisfies bisim as well as minimize offers completely different solutions to the problem. Figure 4 (upper left) displays the full spectrum of solutions: each path is a correct implementation. Concentrating again only on minimal solutions leaves us with the leftmost path

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

which is highlighted in red on the screen by our system.

The Compilation. Satisfactory solutions can be made persistent through compilation into a new module that can be saved in the component repository for later reuse. These repository updates are supported by the hypertext system by automatically associating with each new module a new card, which by default contains appropriately precomputed taxonomic criteria and a short textual description. These criteria are generated by the system from the corresponding criteria of its components. Thus the consistency of the taxonomies is automatically maintained. The user only needs to provide a new name for the new component, then the repository and taxonomies are updated automatically. Of course, the proposed content of the new hypertext card can be also modified interactively.

7 A Synthesis-Based Interactive Editor

The META-Frame environment can also be used as a sophisticated editor for large to huge grain programs written in our high-level language. In this setting, the implementation of large programs is supported by the automatic synthesis of linear portions, resulting in sequential compositions of modules. More complex control structures glueing the linear portions together must be programmed by hand. Still, being able to synthesize linear program fragments drastically improves over other methods where only single components can be retrieved from the underlying repository. This is already true in cases where one is only interested in the functionality of single components, because our synthesis algorithm will automatically determine the required interfacing modules.

We illustrate the use of the editor by showing how to synthesize an efficient tool checking a heterogenous specification-implementation relation for distributed systems specified as process algebra terms. This particular relation was introduced in [ClSt90], but the synthesis procedure can easily be modified to generate verifications tools for any kind of simulation preorder.

Starting point for the synthesis of the preorder checker is the following intuitive specification:

(**) *Preorder-check two CCS terms ccs-expr*$_1$, *ccs-expr*$_2$, *after their minimization*

Since the preorder checker handles a pair of input terms, we will

1. decompose the problem into a non-linear context component and two linear portions. The specification of the non-linear context components implicitly defines the specifications for the two linear subproblems, which can then be automatically treated by our synthesis algorithm.

2. synthesize each subproblem along the lines of the previous section. In this step interaction allows to successively refine the specifications in order to obtain optimized solutions,

3. and finally, construct the global implementation from the partial solutions synthesized in the previous steps.

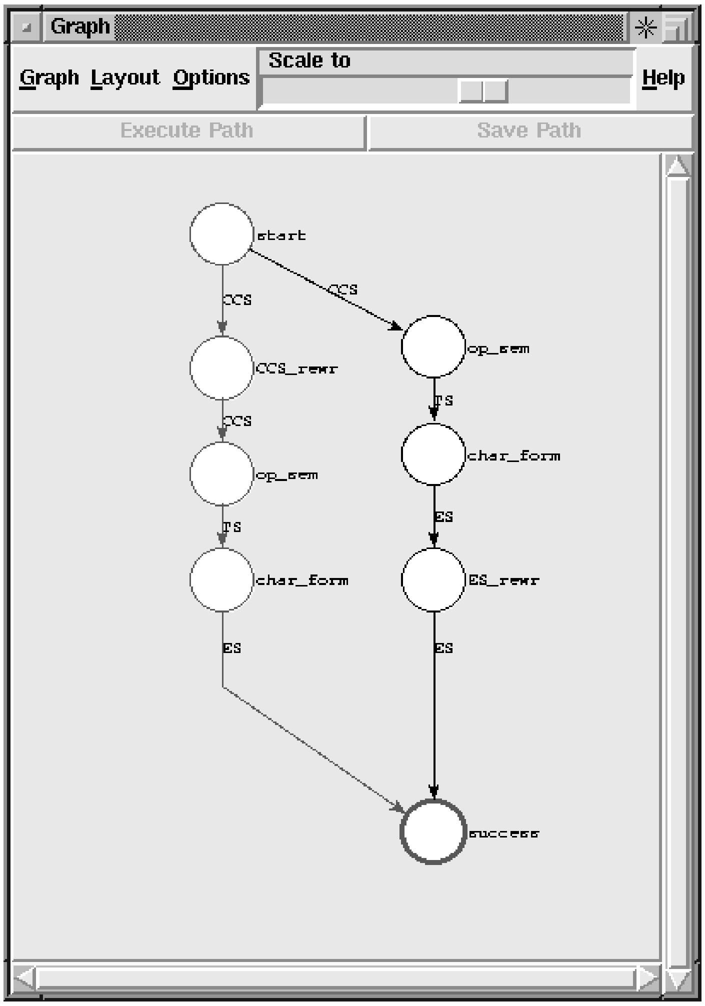

Figure 5: The Transformers to Equational Systems

In the following we discuss our examples according to these three steps.

1. Interactive Problem Decomposition. According to [Stef89, StIn94], preorder checking can be realized through model checking *mod-check* (ts_2, es_1), where ts_2 is the transition system corresponding to the expression *ccs-expr*$_2$, and es_1 is the characteristic formula for *ccs-expr*$_1$. This yields the most efficient preorder checker currently known [CeCl93, ClSt91][6].

A fast suitable model checker is e.g. the one introduced in [SCKK95]. The synthesis problem $(**)$ additionally requires that each of the argument processes is minimized before the model checker is applied[7]. This induces two independent linear synthesis subproblems, which we solve automatically along the lines of the previous section.

[6] A similar algorithm was also presented in [BlPa92].

[7] Minimizing the argument processes before verification is a powerful standard technique [ClLM89, GrSt90, GrLo92, DaGG93].

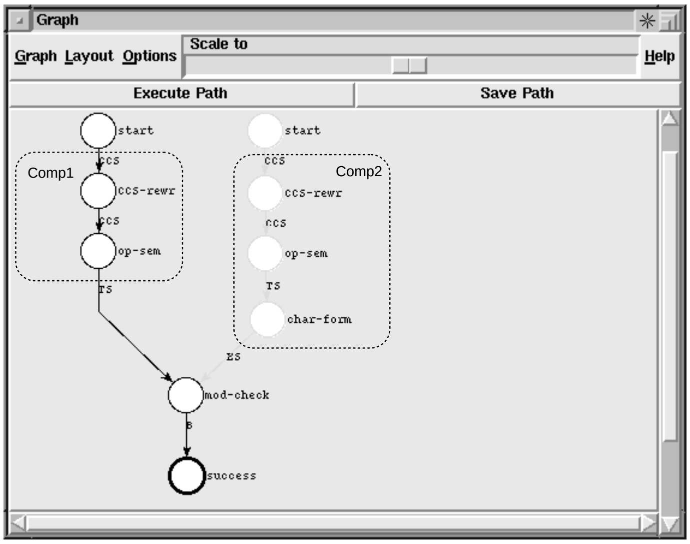

Figure 6: The Optimized Preorder Checker

1. The optimized solution for the conversion of the expression $ccs\text{-}expr_2$ into a transition system is part of the problem discussed in detail in the previous section. The resulting implementation is *op-sem (CCS-rewr (ccs-expr*$_2$*))* , which constitutes the component **Comp 1** of Figure 6.

2. The minimization of the expression $ccs\text{-}expr_1$ and its transformation into an equational system is described below.

2. Synthesizing the Transformation to Equational Systems. The specification is now

```
type ccs, use minimize then type es
```

which can be implemented on the basis of the fragment of the module library shown in Figure 3(middle).

The synthesis process delivers the two minimal implementations displayed in Figure 5. Remembering that one should not generate unminimized transition systems, we let META-Frame evaluate the proposed solutions with respect to the efficiency constraint `not type ts until module minimize`. This additional constraint only leaves the left path, highlighted in red on the screen. Thus we obtain the unique solution

$$char\text{-}form\ (op\text{-}sem\ (CCS\text{-}rewr\ (ccs\text{-}expr_1)))$$

corresponding to component **Comp 2** in Figure 6.

3. The Global Program. We revert the decomposition of the first step, and obtain the following implementation for the optimized preorder checker:

$$\begin{aligned}&\textit{mod-check (op-sem (CCS-rewr (ccs-expr}_2)),\\&\qquad\qquad \textit{char-form (op-sem (CCS-rewr(ccs-expr}_1)))).\end{aligned}$$

which is heterogeneous, since it contains a model checker for the modal μ-calculus, a graph rewriting system for CCS, and algorithms to compute the operational semantic from CCS terms and characteristic formulas in the μ-calculus from transition systems.

8 Comparison with Related Work

Most software library systems described in the literature are **local**, in the sense that they help retrieving single components. E.g.:

- LaSSIE [DBSB91] uses the knowledge representation language KANDOR to store information about a large software project in order to provide semantic retrieval. A taxonomy of classes is induced by a frame-based representation of objects.
- The software engineering environment NORA [SFGK94] is based on logical inference and supports specification-driven single component retrieval.
- Other systems share a core constituted by the knowledge representation language Telos [MBJK90] and the deductive object-oriented database system ConceptBase [Jark91]:
  - In the ITHACA Environment [FuNP92], an application frame contains application classes with a functional component description in terms of a list of operations and information about the meta-level classes with design knowledge describing the use of components. Component selection is based on similarity in the spirit of Prieto-Diaz' work [Prie91, PrFr87].
  - A logic-based software configuration management system is presented in [JJMG92] using the Telos data model for the representation of configuration information. Given a system interface, the search for an implementation tries to reuse existing subsystems and components if possible. Otherwise, proposals are generated applying the deductive and abductive techniques of ConceptBase.
  - An approach based on algebraic specifications and refinement is described in [ChHJ93]. Given an algebraic specification of the desired component in Telos, ConceptBase is used to retrieve software components with matching signatures.

The point of META-Frame, however, is its synthesis capability according to **global** constraints. We know only of three other systems with this feature:

- Amphion [SWLP94] relies on the SNARK resolution-base theorem prover for first order logic for the automatic composition of astronomical software from the Fortran77 subroutine library NAIF. An application domain theory describes the procedures in (a portion of) the library as well as some basic properties of solar-system astronomy which are given as first-order axioms. Graphical specifications (forming a subset of predicate logic) are translated into an existential theorem, which is proved constructively, i.e., by building a solution as composition of basic routines. The deduction along which the implementation is constructed can be fully automatic, but experienced users may increase the efficiency by providing SNARK with guidelines for the proof.

  In contrast to our approach, which support loose specifications, Amphion requires precise specifications, which are interactively constructed with the help of a menu of alternatives. Thus the processes of interaction are quite different and the tools support different application profiles.

- The system reported by Batory and O'Malley [BaMa92] constructs hierarchical software systems from reusable components using grammar rules for high level specification. Their approach strongly relies on structural restrictions of the underlying environment like open architecture software and a strict interface discipline.

- Möhring, Müller and Radermacher, [MöMR94], propose the idea to use scheduling methods to synthesise implementations satisfying precedence constraints. However, they do not provide an appropriate specification language or hints towards its intended expressive power.

9 Perspectives

The experience with our tool is very promising: not only is it unique in making available to the users an unprecedented wealth of special-purpose analysis and verification tools, but it also efficiently helps them in mastering the diversity and complexity of their application profiles. This feature is essential to encourage successful experimentation even in difficult application areas like the analysis and verification of distributed systems without requiring much expertise. Moreover, the tool supports the realization of a highly distributed analysis and verification environment, where tools are logically present and only physically retrieved at need. These properties are particularly important in emerging areas like e.g. the development of distributed systems or Hardware/Software codesign, where the integration of heterogeneous methods is a key point.

Acknowledgements

We are grateful to Volker Braun, Achim Dannecker, Carsten Friedrich, Andreas Holzmann, Dirk Koschützki, Gerald Lüttgen, Falk Schreiber, Markus Schweighofer, Matthias Seul for their constructive cooperation in the design and implementation of META-Frame, as well as to Burkhard Freitag and Uli Zukowski for early discussions and the implementation of a first prototype.

References

[BaMa92] D. Batory, S. O'Malley: "*The design and implementation of hierachical software systems with reusable components*" ACM Transactions on Software Engineering and Methodology, Oct. 1992.

[BlPa92] B. Bloom, R. Paige: "*Computing Ready Simulations Efficiently*", Proceedings of the North American Process Algebra Workshop (NAPAW), 1992.

[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.

[CeCl93] U. Celikkan, R. Cleaveland: "*Generating Diagnostic Information for Behavioral Preorders*," Submitted to *Distributed Computing*, 1993

[ChHJ93] P.S. Chen, R. Hennicker, M. Jarke: "*On the retrieval of reusable software components*", Advances in Software Reuse - Selected Papers from the 2nd Int. Workshop on Software Reusability, Lucca, Italy, pp. 99–107. IEEE Computer Society, 1993.

[ClKS92] R. Cleaveland, M. Klein, B. Steffen: "*Faster Model Checking for the Modal Mu-Calculus*," Proc. of CAV'92, LNCS 663, Springer Verlag, 1992.

[ClLM89] E. Clarke, D. Long, K. McMillan: "*Compositional Model Checking*," Proc. LICS'89, Pacific Grove (CA), June 1989, pp.353–362.

[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.

[ClSt90] R. Cleaveland, B. Steffen: "*A preorder for partial process specification*," Proc. CONCUR'90, LNCS 458, Springer Verlag, 1990, pp.141–150.

[ClSt91] R. Cleaveland, B. Steffen: "*Checking Behavioural Relations, Logically*, " in Proc. ICALP'91, Madrid, LNCS 526, Springer Verlag, 1991.

[DaGG93] D. Dams, O. Grumberg, R. Gerth: "*Generation of Reduced Models for Checking Fragments of CTL*," Proc. CAV'93, Crete (GR), June 1993, LNCS 697, Springer Verlag, 1993, pp.479–490.

[DBSB91] P. Devanbu, R.J. Brachman, P.G. Selfridge, B.W. Ballard: "*LaSSIE: A knowledge-based software information system*" Communications of the ACM, 34(5):35–49, May 1991.

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

[Frau93] U. Fraus: "*TIP : Term-rewriting Induction Prover, User Manual*," Tech. Rep. Universität Passau, Passau, Germany, to appear.

[FrMS94] B. Freitag, T. Margaria, B. Steffen: "*A Pragmatic Approach to Software Synthesis*", ACM SIGPLAN Worksh. on Interface Definition Languages, Portland, Oregon, January 1994, - also in ACM SIGPLAN Notices.

[FuNP92] M. Fugini, O. Nierstrasz, B. Pernici: "*Application development through reuse: the ITHACA tools environment*" ACM SIGOIS Bulletin, 13(2):38–47, Aug. 1992.

[GoLu95] J.A. Goguen, Luqi: "*Formal Methods and Social Context in Software Development*," (invited talk) 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.62-81.

[GrLo92] S. Graf, C. Loiseaux: "*Program Verification by Using Compositional Abstractions*," Project SPECTRE Techn. Rep. N.39, LGI, IMAG, Grenoble (F), Nov. 1992.

[GrSt90] S. Graf, B. Steffen: "*Using Interface Compositional Minimization of Finite State Systems*," Proc. of CAV'90, LNCS 531, Springer Verlag, 1991.

[Jark91] M. Jarke: "*ConceptBase V3.0 User Manual*". Technical Report MIP-9106, Universität Passau, 1991.

[JJMG92] M. Jarke, M. Jeusfeld, A. Miethsam, M. Gocek: "*Towards a logic-based reconstruction of software configuration management*" Proc. of the seventh Knowledge-Based Software Engineering Conference, McLean, Virginia, pp. 132–141. IEEE Computer Society Press, 1992.

[KaRT89] S. Katz, C. Richter, K. The: "*PARIS: A system for reusing partially interpreted schemas*" Software Reusability: Volume I – Concepts and Models, Frontier Series, chapt. 10, pp. 257-273. ACM Press, New York, 1989.

[KnRS95] J. Knoop, O. Rüthing, B. Steffen: '*The power of assignment motion*," Proc. of the ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation (PLDI'95), La Jolla, California, June 18 - 21, 1995, SIGPLAN Notices 30, 6 (1995), 233 - 245.

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

[MBJK90] J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: "*Telos – representing knowledge about information systems*", ACM Transactions on Information Systems, 8(4):325–362, Oct. 1990.

[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.

[MiMM94] A. Mili, R. Mili, R. Mittermeier: "*Storing and retrieving software components: A refinement based system*", Proc. 16th International Conference on Software Engineering, Sorrento, Italy. IEEE Computer Society, 1994.

[MöMR94] R.H. Möhring, R. Müller, F.J. Radermacher: "*Advanced DSS for scheduling: Software engineering aspects and the role of eigenmodels*" Proc. 27th Annual Hawaii Int. Conf. on System Sciences, Maui, Hawaii, 1994.

[MSGS88] J. Malhotra, S. Smolka, A. Giacalone, R. Shapiro: "*Winston: A Tool for Hierarchical Design and Simulation of Concurrent Systems*", Proc. of the Workshop on Specification and Verification of Concurrent Systems, University of Stirling, Scotland, 1988.

[Oust94] John K. Ousterhout: "*Tcl and the Tk Toolkit*," Addison–Wesley, April 1994.

[Prie91] R. Prieto-Diaz: "*Implementing faceted classification for software reuse*", Communications of the ACM, 34(5), May 1991.

[PrFr87] R. Prieto-Diaz, P. Freeman: "*Classifying software for reusibility*", IEEE Software, 4(1):6–16, Jan. 1987.

[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.

[ScDa93] R. Schlör, W. Damm: "*Specification and Verification of System-level Hardware designs using Timing Diagrams*," Proc. IEEE EDAC-EuroASIC'93, Paris (France), Feb. 1993, pp.518-524.

[SCKK95] B. Steffen, A. Claßen, M. Klein, J. Knoop. T. Margaria: "*The Fixpoint Analysis Machine*", to be presented at the 6th Int. Conference on Concurrency (CONCUR'95), Philadelphia (PA), August 21-24, 1995.

[SFCM94] B. Steffen, B. Freitag, A. Claßen, T. Margaria, U. Zukowski: "*Intelligent Software Synthesis in the* DaCapo *Environment*," Proc. 6th Nordic Worksh. on Programming Theory, Aarhus (DK), Oct. 1994, BRICS Report N.94/6.

[SFGK94] G. Snelting, B. Fischer, F.J. Grosch, M. Kievernagel, A. Zeller: "*Die inferenzbasierte Softwareentwicklungsumgebung NORA*" Informatik Forschung und Entwicklung, 9(3), Aug. 1994. (In German).

[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.

[Stef89] B. Steffen: "*Characteristic Formulae*," Proc. ICALP'89, Stresa (I), July 1989, LNCS 372, Springer Verlag, 1989.

[Stef93] B. Steffen: "*Generating Data Flow Analysis Algorithms from Modal Specifications*, " Science of Computer Programming N. 21, 1993, pp. 115-139.

[Stef94] B. Steffen: "*Finite Model Checking and Beyond*, " (invited talk) Proc. 6th Nordic Worksh. on Programming Theory, Aarhus (DK), Oct. 1994, BRICS Report N.94/6.

[StIn94] B. Steffen, A. Ingolfsdottir: "*Characteristic Formulae for Finite State Processes*," *Information and Computation*, Vol. 110, No. 1, 1994.

[StMC95] B. Steffen, A. Claßen, T. Margaria: "*The* META-Frame*: An Environment for Flexible Tool Management*," 6th Int. Conf. on Theory and Practice of Software Development (TAPSOFT'95), Aarhus (Denmark), May 1995, LNCS N.915, pp.791-792.

[StMF93] B. Steffen, T. Margaria, B. Freitag: "*Module Configuration by Minimal Model Construction*," Techn. Rep. MIP 93-13, Fakultät für Mathematik und Informatik, Universität Passau (Germany), December 1993.

[SWLP94] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood: "*Deductive Composition of Astronomical Software from Subroutine Libraries*", Proc. 12th Int. Conf on Automated Deduction (CADE'94), Nancy (F), June 994, LNAI N. 814, Springer V., pp. 341-355.

[WoBD90] P. van der Wolf, P. Bingley, P. Dewilde: "*On the Architecture of a CAD Framework: The NELSIS Approach*," Proc. EDAC'90, IEEE Computer Society Press, pp.29-33.