Proceedings of the International Workshop on

Software Tools for Technology Transfer

STTT '98

Aalborg, Denmark, July 12–13, 1998

Tiziana Margaria
Bernhard Steffen
(editors)

BRICS Notes Series NS-98-4
ISSN 0909-3206 June 1998

STTT'98
International Workshop on Software Tools for Technology Transfer

Organizers

Volker Braun, University of Dortmund (D)
W. Rance Cleaveland, NC State University (USA)
Tiziana Margaria, University of Passau (D)
Bernhard Steffen, University of Dortmund (D), (Chair)

Preface

This volume contains the proceedings of the International Workshop on *Software Tools for Technology Transfer*, STTT'98, which took place in Aalborg (Denmark) on July 12–13 1998, as a satellite of ICALP'98, the 25th *International Colloquium on Automata, Languages, and Programming.*

Tool support for the development of reliable and correct computer systems is in fact of growing importance: a wealth of design methodologies, algorithms, and associated tools have been developed in different areas of computer science. However, each area has its own culture and terminology, preventing researchers from taking advantage of the results obtained by colleagues in other fields: tool builders often are unaware of, and thus unable to use, work done by others. The situation is even more critical when considering the transfer of technology into industrial practice.
STTT'98 addressed this situation by providing a forum for discussion of all aspects of tools that aid in the development of computer systems in the light of a possible tool-oriented link between academic research and industrial practice. Accordingly, the event comprised

- a **one-day Workshop**, on *July 12th*, whose eight talks were organized in three sessions:
  - *Verification of Code Generation*: compiler-specific and program-specific verification.
  - *Model Checking*: variants, also comprising real time aspects.
  - *Technology Transfer*: initiatives and projects.
- a **two-days Tool Exhibition**, on *July 12th and 13rd*, which, in addition to the tools presented at the workshop, comprised a demonstration of the three tools described at the end of the proceedings.

We want to thank the local organization team for handling all the practical matters concerning the workshop, and in particular Josva Kleist who took care of all the technical frame conditions required for the demonstrations. Warm thanks are due also to Uffe Engberg for making the realization of this volume possible in record time, and to Claudia Herbers for her assistance in the coordination of the review process.

July 1998

Tiziana Margaria
Bernhard Steffen

Contents

The Code Validation Tool (CVT)–

Automatic verification of code generated from synchronous languages[*]

A. Pnueli, O. Shtrichman and M. Siegel

Contact author: amir@wisdom.weizmann.ac.il

The Weizmann Institute of Science,
Department of Applied Mathematics and Computer Science
Rehovot, Israel

Abstract

We describe CVT - a fully automatic tool for Code-Validation, i.e. verifying that the target code produced by a code-generator (equivalently, a compiler or a translator) is a correct implementation of the source specification. This approach is a viable alternative to a full formal verification of the code-generator program, and has the advantage of not 'freezing' the code generator design after verification.

The CVT tool has been developed in the context of the ESPRIT project SACRES, and validates the translation from StateMate/Sildex mixed specification into C. The use of novel techniques based on uninterpreted functions and their analysis over a BDD-represented small model enables us to validate source specifications of several thousands lines, which represents a typical industrial size safety-critical application.

1 Introduction

A significant number of embedded systems contain safety-critical aspects. There is an increasing industrial awareness of the fact that the application of formal specification languages and their corresponding verification/validation techniques may significantly reduce the risk of design errors in the development of such systems. However, if the validation efforts are focused on the specification level, the question arises how can we ensure that the quality and integrity achieved at the specification level is safely transferred to the implementation level. Today's process of the development of such systems consists of hand-coding followed by extensive unit and integration-testing.

The highly desirable alternative -- both from a safety and a productivity point of view -- to automatically generate code from verified/validated specifications, has failed in the past due to the lack of technology which could convincingly demonstrate to certification authorities the correctness of the generated code. Although there are many examples of compiler verification in the literature (See, for-example, [1][2][3] and [4]), the formal verification of industrial code-generators is generally prohibitive due to their size. Another problem with compiler verification is that the formal verification freezes their designs, as each change to the code generators nullifies their previous correctness proof.

Alternately, code-validation suggests to construct a fully automatic tool which establishes the correctness of the generated code individually for each run of the code generator. In general, code-validation is the key enabling technology to allow the usage of code generators in the development cycle of safety-critical and high quality systems. Remarkably, the combination of automatic code generation and validation improves the design flow of embedded systems in both safety and productivity by eliminating the need for hand-coding of the target code (and consequently coding-errors are less probable) and by considerably reducing unit/integration test efforts.

The work carried out in the SACRES project proves the feasibility of code-validation for the industrial code generators used in the project, and demonstrates that industrial-size programs can be verified fully automatically in a reasonable amount of time. In the next section we describe the SACRES project and the role of code validation in this context. In section 3 we briefly describe the logical basis of the correctness proof. In section 4 we describe the architecture of CVT and the role of each of its modules, and we summarize in section 5 by presenting preliminary results from an industrial case study that we are currently working on.

[*] This research was done as part of the ESPRIT project SACRES, and was supported in part by a grant from the Deutsche Forschungs Gemeinschaft, the Minerva Foundation, and an infra-structure grant from the Israeli Ministry of Science and Art.

2 Code validation in the context of the SACRES project

The Code Validation Tool (CVT) is developed as part of the ESPRIT-supported project SACRES (which stands for *Safety Critical Real-time Embedded Systems*)[7]. The objective of this project is to provide designers of safety-critical systems with an enhanced design methodology supported by a toolset, significantly reducing the risk of design errors and shortening the overall design time. The emphasis within the project is on formal development of systems, providing formal specification, model checking technology and validated code-generation.
The architecture of the SACRES toolset is shown in Figure 1.

Following is a typical scenario of usage of the toolset: After completing the design in her favorite design tool (currently the 'StateMate' and 'Sildex' tools are supported), the user invokes the automatic translation of designs into DC+, the common format for synchronous languages. The design can be mixed: different components can be designed in different tools, as long as these tools are supported in the toolset. In the next step the user invokes the *Proof-Manager*, and performs *component and system verification*. In this stage the user verifies that the design satisfies various properties, which she expresses in the requirement specification language of *Timing Diagrams,* using the *Timing-Diagrams Editor (TDE)*. These properties typically correspond to the requirements listed in a requirement document, or to general safety and liveness properties of the system, such as the absence of deadlocks.

The Proof-Manager combines BDD-based automatic verification tool and a theorem-prover, which is invoked when the automatic verification fails (typically due to the size of the model). The various components thus can be verified by different means, while the proof-manager guarantees that the necessary compositionality requirements are maintained. If the system finds a design error, it presents a counter example by means of simulation (either in StateMate or in TDE).

After the design is verified, the user invokes the code generator (produced by the SACRES partner TNI) to automatically generate executable code (C or ADA). This is where the code validation tool is invoked: The validation of the generated code via CVT establishes that the code generator worked as expected and thus the properties which were verified at the specification level are preserved at the implementation level. We expect that the process of code validation will provide the convincing evidence required by the certification authorities in order to allow the use of automatic code generators for the development of safety-critical systems.

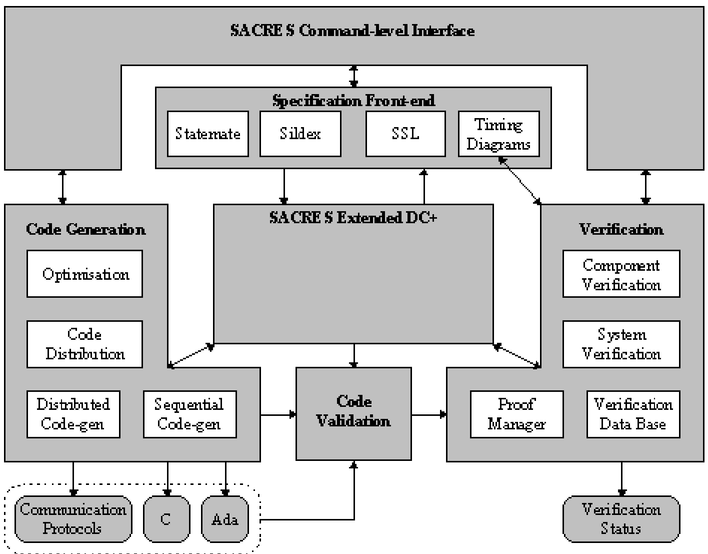


Fig. 1: SACRES Global Architecture

3 The Verification Condition

The theoretical background behind the construction of the verification condition is elaborated in [5]. Following is a brief description of the structure of the verification condition, which, if proven correct, guarantees the correctness of the translation.

In the following, we refer to the DC+ program as the *abstract* system, as it represents the specification, and to the C program as the *concrete* system, as it represents the concrete implementation. We denote the variables, initial condition and transition relation of these systems by V_A, θ_A and ρ_A (abstract) and V_C, θ_C and ρ_C (concrete), respectively. In order to establish that the concrete system correctly implements (*refines)* the abstract system, we use two premises (verification conditions), **R1** – the *base case*, and **R2** – the *induction step.*

The base case requires that θ_C implies θ_A, after performing an appropriate substitution α of each (observable) variable $v \in V_A$ by an expression ε over V_C. Such a substitution induces an (abstraction) mapping between the states of the two systems.

The induction step requires that ρ_C implies ρ_A, once again, after an appropriate substitution α.
Taken all together, the refinement rule has the following structure:

Let α: $V_A \rightarrow \varepsilon(V_c)$ be a substitution

R1: $\theta_c \rightarrow \theta_A[\alpha]$	The base step
R2: $\rho_c \rightarrow \rho_A[\alpha]$	The induction step

C **imp** A

The Verification Condition Generator, which is the first module invoked in CVT, generates these implications from the C and DC+ source codes. ρ_A and ρ_c are both large conjunctions of atomic sub-formulas, where typically (but not always) each sub-formula corresponds to an assignment line in the code or a constraint imposed by the abstraction (see section 4 for more details). These sub-formulas reflect the semantics of the source languages and the mapping between their variables.

4 *The CVT - Architecture*

The code-validation package offers a fully automatic routine which establishes the correctness of the generated code individually for each run of the code generator. Therefore, there is no user-interface to this tool – just configuration parameters and a command line.

The overall architecture of CVT can be seen in Figure 2. We will not focus in this (short) paper on the underlying theory behind the verification condition. As mentioned before, a detailed explanation of this can be found in [5]. Rather we explain what is the role of each module and what are its inputs and outputs.

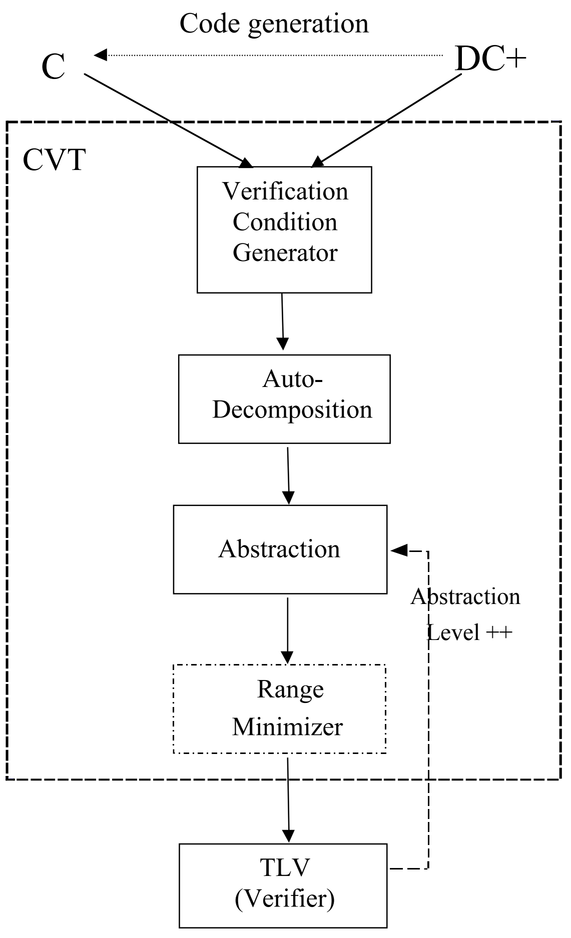


Fig. 2 : CVT Architecture (The 'Range Minimizer' module is not yet part of CVT)

4.1 The Verification Condition Generator module

CVT receives as input the DC+ and C source codes. These are the source and target code for the code generator. Two separate sub-modules (appearing united in Figure 1 as the Verification Condition Generator module) generate the verification conditions (which are actually a large logical implication) by means of various translations and transformations. The validity of this logical implication implies the correctness of the generated code w.r.t. the source code while its invalidity indicates a potential mistake in the code generation process. Each of the conditions is separated into two files representing the left and right hand side of the implication (in **R2** these are ρ_c and ρ_A) . Since at the end of this process we use TLV [6] as a decision procedure, the verification condition is generated in the appropriate format (the models TLV expects are compatible with the more broadly used SMV model-checker).

4.2 The Auto-Decomposition module

The next step is Auto-Decomposition. We are interested in handling industrial-size programs, and therefore decomposition is essential. As will be demonstrated in section 5, the auto-decomposition is one of the key enabling steps for scalability. The Auto-Decomposition module takes advantage of the fact that the right hand side of the premises are in the form of a conjunction (typically of hundreds of expressions), and simply breaks it into smaller conjuncts which can be verified independently.

The size of the decomposed conjuncts is set by a configuration parameter (called the 'chunk size'), and can range from 1 (a single conjunct) to the total number of conjuncts. In the later case the entire formula will be verified at once, which is only possible for relatively small files. After breaking the right hand side, the Auto-Decomposition module returns to the left hand side of the implication, and calculates the *cone of influence,* i.e. the portion of the formula in the left hand side that is needed for proving the selected conjuncts on the right-hand side. After repeating this process until all conjuncts are covered, we are left with (possibly hundreds) pairs of files, each significantly smaller than the original ones. There is an obvious tradeoff between having files with very small right hand side, which leads to significantly shorter verification time, and the number of these files which incurs an additional invocation overhead cost associated with each file. It is therefore left to the user to decide on the chunk size which may be optimal for her case.

Another configuration option in the Auto-Decomposition module is called 'back calculation'. When this flag is set, after calculating the cone of influence, the program returns to the right hand side and looks for additional conjuncts that can be proven with the same cone that was just calculated. This option is useful for reducing the number of files and reducing the over-all time for performing the proof (the time TLV takes mainly depends on the transition relation of the model, i.e. the left hand side. Thus if we use the same model for proving more conjuncts, we save time). When setting this option, the 'chunk size' is no longer an exact number of conjuncts taken each time, rather it is the size of the initial set of conjuncts, which possibly grows after the back calculation. The efficiency of the back calculation obviously depends on the ordering of conjuncts we are investigating. An optimal ordering would be such that if $\mathtt{cone(C_i)} \subseteq \mathtt{cone\ (C_j)}$ then $\mathtt{C_i}$ and $\mathtt{C_j}$ are verified together (with simple sequential ordering this will happen only if $\mathtt{C_j}$ appears first or if $\mathtt{cone(C_i)\ =\ cone\ (C_j)}$). This ordering can be achieved, for example, by calculating all the cones and then partitioning the files accordingly. We did not implement this because we suspect that the overhead of this calculation will be larger than the saving resulting from the better ordering.

4.3 The Abstraction Module and the Range-Minimizer module

After decomposing the files, CVT invokes the Abstraction module. Once again, the underlying theory of the abstraction is detailed in [5]. Basically, abstraction is needed since we are trying to verify a model which contains integer and float variables, as well as functions over these variables using a BDD-Based decision procedure for finite-state models. The abstraction module treats these functions as *uninterpreted functions,* replacing them by new symbols. The faithfulness of this technique depends on the way that the compiler manipulates these functions and the kind of functions we leave interpreted. The more we interpret, the more faithful the model is. On the other hand, the less we interpret, the smaller the model is.

The abstraction module works in an incremental manner, following an *abstraction hierarchy* designed according to the specific optimizations the compiler performs. We begin with maximum abstraction (called *Level-0 abstraction*) where all functions except equalities, Boolean operators and if-then-else are uninterpreted. If the proof fails, CVT invokes the abstraction module again, asking for *Level-1 abstraction*, where additionally comparisons operators on integers ('>', '<', etc) are left interpreted.

If, for example, the compiler reads 'a < b' in the abstract system and transforms it to 'b > a' in the concrete system (which are obviously semantically equivalent) , Level-0 abstraction will result in a false negative where as level-1 will succeed.

The reason we first interpreted the comparison operators on moving from level-0 to level-1 is that the compiler we are considering employs these kinds of optimizations frequently. To handle comutativity of the '+' function, for example, we need another abstraction level. However, so-far Level-0 and Level-1 abstractions proved to be sufficient for the purposes of code-validation of the examples we have considered.

After we replace the appropriate terms by new variables, we impose additional constraints on the verification conditions to ensure functionality. This leaves us with a quantifier-free first-order logic formula which enjoys the small model property (i.e. it is satisfiable iff it is satisfiable over a finite domain). Therefore the next issue the abstraction module handles is the calculation of a finite domain, such that the formula is valid iff it is valid over all interpretations into these domains. The latter can be checked algorithmically, using BDD techniques. The domain that is currently taken is simply a finite set of integers whose size is the number of (originally) integer/float variables, although smaller domains can be achieved by analyzing the structure of the formula considered. This analysis is performed by the 'Range Minimizer' module, not yet implemented, which we expect to significantly reduce the range of each of these (now enumerated type) variables, and thus increase the size of programs we can handle. Preliminary results from this approach show that most models can be verified by using a state-space which is orders of magnitude smaller than the state-space resulting from using our current method.

4.4 The Verifier module (TLV)

The validity of the verification conditions is checked in TLV [6], an SMV-based tool which provides the capability of BDD-programming and has been developed mainly for finite-state deductive proofs (and thus convenient in our case for expressing the refinement rule). In the case that the equivalence proof fails, a counter example is displayed. Since it is possible to isolate the conjunct(s) that failed the proof, this information can be used by the compiler developer to check what went wrong. CVT invokes TLV for each pair of files generated by the Auto-Composition module. A proof log is generated as part of this process, indicating which files were proved, at what level of abstraction and when.

5 A case study

Currently we are working on the validation of an industrial size program, a code generated for the case study of a turbine developed by SNECMA, which is one of the industrial case studies in the SACRES project. The program was partitioned manually (by SNECMA) into 5 units which were separately compiled. Altogether the DC+ specification is a few thousand lines long and contains more than 1000 variables. After the abstraction we had about 2000 variables (as explained in subsection 4.3, the abstraction module replaces function symbols with new variables). Following is a summary of the results achieved so far:

Module	Conjuncts	Verified	Time (min.)
M1	530	100%	4:14
M2	533	100%	1:30
M3	124	92%	?
M4	308	99.3%	3:32 + ?
M5	860	80%	?
Total :	**2355**	**93.9%**	**9:16 + ?**

As can be seen, about 6.1% of the conjuncts in our case could not be verified in reasonable time using the current implementation of CVT. We hope that after installing the Range-Minimizer this problem will be solved.

References

[1] B. Buth, K. Buth, M. Franzle, B. Karger, Y. Lakhneche, H. Langmaack, and M. Muller-Olm. Provably correct compiler development and implementation. In *Compiler Construction '92*, 1992.

[2] D.L. clutterbuck and B.A. Carre. The verification of low-level code. *Software Engineering Journal*, pages 97-111, 1998.

[3] P. Curzon. A verified compiler for a structured assembly language. In *proceedings of the 1991 international workshop on the HOL theorem Proving System and its applications*. IEEE Computer Society Press, 1992.

[4] I. M. O'Neill, D. L. Clutterbuck, P.F. Farrow. The formal verification of safety-critical assembly code. In *proceedings of the IFAC Symposium on safety of computer control systems* 1988.

[5] A. Pnueli, O. Shtrichman and M. Siegel. Translation Validation for Synchronous Languages. To appear in *ICALP' 98*.

[6] A. Pnueli and E. Shahar, A Platform for Combining Deductive with Algorithmic Verification. *8th Conference on Computer Aided Verification*, Springer-Verlag, 1996

[7] The SACRES web page: http://www.ilogix.co.uk/ilogix/technica.html

Mechanized Verification of Compiler Backends

Axel Dold[1], Thilo Gaul[2], and Wolf Zimmermann[2]

[1] University of Ulm, Oberer Eselsberg, D-89069 Ulm,
{dold}@ki.informatik.uni-ulm.de
[2] University of Karlsruhe, Zirkel 2, D-76131 Karlsruhe,
{gaul;zimmer}@ipd.info.uni-karlsruhe.de

Abstract We describe an approach to mechanically prove the correctness of BURS specifications and show how such a tool can be connected with BURS based back-end generators [9]. The proofs are based on the operational semantics of both source and target system languages specified by means of Abstract State Machines [15]. In [31] we decomposed the correctness condition based on these operational semantics into local correctness conditions for each BURS rule and showed that these local correctness conditions can be proven independently. The specification and verification system PVS is used to mechanicalyy verify BURS-rules based on formal representations of the languages involved. In particular, we have defined PVS proof strategies which enable an automatic verification of the rules. Using PVS, several erroneous rules have been found. Moreover, from failed proof attempts we were able to correct them.

1 Introduction

There exist a variety of techniques to construct efficient code producing compiler back-ends with sufficient tool support. There do also exist approaches on the construction of verified code generators but they do not fullfill at least one of the following requirements that are essential to the code generation part of pratical compiler environments: (i) compilation to native machine code, (ii) ability to deal with complex real-life programming languages, (iii) to produce efficient machine code, comparable to non-verified compilers and (iv) tool support for the compiler writer.
Usually the correctness proofs of programs or program derivations assume that the programs are written in higher-level languages. However, the program is compiled into binary code and it is this code, that is executed. Therefore the correctness of programs depends also on the correctness of the compiler which compiles the higher-level language into the machine language of the processor, and on the correctness of the processor. This paper discusses aspects for the construction of realistic correct compilers.
Realistic correct compilers should produce machine code whose performance is comparable to machine code produced by usual compilers. Practical experiences show that the main performance gains and losses result from the back-end of compilers. Therefore, we focus on the construction of correct compiler back-ends. Compiler back-ends transform low-level intermediate language programs into machine programs.
One of the well known techniques to produce efficient machine code are bottom up rewrite systems (BURS). This specification technique [9, 10, 22, 23, 27] has two advantages: (i) the back-end can be generated from such a specification, (ii) it is possible to specify back-ends which produce efficient code.
We show in this paper how rewrite rules for code generation can be pratically proven correct, thus the BURS technology becomes applicable to the construction of verified compilers.
In [31] we decomposed the correctness of BURS-based compiler back-ends into the local correctness of single term-rewrite rules, and the global correctness. Furthermore we proved the global correctness under the same constraints as the applicability of the BURS-technology, i.e. no specific assumptions on the term-rewrite system are required. In this paper we put the main stress on the local correctness of single rewrite rules.

In practice BURS specifications have a large number of TRS rules. Therefore a mechanical verification of the local correctness is necessary. This paper shows proof strategies sufficient to prove local correctness of TRS rules and its mechanization using PVS [24]. Our approach is to give operational semantic specifications for source and target language, and to prove the correctness of single transformations by symbolic execution of the program pieces. The semantic and transformation rule specifications have been formalized into PVS. Together with PVS proof strategies we are able to automatically verifiy the rules. Using these strategies, errorneous rules have been found, and, moreover, from failed proof attempts these errors could be corrected by a careful inspection of the proof state.
We demonstrate our approach with a typical basic block oriented intermediate language with expressions (**MIS**) and the translation to the DEC-Alpha processor family. The operational semantics is formalized by abstract state machines (Section 2). In this paper, the mapping of composite datatypes such as records, arrays etc. is considered as a front-end task.
The first work on correct compilers is [17]. Most of the following work on correct compilation is based on denotational semantics (e.g.[4, 18, 19, 25, 26, 30]) or on refinement calculi (e.g.[5, 6, 16, 20, 21]). Other work on compiler correctness based on refinement use abstract state machine (e.g.[1–3]). Most of these works do not compile high-level programming languages into assembler languages. To our knowledge, only [2, 3, 20, 21] and ProCos [16] discus transformations into machine code. The performance of code generated by semantics driven code generation is poor and by one or more orders of magnitude slower than the code generated by compilers used in practice [25]. [2, 3, 20, 21] consider the compilation into transputer code. We compare our measerument results with those from [7]. To our knowledge the only work on "correct compilers" which provides performance measurements.
This article is organized as follows: First we give the basic definitions of abstract state machines and correctness notion in Section 2.
Our definition of BURS specifications (Section 3) define the local correctness conditions and central theorem of [31]. Section 4 gives the general proof strategy for proving local correctness and demonstrates it by some examples. Section 5 describes the formalization of the intermediate and target language ASM's, the formal representation of the TRS rules and their verification using proof strategies. In particular, the detection and correction of an erroneous rule using PVS is presented.

2 Basic Definitions

The semantics of our languages are defined operationally by abstract state machines [14]. The notation is oriented on [15], and can be found in following subsection.
In Appendix A we define a typical intermediate language with expressions (**MIS**), which is based on basic block graphs (see e.g. [29]). Our example target language is the machine language of the DEC-Alpha processor family (Appendix B). The complete definitions can be found in [12] and [13].

2.1 Abstract State Machines

An abstract state machine is a tuple $\mathcal{A} = (\Sigma, Q, S, \rightarrow, I)$, where

(i) Σ is a *signature*, i.e. a finite collection of function names, each of a fixed arity.
(ii) the set of *states* Q is a set of Σ-algebras,
(iii) a superuniverse S representing the sorts,
(iv) $\rightarrow \subset Q \times Q$ is the *transition relation*, and
(v) $I \subset Q$ is the set of *initial states.*

f_q denotes the interpretation of $f \in \Sigma$ in state $q \in Q$. Interpretations on S of function names in Σ are called *basic functions.* The superuniverse does not change when the state of

$\mathcal{A}$ changes, the basic functions may. The superuniverse contains distinct elements *true*, *false* and *undef* ($\perp$) that allow to deal with binary relations and partial functions. They do not appear in the signature.
A *universe* U is a special type of basic function: a unary relation identified with the set $\{x : U(x)\}$. Any sort $U \in S$ denotes a universe. The universe $BOOL$ is defined as $BOOL = \{true, false\}$. A function $f : U \to V$ from an universe U to an universe V is an unary operation on the superuniverse such that $f(a) \in V$ for all $a \in U$ and $f(a) = \perp$ otherwise. The type INT is also used to denote the 64-bit integer arithmetic of the DEC-Alpha Processor. $INT[k]$ denotes the integer represented by k Bits, i.e. the range $-2^k, \ldots, 2^k - 1$. Floating point types are not used in this article.
A term over the signatur Σ is defined as usual. $\mathcal{T}(\Sigma)$ denotes the set of terms over the signature Σ. The interpretation of a term $t \in \mathcal{T}(\Sigma)$ in state q is denoted by $[\![t]\!]_q$. Sometimes we need a set of variables V. $\mathcal{T}(\Sigma, V)$ denotes the set of term over signatur Σ and variables V. As usual, $\overset{*}{\to}$ denotes the reflexive, transitive closure and $\overset{+}{\to}$ denotes the transitive closure of $\to$. A state $q \in Q$ is *reachable* iff there is an initial state $i \in I$ such that $i \overset{*}{\to} q$. The relation $\to$ is defined by a finite collection of transition rules of the form:
if Condition **then** Updates **endif**
For example
if t_0 **then** $f(t_1, \ldots, t_n) := t_{n+1}$ **endif**
where $t_0, t_1, \ldots t_{n+1} \in \mathcal{T}(\Sigma)$ is a transition rule. Let q be a state before and q' be a state after applying the rule. The meaning of the rule is:If $[\![t_0]\!]_q = true$ then for all $g \in \Sigma \setminus f$ $g_{q'} = g_q$, and $f_{q'}$ is defined as follows:

$$f_{q'}(x_1, \ldots, x_n) = \begin{cases} [\![t_{n+1}]\!]_q & \text{if for all } i,\ 1 \le i \le n,\ [\![t_i]\!]_q = x_i \\ f_q(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

If $[\![t_0]\!]_q = false$ then $f_q = f_{q'}$ for any $f \in \Sigma$. Thus the interpretation changes the value of the basic function f at the value of the tuple $(t_1, \ldots, t_n)$ to the value t_{n+1}, provided that $[\![t_0]\!]_q = true$. If several updates contradict then one update is chosen nondeterministically.
We distinguish the following classes of functions:

- *dynamic functions*: the interpretation of a dynamic function is changed by transition rules, i.e. f is called a dynamic function if an assignment of the form $f(t_1, \ldots, t_n) := t_{n+1}$ appears anywhere in a transition rule.
- *static functions*: the interpretation of a static function is never changed by a transition rule.

Macros are abbreviations and denoted by $macro \hat{=}\ expr$. Whenever $macro$ is used, it is replaced by $expr$. We assume that macro definitions define a noetherian and confluent rewrite system.

2.2 Correctness Notion

The definition of compiler correctness is based on the operational semantics of the source and target languages, respectively. An operational semantics for a language L defines a family of abstract state machines $\mathcal{A}_\pi$ for every program $\pi \in L$. The state transitions are based on the instruction set of L (cf. Appendix A). Not all state transitions are relevant. State transitions that correspond to jumps, memory access, procedure calls etc. must be distinguished from state transitions that read an input and write an output. The former are not observable while the latter are observable. We therefore distinguish *observable behaviour*. Formally, a *behaviour of program* π is a set of finite or infinite sequences of states $q_0 \to q_1 \to \cdots \to q_n$, where $q_i \to q_{i+1}$ is the transition from state q_i into state q_{i+1} by the ASM of the program. The observable behaviour focuses on the input/output. Figure 1 illustrates the observable

behaviour by merging all states where the following state transition does not change input or output. [31] defines these definitions formally.
A *compiler* which compiles a program $\pi_1 \in L_1$ into a program $\pi_2 \in L_2$ implements a relation $\mathcal{C} : L_1 \times L_2$. Intuitively, $\mathcal{C}$ is correct if π_1 and π_2 have the same observable behavior. We base the correctness definition on simulations, i.e. $\mathcal{A}_{\pi_2}$ simulates $\mathcal{A}_{\pi_1}$ in a sense similar to the notion of simulations used in complexity and computability theory. Figure 1 illustrates these ideas. Full definitions can be found in [31].
The relation $\bar{\rho}$ maps injectively the observable part of the states of the target program to the observable part of the states of the source program. For every observable behaviour of the target program, there must be a corresponding observable behaviour of the source program. $\bar{\rho}$ can be implemented by a relation ρ which is compatible with $\bar{\rho}$, i.e., any states can be related by ρ whose observable part is related by $\bar{\rho}$. ρ is usually implemented by a compiler. The freedom of choice of ρ allows optimizations. In particular, the order of instructions can be changed as long as the observable behaviour is preserved.

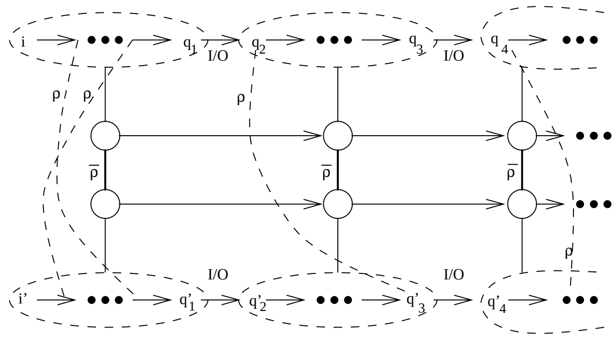


Figure1. Compiler Correctness

2.3 Code Generation by Term Rewriting

Term rewriting is commonly used in compiler back-end generators for the specification of the transformation to be performed by the code selection, i.e. the mapping $\mathcal{CS} : IL \Rightarrow IL'$. Any intermediate language command can be viewed as a term. The basic idea is to reduce this term to a constant and to generate code for each applied reduction step. The sub-class of rewrite systems used in this article are the Bottom-Up-Rewrite-Systems, the commonly used technique for code generators.
A term-rewrite rule $t \Rightarrow X; \{m_1, \ldots, m_n\}$ specifies a compilation in the sense that sequence $m_1, \ldots, m_n$ is the sequence of machine instructions that implement t. The machine instructions $m_1, \ldots, m_n$ may contain non-terminals as well as registers.
This application is more special than for general term rewriting systems. A *BE-substitution* σ is just a renaming of non-terminals to registers. A term t_1 *BE-matches* a term t_2 if there is a BE-substitution σ such that $\sigma(t_1) = t_2$, i.e. t_1 and t_2 are equal up to renaming of non-terminals. This allows to use any optimizing register allocation algorithm that fullfills the constraint, that no value containing register is reassigned. A full formal definition can again be found in [31].

3 Correctness of TRS-based code generation

In [31], we reduced the correctness condition for BURS-based compiler back-ends to the following two local correctness conditions:

(i) A term-rewrite rule $t \Rightarrow X; \{m_1, \ldots, m_n\}$ is *locally correct* iff for every state q there are states $q_1, \ldots, q_n$ such that

$$q \overset{m_1}{\to} q_1 \overset{m_2}{\to} q_2 \cdots \overset{m_n}{\to} q_n$$

and $[\![eval(t)]\!]_q = [\![content(\sigma(X))]\!]_{q_n}$ where $\sigma(X)$ is the register assigned to X.

(ii) A term-rewrite rule $t \Rightarrow \bullet; \{m_1, \ldots, m_n\}$ is *locally correct* iff for every state q there are states $q_1, \ldots, q_n$ such that

$$q \overset{m_1}{\to} q_1 \overset{m_2}{\to} q_2 \cdots \overset{m_1}{\to} q_n$$

and $content_{q_n} = content_{q'}$, $content_{q_n}(R) = content_q(R)$ for all locked registers, and $PC_{q_n} = PC_{q'}$ where q' is the state such that $q \overset{t}{\to} q'$.

Remark 1. In [31] we showed that it is possible to express the operational semantics of the intermediate language with the state space of the processor (except instruction pointers). This includes the memory mapping of the intermediate language onto the machine memory and storing environment pointers in certain registers. These registers are *locked.* Clever register assignment algorithms may lock more registers. The set of locked registers may be different for different program points. The operational semantics of the intermediate language using the state space of the processors allows to use machine-instructions as well as intermediate language instructions. Therefore, the state space of the states used in the local correctness conditions (i) and (ii) is equal. For the rest of this paper, we always assume this equality of state spaces.

The basic result we use in this paper is the

Theorem 1. *Let $\mathcal{S}$ be a BURS-specification that specifies the transformation of basic-block graphs into machine code of a register machine. If every term-rewrite rule $r : t \Rightarrow X; \{m_1, \ldots, m_n\} \in \mathcal{S}$ is applied to expression trees only if no register occuring outside of t is assigned to the free non-terminals of r, no locked register is assigned to the free non-terminals of r and every term-rewrite rule satisfies (i) and (ii), then the BURS-specification specifies a correct compilation.*

Thus, if a correct back-end generator applies the TR-rules conditionally (with the requirements on register assignment), and (i) and (ii) is satisfied for each TR-rule of a BURS-specification, then a correct compiler back-end is generated.

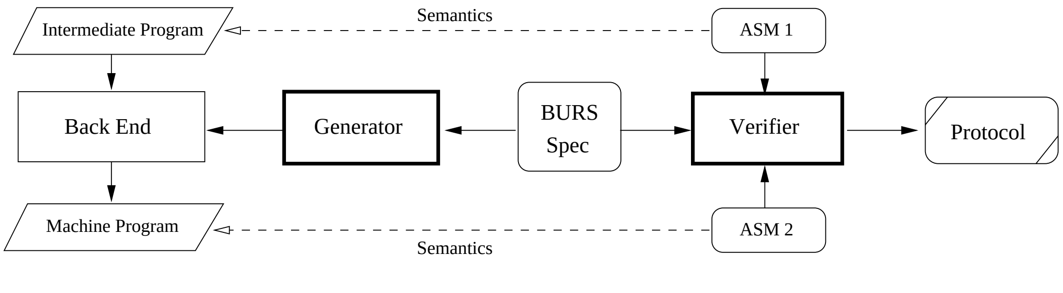


Figure2. Verified Back-End Generator Architecture

4 Proof Strategies for Local Correctness

This section consists of three parts. First we give two proof strategies related to local correctness conditions (i), and (ii). Then we show two examples applying these two proof strategies, respectively. Finally, we show that the application of these proof strategies to an erroneous term-rewrite rule.
Consider a term-rewrite rule $r : t \Rightarrow X; \{m_1, \ldots, m_n\}$ the local correctness condition (i). It specifies a finite execution of ASM-transition rules and the application of an *eval*-macro. Since the latter must define a noetherian and confluent rewrite system, the number of macro applications is also finite. Therefore the proof strategy for (i) is

1. Apply symbolically (using the non-terminals occuring in t) the *eval*-macro,
2. Apply symbolically (using the non-terminals occuring in t and $\{m_1, \ldots, m_n\}$) the state transitions of the ASM for the target machine
3. If $\mathit{content}(X) = \mathit{eval}(t)$ then (i) is satisfied. Otherwise, output both terms.

Theorem 2. *The above proof strategy is correct for term-rewrite rules* $r : t \Rightarrow X; \{m_1, \ldots, m_n\}$ *where* $X \neq \bullet$.

Proof. In any program, if r is applied onto a term t', then there is a register substitution σ such that $\sigma(t) = t'$ (i.e. σ substitutes non-terminals by registers). Consider a symbolic update $f(X_1, \ldots, X_n) := g(Y_1, \ldots, Y_n)$ performed in a state transition of the second step. Then the execution of the program performs the update $f(\sigma(X_1), \ldots, \sigma(X_n)) := g(\sigma(Y_1), \ldots, \sigma(Y_n))$. A similar argument applies to the application of the *eval*-macro. Thus, it holds $[\![\mathit{content}(\sigma(X))]\!]_{q_n} = [\![\mathit{eval}(\sigma(t))]\!]_q$. Therefore, the symbolic equality in the third step implies $[\![\mathit{content}(\sigma(X))]\!]_{q_n} = [\![\mathit{eval}(\sigma(t))]\!]_q$, i.e. (i) holds.

Example 1. *Consider the term-rewrite rule add* : $\mathit{intadd}(X, Y) \Rightarrow Z$; $\{\mathit{ADD}\ X, Y, Z\}$ *where ADD is a DEC-Alpha instruction. The first step yields*

$$\begin{aligned}\mathit{eval}(\mathit{intadd}(X, Y)) &= \mathit{eval}(X) \oplus_I \mathit{eval}(Y)\\ &= \mathit{content}(X) \oplus_I \mathit{content}(Y)\end{aligned}$$

The second step applies the state transition $\mathit{content}(Z) := \mathit{content}(X) \oplus_I \mathit{content}(Y)$ *of the ADD-instruction. Thus, after the symbolic execution of ADD* X, Y, Z, *it is* $\mathit{content}(Z) = \mathit{content}(X) \oplus_I \mathit{content}(Y)$. *Thus, the two terms to be compared in the third step are equal and the term-rewrite rule add is locally correct.*

Thus, if the symbolic equality cannot be proven, the strategy postulates an equality that must always be satisfied in order to ensure (i). Thus, without algebraic identities such as the commutativity of $\oplus_I$, the proof strategy would also be complete: Suppose, that the strategy outputs that (i) is not satisfied. Then, we have to show that there is a state q, a term t' where r can be applied, and a valid register assignment such that (i) is violated, i.e. $[\![\mathit{eval}(t')]\!]_q \neq [\![\mathit{content}(\sigma(X))]\!]_q$. Suppose steps 1 and 2 show that $\mathit{eval}(t) \neq \mathit{content}(X)$. Observe that $\mathit{content}(X) = s$ using the non-terminals occuring in t. Let σ be the substitution used when applying r such that $\sigma t \neq \sigma s$. Then it is easy that there are values of the registers such that these two expressions yield different results, i.e. (i) is violated. If $\sigma t = \sigma s$ for all substitutions, then it is must be $s = t$, contradicting our assumption $s \neq t$.
Consider now the term-rewrite rules of the form $t \Rightarrow \bullet; \{m_1, \ldots, m_n\}$. Then, the following proof strategy is used:

1. Execute symbolically (using the non-terminals in t) the state transitions for t in the intermediate language ASM.
2. Execute symbolically $\{m_1, \ldots, m_n\}$ in the machine language ASM.
3. For all non-terminals of t, compare symbolically $\mathit{content}(X)$ after the first and the second step. Also compare $\mathit{content}(R)$ for locked registers. If all of them are equal, then (ii) is satisfied. Otherwise, output the inequalities.

Theorem 3. *The above proof strategy is correct for term-rewrite rules* $r : t \Rightarrow \bullet; \{m_1, \ldots, m_n\}$.

Proof. (Sketch) The proof is analogous to the proof of Theorem 2. The only difference is that more comparisons have to be done, and that for every states q, q_n, and q' as defined by local correctness condition (ii) the instruction pointers point to the same instruction if just r is applied. Let π be a program and π' be the program where r is applied onto a term t'. Since r defines a local replacement, the instruction after t' in π and the instruction after m_n in π' is the same.

To show a more complex proof, we give an example with a bit more complicated bit manipulation. Consider the generation of a 32 bit constant:

Example 2.

$$\begin{aligned} \mathit{intconst}(i32) \longrightarrow X;\ \{\ & \mathit{LDA}\ (T1, i32.L, R31, \text{“}L\text{“}) \\ & \mathit{ZBI}\ (T1, \#11111100_2, T1) \\ & \mathit{LDA}\ (X, i32.H, T1, \text{“}H\text{“})\ \} \end{aligned}$$

The coding of 32 bit integer constants uses the machine instructions “load-address” (LDA) and “zero-bytes-immediate” (ZBI). LDA loads the integer value that results from the addition of the second operand and the immediate value, which is shifted by 16 according to the type parameter, into the first operand. “zero-bytes” sets some bit patterns to zero according to the immediate value.
For the purposes of the simulation proof we need some definitions for the integer constant:

i32	$= \mathrm{hhll}_{256}$	That means we have:		
s_l	$= (\mathit{i32})\langle \mathit{15} \rangle$	i32.L	$= \mathrm{00ll}_{256}$	
s_h	$= (\mathit{i32})\langle \mathit{31} \rangle$	i32.H	$= \mathrm{00hh}_{256}$	
i32.L	$= (\mathit{i32})\langle \mathit{0} : \mathit{15} \rangle$	Sext_{16} (i32.L)	$= \mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{ll}_{256}$	
i32.H	$= (\mathit{i32})\langle \mathit{16} : \mathit{31} \rangle$	Sext_{16} (i32.H)	$= \mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{hh}_{256}$	

“i32.L” denotes the lower 16 bit of the lower word (value “ll”), “i32.H” denotes the upper 16 bit of the lower word (value “hh”). Sext_k(Y) extends the kth bit of Y to 64 bit. $X\langle \mathrm{i} \rangle$ addresses the ith bit of X.
The proof obligations can be formulated in the same way like before. The proof itself is more tedious but is nevertheless a straightforward simulation:

rule application	T1	X	state

LDA T1, i32.L, R31, “L”
content(T1) := content(R31) $\oplus_I$ Sext_{16} (i32.L)
content(T1) := 0 $\oplus_I$ Sext_{16} (i32.L)
content(T1) := Sext_{16} (i32.L)
	$\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{s}_l\mathrm{ll}_{256}$		q_1'

ZBI T1, #11111100$_2$, T1 content(T1) := ByteZap (content(T1), #11111100$_2$) $\mathrm{content(T1)}\langle \mathrm{i} \rangle := \begin{cases} \mathit{content}(T1)\langle \mathrm{i} \rangle \text{ case } 0 \le i \le 15 \\ 0_2 \text{ otherwise} \end{cases}$	$\mathrm{000000ll}_{256}$		q_3'

LDA X, i32.H, T1, “H”
content(X) := content(T1) $\oplus_I$ LogShift$_L$(Sext_{16} (i32.H), 16)
content(X) := content(T1) $\oplus_I$ $\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{hh00}_{256}$
content(X)
		$\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{s}_h\mathrm{hhll}_{256}$	q_3'

The last line of the simulation table proves our assumption correct. T1 and X can contain any preassigned value.

5 Implementation of Proof Strategies and Error Detection in PVS

Beneath careless mistakes in combining instructions or operand ordering, bit-manipulations in sequences of instructions are the most erroneous kind of code. An example taken from our own specification development cycle is the rewrite rule for large integer constants, where happened to be some nontrivial bit-manipulation errors.
As an example, how PVS can help detecting errors we present the incorrect rule for large integer constants which has been used in a first attempt:

$$\begin{aligned} \mathit{intconst}(i32) \longrightarrow X;\ \{\ & \mathit{LDA}\ (T1, i32.L, R31, \text{“}L\text{“}) \\ & \mathit{LDA}\ (X, i32.H, T1, \text{“}H\text{“})\ \} \end{aligned}$$

The formalization in PVS looks like:

```
medium_int_const : LEMMA (r0 /= r3) IMPLIES
 bv2int[32](eval(intconstS(val32), rho(ms))) =
  LET final = bb_interp(
  cons(store(LDA(0), r3, 31, val32^(15,0)),
  cons(store(LDA(1), r0, r3, val32^(31,16)), null))), ms)
 IN bv2int[64](IntReg(final)(r0))
```

This rule does incorrectly treat sign extensions if the most-significant-bit (MSB) of the lower word is set. Invoking strategy `(simul)`, the prover stops in the following situation:

```
 (val!1  =
     plus_Q((bitcopy((val!1 ^ (31, 16))(15))(32)
                o val!1 ^ (31, 16))
                o bvec0[16],
            (bitcopy((val!1 ^ (15, 0))(15))(48)
                o val!1 ^ (15, 0))))
```

A more compact notation for this equality is

$$\mathit{val} = \ \mathit{val}(31,16) \circ 0_{16} \ \oplus_I \ (s_{15})_{16} \circ \mathit{val}(15,0)$$

(“$\circ$” denotes bit sequence concatenation). Obviously, this equality holds only if the MSB of the lower word is 0, i.e.

$$s_{15} = 0.$$

The example given in the previous section contains the correct implementation where this erroneous sign extension is corrected with an ZBI (Zero Bytes) instruction.
In addition, some more proof obligations have to be discharged. For example, one has to prove that the instruction sequences above are valid basic blocks. When type-checking the theory, PVS automatically generates such a type correctness condition (TCC). The proof is easily established using the built-in `(grind)` strategy.
Sometimes, it is useful to have the possibility to step through each machine instruction interactively. If one likes to see the effect of each instruction separately, one may use strategy `(one-stp)`. First, the necessary rewrite rules have to be established using `(init)`, then each execution of an instruction is invoked using `(one-stp)` which expands the definition of the basic block interpreter and then applies the rewrite rules and decision procedures.

6 Conclusions

In this article we showed how to prove the local rewrite rules for language transformations correct with a symbolic simulation proof technique. The task is no longer to find a proof magically but we gave an constructive approach for proving rewrite rules. The correctness is proved w.r.t. operational semantics of both, intermediate and target language. The tool performs the proofs strategies mechanically using the proof checker PVS. Thus, the integration of such a prover with BURS back-end generators can guarantee – together with the correctness of a generator – the correctness of a compiler back-end. Moreover we showed that useful hints to errors in TRS-rules can be given. First results indicate that the quality of the binary machine code generated by correct compiler back-ends described in this paper is comparable to

		DEC-Alpha				Intel-Pentium		SIMP
		Verifix		C-Compiler		C-Compiler		AM in C
	Iterations	non-opt	opt	non-opt	opt	non-opt	opt	min
Loop	10000	0.57ms	0.57ms	0.35ms	0.31ms	0.62ms	0.50ms	5.0s
	100M	5.72s	5.70s	3.49s	3.05s	6.12s	5.04s	13h53m*
Sieve	1	1.63ms	1.23ms	0.82ms	0.56ms	1.02ms	0.89ms	4.00s
	10000	16.35s	12.26s	8.25s	5.65s	10.23s	8.94s	11h6m*

DEC-Alpha:	DEC-AXP(233MHz), OSF1, CC: DEC(V4.2)
Intel-Pentium:	Pentium(133MHz), Linux, CC: GNU(V2.7.0)
SIMP:	Pentium, execution times taken from [7], abstract machine implemented in C
Iterations:	Loop: loop iterations, Sieve: searching the primes less than thousand, n times repeated
	SIMP: line 1 from [7], line 2 extrapolotion(*) on repeated iterations
Optimization (opt):	Verifix: Peephole, C: Option -O4, SIMP: minimal execution times

Table1. Comparison of the Performance of the Machine Code generated by Correct Compilers

standard compilers and therefore orders of magnitudes faster than code generated by correct compilers constructed by other approaches [25, 7]. Table 1 shows the comparison between our approach, the approach in [7] (SIMP), and a standard unverified *C*-compiler. Loop is a program that initializes a variable with a positive integer and decrements this integer by one until the content of this variable is zero, Sieve implements the sieve of Erastothenes. The complete test scenario can be found in [11].
Our aim is to integrate more and more established compilation techniques. If a library of correct data structures, algorithms, and generators is provided, then for the correctness of any transformation of one intermediate language to another, it will be sufficient to prove local correctness properties of transformation rules similar to those of term-rewrite rules.
The above performance results show that this approach seems feasible to construct realistic correct compilers compiling programs of real-life programming languages into binary machine code of real processors, and produce efficient code.
Acknowledgements: This work is supported by the *Deutsche Forschungsgemeinschaft* project Go 323/3-1 *Verifix* (Construction of Correct Compilers). We are grateful to our collegues in *Verifix*.

References

1. E. Börger and D. Rosenzweig. The WAM-definition and Compiler Correctness. Technical Report TR-14/92, Dip. di informatica, Univ. Pisa, Italy, 1992.
2. Egon Börger, Igor Durdanovic, and Dean Rosenzweig. Occam: Specification and Compiler Correctness.Part I: The Primary Model. In U. Montanari and E.-R. Olderog, editors, *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi).* North-Holland, 1994.
3. E. Brger and I. Durdanovic. Correctness of compiling occam to transputer. *The Computer Journal*, 39(1):52–92, 1996.
4. D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *Compiler Compilers 92*, volume 641 of *Lecture Notes in Computer Science*, 1992.
5. B. Buth, K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
6. Bettina Buth and Markus Müller-Olm. Provably Correct Compiler Implementation. In *Tutorial Material – Formal Methods Europe '93*, pages 451–465, Denmark, April 1993. IFAD Odense Teknikum.
7. S. Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines.* PhD thesis, Universität Saarbrücken, 1996.
8. Axel Dold. Representing the Alpha Processor Family using PVS. Verifix Working Paper [Verifix / Uni Ulm / 4.1], Universität Ulm, November 1995.
9. H. Emmelmann. Code selection by regularly controled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, Workshops in Computing. Springer-Verlag, 1992.

10. H. Emmelmann, F.-W. Schröer, and R. Landwehr. Beg - a generator for efficient back ends. In *ACM Proceedings of the Sigplan Conference on Programming Language Design and Implementation*, June 1989.

11. T.S. Gaul. Bechmarking code-generation for IS to DEC-Alpha. Verifix Working Paper [Verifix/UKA/11], University of Karlsruhe, 1996.

12. T.S. Gaul, A. Heberle, and W. Zimmermann. An Evolving Algebra Specification of the Operational Semantics of MIS. Verifix Working Paper [Verifix/UKA/3], University of Karlsruhe, 1995.

13. T.S. Gaul and W. Zimmermann. An Evolving Algebra for the Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], University of Karlsruhe, 1995.

14. Y. Gurevich. Evolving Algebras; A Tutorial Introduction. *Bulletin EATCS*, 43:264–284, 1991.

15. Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

16. C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

17. J. McCarthy and J.A. Painter. Correctness of a compiler for arithmetical expressions. In J.T. Schwartz, editor, *Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.

18. P. D. Mosses. Abstract semantic algebras. In D. Bjørner, editor, *Formal description of programming concepts II*, pages 63–88. IFIP IC-2 Working Conference, North Holland, 1982.

19. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

20. Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.

21. Markus Müller-Olm. *Modular Compiler Verification*. PhD thesis, Techn. Fakultät der Christian-Albrechts-Universität, Kiel, June 1996. Erscheint als LNCS Band im Springer-Verlag.

22. Albert Nymeyer and Joost-Pieter Katoen. Code Generation based on formal BURS theory and heuristic search. Technical report inf 95-42, University of Twente, 1996.

23. Albert Nymeyer, Joost-Pieter Katoen, Ymte Westra, and Henk Alblas. Code Generation = A* + BURS. In Tibor Gyimothy, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science*, pages 160–176, Heidelberg, April 1996. Springer-Verlag.

24. S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *Proceedings 11th International Conference on Automated Deduction CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, October 1992. Springer-Verlag.

25. J. Palsberg. An automatically generated and provably correct compiler for a subset of ada. In *IEEE International Conference on Computer Languages*, 1992.

26. L. Paulson. *A compiler generator for semantic grammars*. PhD thesis, Stanford University, 1981.

27. Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

28. Richard L. Sites. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

29. W. Waite and G. Goos. *Compiler Construction*. Texts and Monographs in Computer Science. Springer, 1985.

30. M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, June 1984. SIGPLAN 84 Symp. On Compiler Construction.

31. W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.

A Basic Block Graphs

A *BB*-program is given by a set of basic blocks where each block consists of a sequence of instructions where the last one in a block is a jump or stop. *INSTR* denotes the universe of instructions. The data types used in this article are the type of 64-bit integers *INT* and the addresses *ADDR* on the target machine. *VALUE* denotes the union of all universes. Expressions are defined on these types and include only integer expressions and address expressions for simplicity. The universe *EXPR* denotes all expressions. The complete abstract state machine specification can be found in [12]. Expressions are evaluated by $eval : EXPR \to VALUE$ defined recursively over its structure (see ASM1). $\oplus_A$ is the add operation on addresses of the machine, which is in our case equivalent to $\oplus_I$. Instructions consist of assignment instructions for different kind of expressions, jumps and procedure calls (ASM2).

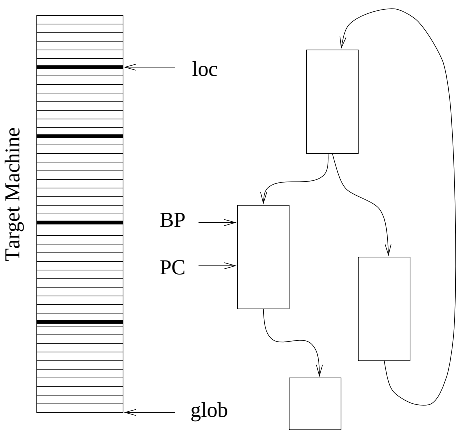


Figure3. Basic block graphs

ASM1

$$\begin{aligned}
eval(local(a)) &\mathrel{\hat{=}} content(eval(a))\\
eval(intconst_i) &\mathrel{\hat{=}} i\\
eval(intadd(e_1,e_2)) &\mathrel{\hat{=}} eval(e_1) \oplus_I eval(e_2)\\
&\vdots\\
eval(local(i)) &\mathrel{\hat{=}} local \oplus_A i\\
eval(global(i)) &\mathrel{\hat{=}} global \oplus_A i
\end{aligned}$$

ASM2

```
if $IP = intassign(a,e)$ then
   $content(eval(a)) := eval(e)$;
   $IP := NextInstr(IP)$
endif
```

ASM3

```
if $IP = condjump(e,b_1,b_2)$ then
   if $eval(e)$ then $BP := b_1$;
      $IP := first(b_1)$
else $BP := b_2$;
      $IP := first(b_2)$
   endif
endif
```

B The Dec-Alpha Processor Family

In this section we sketch the formal represenation of the DEC-Alpha based on the more or less informal specification in the manufacturer manual [28].
The formalization shows parts of the derived abstract state machine specification. It includes the instruction set, addressing modes, register files and the memory, i.e. it models the programmer's view. More details can be found in [13] and [8]. The semantics of DEC-Alpha instructions are given by state transition functions. Dynamic functions of the abstract state machine constitute the state which consists of

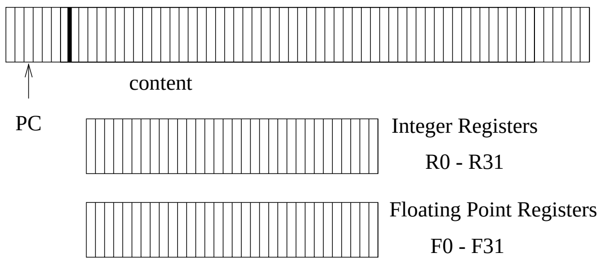


Figure4. State target machine

- the memory represented as a function $mem : QUAD \to BYTE$,
- two register files, $IntReg : REG$ for the available 32 integer registers and $FloatReg : REG$ for 32 floating-point registers, and
- the program counter $PC : QUAD$.

```
LOAD-ADDRESS                                                          ASM4
if $cmd$ is LDA (ra, disp, rb, high)
then
        if high="L"
        then Content (ra) := Content (rb) $\oplus_I$ $Sext_{16}$ (disp)
        else Content (ra) := Content (rb) $\oplus_I$ $LogShift_L$ ($Sext_{16}$ (disp), 16)
        endif
        CurInstr := NextInstr (CurInstr)
endif
```

```
STORE                                                                 ASM5
if $cmd$ is ST (ra, disp, rb, type)
then
        Content (Content (rb) $\oplus_I$ $Sext_{16}$ (disp) ) := Content (ra)
        CurInstr := NextInstr (CurInstr)
endif
```

```
ADD-REGISTER                                                          ASM6
if $cmd$ is ADD (ra, rb, rc, type)
then
        Content (rc) := Content (ra) $\oplus_I$ Content (rb)
        CurInstr := NextInstr (CurInstr)
endif
```

```
ZERO-BYTES-IMMEDIATE                                                  ASM7
if $cmd$ is ZBI (ra, immed, rc)
then
        Content (rc) := ByteZap (Content (ra), $Zext_8$ (immed))
        CurInstr := NextInstr (CurInstr)
endif
```

The addressable memory unit is a byte. In order to load and store quadwords – the usual integer type for DEC-Alpha architectures – or floats we introduce the function *Content* : $QUAD \cup REG \rightarrow VALUE$ which loads and stores 8 bytes from/into memory. For example, fetching a quadword or float from memory is carried out by concatenating 8 subsequent bytes starting at the given address.
The DEC-Alpha is a typical "load-store" architecture, what means that the only memory accessing functions are load and store instructions, addresses are given register relative. Some examples are given in ASM4-7, auxiliary functions have the following definition:
ByteZap results operand $op1$ with byte n set to zero, if bit n of operand $op2$ is set:

$$(ByteZap(op1, op2))\langle 8*n+7 : 8*n\rangle = \begin{cases} (op1)\langle 8*n+7 : 8*n\rangle: & (op2)\langle n\rangle = 0_2 \\ 0_2^8: & otherwise \end{cases}$$

$Sext_k(X)$ returns operand X sign extended from bit k to 64 bits. $Zext_k(X)$ returns operand X zero extended from bit k to 64 bits.

NuSmv: a reimplementation of Smv

Alessandro Cimatti[1] Ed Clarke[2] Fausto Giunchiglia[1] Marco Roveri[1,3]

{cimatti,fausto,roveri}@irst.itc.it Edmund.Clarke@cs.cmu.edu
[1]IRST, 38050 Povo, Trento, Italy
[2]Carnegie Mellon University, Pittsburgh, Pennsylvania
[3]DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract

This paper describes the first results of a joint project between CMU and IRST whose goal is to produce a reimplementation of Smv. The idea is that this new model checker, called NuSmv, should be usable, customizable and extensible, with as little effort as possible, also by people different from the developers. A further goal is to produce a system which is very robust, and close to the standards required by industry.

1 Introduction

This paper describes the first results of a joint project between CMU and IRST whose goal is to produce a reimplementation of Smv [11]. The new model checker, called NuSmv, is designed to be a well structured, flexible and documented platform, and should be usable, customizable and extensible with as little effort as possible also from people different from the developers. Furthermore, in order to make NuSmv applicable in technology transfer projects, it was designed to be very robust, close to the standards required by industry, and to allow for expressive specification languages.

With respect to Smv, NuSmv is being upgraded along three dimensions, namely:

- **quality of the implementation**. This will allow us to have a system which is very robust, and whose code is well documented, easy to understand and modify.
- **system architecture**. This will allow us to have a system whose architecture is very modular (thus allowing the substitution or elimination of certain modules) and open (thus allowing the addition of new modules). A further feature is that in NuSmv the user can control, and possibly change, the order of execution of some of the system modules.

- **system functionalities**. This will allow us to have a system with more user functionalities (e.g., multiple interfaces, a simulation mode) and more heuristics for, e.g., achieving efficiency or partially controlling the state explosion.

The first two dimensions involve a lot of software engineering and are instrumental to the third. NuSMV is currently being beta-tested, and will be shortly distributed publicly in the Web. Our work will then concentrate on adding new functionalities, possibly developed by other groups. We briefly describe the implementation, architecture and functionalities of NuSMV in Sections 2, 3, and 4, respectively.

2 Implementation

The implementation of NuSMV has the following features:

1. NuSMV is written in ANSI `C` and is `POSIX` compliant. This makes the system portable to any platform compliant with these standards. NuSMV has been throughly debugged with `Purify`[1] to detect memory leaks and runtime memory corruption errors.

2. The code of NuSMV has been documented following the standards of the EXT system[2]. This allows for the automatic extraction of the documentation from the comments in the system code. The documentation (e.g. the *User Manual*, the *Tutorial*, the *Programmer Manual*) will be available in different formats (for instance POSTSCRIPT, DVI, PDF, INFO, HTML and TXT), directly from the NuSMV interaction shell (see below), via an HTML viewer or in hardcopy.

3. A kernel has been isolated which provides the low level functionalities such as dynamic memory allocation, and manipulation of basic data structures (e.g. cons cells, hash tables). The kernel also encapsulates the state of the art CUDD binary decision diagrams (BDD) package developed at Colorado University [13]. This kernel can be used as a black box, following coding standards which have been precisely defined.

[1]Purify is a commercial product for run-time error detection. More information on this tool can be found at the url "`http://www.pureatria.com`".

[2]The EXT system is a set of programs that generate documentation for the World-Wide Web from specially-formatted C programs. These are being used in a variety of large software projects and have been shown to simplify the programmer's task. More information about this documentation tool can be found at the url "`http://www.alumni.caltech.edu/~sedwards/ext`"

4. In order to implement a top level interaction shell, the code of NuSmv was separated in different packages, each implementing a set of related functionalities. Each package is associated with a set of commands which can be interpreted by the NuSmv interaction shell. The implementation of the interaction shell required the definition and implementation of an error trapping mechanism (Smv exits if any kind of error occurs).

3 Architecture

The architecture of NuSmv is organized in the following modules:

1. The parsing routines, which process a file written in Smv language, build a parse tree representing the internal format of the input language, and check its syntactic correctness.

2. The routines for the encoding of data types and finite ranges into boolean domains. Different encoding policies can be more appropriate for different kinds of variables (e.g. control path, data path). Currently only the standard Smv encoding is possible. The separation of the encoding procedures will allow for the integration in the same architecture of other forms of encoding, e.g. those used in Word-Level Smv ([5]).

3. The model compiler, which from the parse tree builds the finite state machine (FSM) representing the model (e.g. the transition relation, the initial states, the fairness conditions).

4. The routines for constructing and manipulating FSM's. FSM's can be represented in monolithic or partitioned form [10]. The interface to other modules is given by the primitives for the computation of image and counter-image of sets of states.

5. The reachability and model checking routines. These are independent of the particular method used for representing the FSM. They use the routines for computing image and counter-image which are independent of the actual partitioning method used.

6. The routines for counterexample and witness generation and inspection. Counterexamples and witnesses can be produced with different levels of verbosity, in the form of reusable data structures, and can subsequently be inspected and navigated.

7. The interaction with the user, which has two modes. The first is the usual Smv batch mode, where the different computation steps (e.g. parsing, model construction, reachability, model checking) are activated according

to a fixed, predefined algorithm. The other (new) mode is a top level interaction shell. Through the shell the user can activate, as system commands with different options, various NuSmv computation steps, which can therefore be invoked separately, possibly undone, and repeated under different modalities. These steps include the definition of a FSM, the parsing and model checking of CTL formulae, and the configuration of the BDD package. For instance, several automatic variable ordering methods and cache configuration mechanisms can be suitably tuned according to the application.

Most of the features described above were already present in Smv. In NuSmv, however, all the code has been restructured to be organized according to the system architecture described in this section. Substantial recoding has been done in order to achieve the architectural separation of the modules, in particular of the kernel and the model compiler. New code has been added for the implementation of the user interface.

4 Functionalities

With respect to Smv, the following functionalities have been added:

1. NuSmv can perform enhanced reachability analysis. Specialized routines allow for checking invariants, i.e., formulae which must hold uniformly on the model, on the fly.

2. NuSmv allows for the conjunctive and disjunctive partitioning of the model [10], and inspection and definition of a suitable order of partitions according to the heuristics defined in [7] and in [12].

3. NuSmv integrates LTL model checking primitives which are obtained according to the algorithm proposed in [6]. LTL can be very important for selective analysis, as it allows the user to limit the paths of interest with an expressive language. An LTL formula is automatically converted into a tableau, which is then used to extend the model in synchronous product. The result is provided by checking the truth of a CTL formula in the extended model. The loose integration suggested in [6] allows for using Smv as a black box by generating an additional module to the source file. However, for each LTL formula to be verified, the verification process must restart from scratch from the parsing of the new file. In NuSmv the verification of LTL formulae is tightly integrated, and only requires the generation of the corresponding tableau, but not the reconstruction of the whole model.

The following further functionalities are currently under development:

1. A graphical interface (based on TCL/TK), for a more user friendly interaction. This interface will allow the user to activate the functionalities of the system and present graphically the defined model and the results of the analysis.

2. A sequential input language. The problem with the SMV language is that it is most amenable for hardware and hardware-like systems. However, the ability to model software systems is left to the user, and requires the burden of a complex model generation activity (e.g. the explicit introduction of program counters). In general this can be hardly acceptable, as many specification languages (e.g., SDL) are intrinsically sequential. A possible new input language for NuSMV is VERUS [3].

3. A simulation functionality, which allows the user to acquire confidence with the correctness of the model before the actual verification of properties.

Some lines of research which are currently under study and whose results we plan to integrate inside NuSMV are specific reduction techniques for sequential systems, a set of abstraction techniques which implement certain heuristics developed in the theorem proving community [9] and that we believe will be very effective, the extension of CTL model checking to multi-agent systems and security applications [1], and the integration of model checking and theorem proving (SAT in particular) following on the ideas reported in [8].

5 Conclusion

In this paper we describe the first result of a project aiming at the development of NuSMV, a robust, well designed and flexible model checker that could be applicable in technology transfer projects. NuSMV has already been used as the kernel of MBP, a planner based on model checking, able to synthesize reactive controllers for achieving goals in nondeterministic domains [4]. The development of MBP was greatly simplified by the architecture and features of NuSMV. Many ideas about the architecture of NuSMV have been taken from a close analysis of the implementation and architecture of the VIS system [2]. NuSMV should soon be available public domain in the Web.

References

[1] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. To appear in Computational & Logical Aspects of Multi-Agent Systems. A special Issue of the Journal of Logic and Computation, 1997. Also IRST-Technical Report 9708-07, IRST, Trento, Italy.

[2] R. K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz S., Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A system for Verification and Synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. Computer Aided Verification (CAV'96)*, number 1102 in LNCS, New Brunswick, New Jersey, USA, July/August 1996. Springer-Verlag.

[3] S. Campos, E. Clarke, and M. Minea. The Verus Tool: A quantitative approach to the formal verification of real-time systems. In Orna Grumberg, editor, *Proc. Computer Aided Verification (CAV'97)*, number 1254 in LNCS, Haifa, Israel, June 1997. Springer-Verlag.

[4] A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Carnegie Mellon University, Pittsburgh, USA, June 1998. AAAI-Press.

[5] E. Clarke and X. Zhao. Word Level Symbolic Model Checking: A New Approach for Verifying Arithmetic Circuits. Technical Report CMU-CS-95-161, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA, May 1995. `ftp://reports.adm.cs.cmu.edu/usr/anon/1995/CMU-CS-95-161.ps`.

[6] O. Grumberg E. Clarke and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):57–71, February 1997.

[7] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Proc. Computer Aided Verification (CAV'94)*, number 818 in LNCS, Stanford, California, USA, June 1994. Springer-Verlag.

[8] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. of the 13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, New Brunswick, NJ, USA, August 1996. Springer Verlag. Also DIST-Technical Report 96-0037 and IRST-Technical Report 9601-02.

[9] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57(2-3):323–390, 1992. Also IRST-Technical Report 9001-14, IRST, Trento, Italy.

[10] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 13(4):401–424, April 1994.

[11] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publ., 1993.

[12] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.

[13] F. Somenzi. CUDD: CU Decision Diagram package — release 2.1.2. Department of Electrical and Computer Engineering — University of Colorado at Boulder, April 1997.

XTL: A Meta-Language and Tool for Temporal Logic Model-Checking

R. Mateescu* and H. Garavel‡

**CWI / SEN2 group*
413, Kruislaan
NL-1098 SJ Amsterdam, The Netherlands
Radu.Mateescu@cwi.nl

‡*INRIA Rhône-Alpes / VASY group*
655, avenue de l'Europe
F-38330 Montbonnot St. Martin, France
Hubert.Garavel@inria.fr

Abstract

We present a temporal logic model-checking environment based on a new language called XTL (*eXecutable Temporal Language*). XTL is a functional programming language designed to allow a compact description of various temporal logic operators, which are evaluated over a Labelled Transition System (LTS). XTL offers primitives to access the data values (possibly) contained in the states and labels of the LTS, as well as to explore the transition relation. The temporal logic operators are implemented by means of iteration expressions computing sets of states and sets of transitions. Various useful modal and temporal logics like HML, CTL, LTAC and ACTL, have been implemented in XTL, and several industrial case-studies, such as the BRP protocol designed by Philips, the IEEE-1394 serial bus standardized by IEEE, and the CFS protocol developed by Bull and INRIA, have been successfully validated using the XTL model-checker.

1 Introduction

The last years have witnessed an increasing application of formal methods in the design and validation of complex applications, such as communication protocols and distributed systems. One of the most popular techniques of program verification is the so-called *model-checking*. In this approach, the application is first described using an appropriate high-level language, such as LOTOS[1] [17] or μCRL[2] [12]. Next, the program is translated into a Labelled Transition System model (LTS for short), over which the desired correctness properties, expressed as temporal logic formulas, are verified by means of specialized tools called *model-checkers*.

The literature concerning this area is very rich in results: a large variety of temporal logics have been defined, allowing to capture different kinds of correctness properties, and several corresponding model-checking algorithms have been proposed. Also, numerous tool environments allowing verification by model-checking have been developed, such as EMC [4], CWB [6], SPIN [15], TAV [19], MEC [1], JACK [3], and Concurrency Factory [5], to mention only a few of them.

However, many of the currently available tools are either dedicated to a particular description language and/or temporal logic (e.g., the language PROMELA [15] used in

[1]Language of Temporal Ordering Specification
[2]*micro* Common Representation Language

SPIN, the logic ACTL [23] used in JACK, etc.), or they are based on a particular model-checking algorithm (e.g., the boolean resolution algorithm [2] used in MEC). Therefore, most of these tools have limited applicability in different contexts, and their adaptation to another setting may be overwhelming in terms of time and implementation effort.

Another important issue is the handling of data values, both at the level of the description language and of the temporal logic. For instance, the LTS models corresponding to value-passing description languages as LOTOS or μCRL contain data values in the states and/or transition labels. This requires the ability to express and verify temporal properties involving data, e.g., "*after a message* `m` *has been sent, the same message* `m` *will be eventually received.*" Although studied in the theorem-proving approach [26], this issue has received little attention in the setting of automated model-checking.

In this paper, we present an approach to temporal logic model-checking that attempts to reduce the shortcomings mentioned above. Our method is based on a meta-language called XTL (*eXecutable Temporal Language*), which is a functional programming language designed to allow a compact description of temporal logic operators. We use the term "meta-language" to emphasize that XTL allows not only to handle, in a uniform way, the data objects (i.e., types and functions) defined in the program to be verified, but also the states, transitions, and labels of the corresponding LTS model. Furthermore, since XTL is a programming language, it can be used to define non-standard temporal operators and, more generally, to perform any computation on an LTS model (e.g., to calculate the branching factor, print the list of labels, etc.).

The XTL model-checker has been developed as part of the CADP (CÆSAR/ALDÉBARAN Development Package) protocol engineering toolset [9]. We describe here the version 1.1 of the XTL tool, which is currently integrated in CADP[3].

The paper is organized as follows. Section 2 gives an overview of the XTL language and shows various examples of temporal logic operators implemented in XTL. Section 3 briefly describes the architecture of the XTL model-checker. Section 4 presents several applications to industrial case-studies. Finally, Section 5 contains some concluding remarks and directions for future work.

2 Overview of the XTL language

In this section, we first describe the (extended) LTS models over which XTL programs are interpreted, and next we present the basic XTL constructs, illustrating their use by means of various examples. Due to space limitations, we cannot describe here in full detail the whole XTL language. A more detailed presentation can be found in [22] and in the technical documentation of the CADP toolset.

2.1 Labelled Transition Systems

In order to verify temporal properties of programs written in value-passing description languages as LOTOS and μCRL, we must naturally use an adequate representation of the corresponding LTS models, the states and labels of which may contain data values. Such a representation is available within the CADP toolset as a special file format called BCG (*Binary Coded Graphs*) [10]. A BCG file representing the LTS model of a program to be verified (denoted by "source program" in the remainder of the paper) contains essentially the following informations:

[3]The CADP toolset can be obtained at the URL `http://www.inrialpes.fr/vasy/cadp.html`.

- A set of *states*, each of them being represented as a tuple containing the values of all the program variables (the so-called *state-vector*). An initial state is identified.
- A set of *actions* (also called *labels*), each of them being represented as a list of typed values. In BCG files generated from LOTOS programs, the labels have the form "$G\ v_1 \ldots v_n$," where G is a gate name.
- A *transition relation*, represented as a list of *transitions* encoded as tuples of the form (s_1, a, s_2), each of them indicating that the program can move from state s_1 to state s_2 by performing action a.

Besides the above elements, a BCG file generated from a source program contains also a *type area* and a *function area* that give access to the types and functions defined in the source program, respectively. Throughout this section, we implicitly consider an LTS model represented in BCG format, over which the XTL constructs will be interpreted.

2.2 Types, expressions, and functions

As we mentioned earlier, XTL allows to handle, in a uniform way, the data values used in the source program, as well as the elements of the corresponding LTS model. To achieve this, the XTL language allows to define and use objects belonging to the following types.

external types: These are the types defined in the source program; they are exported by the type area of the BCG file encoding the LTS. The data values belonging to these types can be handled using the functions defined in the source program, which are exported by the function area of the BCG file.

internal types: These are the types predefined in XTL. The standard predefined types (`boolean`, `integer`, `character`, etc.) are provided, together with their usual operators. Beside these types, there are also the so-called *meta-types* `stateset`, `state`, `edgeset`, `edge`, `labelset`, and `label`, denoting the (sets of) states, transitions, and labels of the LTS, respectively. These types are equipped with the *meta-operators* given in Table 1, which allow to access the initial state of the LTS and to explore the transition relation (some of these operators are inspired from Dicky's calculus [8]).

OPERATOR	MEANING
`init : -> state`	initial state
`succ, pred : state -> stateset`	successors and predecessors of a state
`in, out : state -> edgeset`	incoming and outgoing transitions of a state
`source, target : edge -> state`	origin and destination states of a transition

Table 1: Basic XTL meta-operators

The basic XTL expressions are shown in Table 2. Function calls may be either in prefix, or infix notation (in the case of binary operators, such as the predefined operations "`+`", "`*`", "`<=`", etc.). The label-matching expression returns a result of type `boolean`. The construct enclosed in its brackets (called *action pattern*) allows to examine the structure of a transition label of the LTS and (possibly) to extract the values of its fields and bind them to variables. Quantifiers over finite domains and comprehensive set definitions have a syntax close to their usual mathematical notation.

The most simple way to implement in XTL temporal operators expressing action or state properties is to compute their denotational semantics, i.e., the sets of LTS labels or states satisfying them. For example, the following expression computes the set of labels corresponding to the emission of a signal with different source and destination addresses (identifiers are in upper-case letters and keywords in lower-case):

```
{ L:label where L -> [ SIGNAL ?S:Addr ?D:Addr where S <> D ] }
```

The variables S and D, initialized by pattern-matching with the corresponding values contained in the label, are used in the "**where**" clause, which allows additional filtering using a boolean condition.

EXPRESSION	MEANING
$F(E_1, ..., E_n)$	prefix function call
E_1 F E_2	infix function call
E -> [G ?x:T !E_1 [**where** E_2]]	label matching
exists x:T **in** E **end_exists**	existential quantifier
forall x:T **in** E **end_forall**	universal quantifier
{ x:T **where** E }	set in comprehension
let x:T:=E **in** E_1 **end_let**	variable definition
if E **then** E_1 **else** E_2 **end_if**	conditional
for [x_0:T_0] [**in** x_1:T_1] [**while** E_1] **apply** F **from** E_2 **to** E_3 **end_for**	iteration

Table 2: Summary of the basic XTL expressions

Used together with the quantifiers, the set definition construct allows to easily express modal operators. For instance, the $\langle\alpha\rangle\,\varphi$ modality of the Hennessy-Milner logic HML [14] can be implemented by the XTL function below:

```
def Dia (A:labelset, F:stateset) : stateset =
    { S:state where
        exists T:edge among out (S) in
            (label (T) among A) and (target (T) among F)
        end_exists
    }
end_def
```

The parameters A and F denote the sets of labels and states satisfying α and φ, respectively. The function call "Dia (A, F)" returns the states satisfying $\langle\alpha\rangle\,\varphi$, i.e., the states having an outgoing transition whose label satisfies α and whose target state satisfies φ.

The "**let**" and "**if**" constructs shown in Table 2 have their usual meaning (e.g., as in ML). The evaluation of the iteration construct "**for**", which allows to perform repeated computations, proceeds as follows. We first assume that the declaration x_0:T_0 is present, but the "**in**" and "**while**" clauses are absent. The semantics of "**for**" uses an implicit variable v_{acc} (called *accumulator*) initialized with the value of E_2. For each value of x_0 (called *iteration variable*) in the finite domain T_0, an iteration is performed, that consist in evaluating the expression $F(v_{acc}, E_3)$ and assigning it to v_{acc} (note that F must be a binary function). The result of the "**for**" expression is the value of v_{acc} after the last iteration. For example, the following XTL expression computes the maximal branching factor (i.e., the maximal number of transitions going out of a state in the LTS):

```
for S:state
    apply max
    from  0
    to    card (out (S))
end_for
```

where `max` denotes the maximum of two integer numbers and `card` gives the number of elements of a transition set.

Optionally, the "**in** $x_1 : T_1$" clause allows to give a name x_1 to the accumulator v_{acc} so that it can be referenced in E_1 and/or E_3. If present, the "**while** E_1" clause allows to control the execution of the "**for**" expression: the iterations are performed as long as the boolean expression E_1 (re-evaluated before each iteration) remains true. An absence of the iteration variable x_0 means a "forever" loop: in this case, the iterations are stopped using the "**while** E_1" clause (which must be present in order to ensure termination).

Using the "**for**" construct, temporal operators can be defined in a compact form. Thus, the following XTL function implements the operator $\mathbf{EF}_\alpha \varphi$, which is a derived modality of ACTL [23]:

```
def EF_A (A:labelset, F:stateset) =
    for in    X:stateset
        while X <> (F or Dia (A, X))
        apply or
        from  false
        to    F or Dia (A, X)
    end_for
end_def
```

A state satisfies $\mathbf{EF}_\alpha \varphi$ if it is the origin of a path leading, via zero or more actions satisfying α, to a state satisfying φ. This can be characterized as the least solution of the fixed point equation $X = \varphi \vee \langle\alpha\rangle\, X$, that is iteratively computed by the "**for**" expression above. Note the overloading of the boolean operators `or` and `false`, that denote the union and the empty set of states, respectively. Alternately, the $\mathbf{EF}_\alpha \varphi$ operator could be defined using a recursive XTL function.

2.3 Macros, libraries, and programs

In order to express temporal properties involving data conveniently, a higher-order mechanism for handling predicates containing free variables is needed. For this purpose, we incorporated in XTL a macro-expansion mechanism, which covers most practical user needs and can be implemented simply and efficiently. The following example of XTL macro-definition implements the $[\alpha]\,\varphi$ modality of HML, characterizing the states from which all outgoing actions satisfying α lead to states satisfying φ:

```
macro Box (A, F) =
    { S:state where
        forall T:edge among out (S) in
            if T -> [ A ] then target (T) among F else true end_if
        end_forall
    }
end_macro
```

The A and F parameters above denote (the textual representation of) an action pattern and an expression of type stateset. A macro call "Box (<textA>, <textF>)" is replaced in the XTL program by the body of the macro, in which the occurrences of A and F have been textually substituted with <textA> and <textF>. For instance, the following XTL macro call evaluates the set of states from which every message M sent on gate SEND can be potentially received on gate RECV:

```
Box (SEND ?M:Msg, EF_A (true, Dia (RECV !M, true)))
```

where Dia is a macro implementing the $\langle\alpha\rangle\,\varphi$ modality and EF_A is the function defined in Section 2.2. The variable M, which extracts the message contained in the SEND labels, is visible in the second argument of the Box operator; this is ensured by the static semantics of the "**if**" expression used in the body of the Box macro-definition above. Note also that the type Msg is external, i.e., it is defined in the source program.

XTL allows the macro-definitions to be overloaded: several macros having the same name, but different number of parameters, may be used in the same scope. This is convenient for defining derived temporal operators having the same name, for instance the $\mathbf{pot}(\varphi_1, \varphi_2)$ and $\mathbf{pot}(\varphi)$ operators of LTAC [25], which are similar to the $\mathbf{E}[\varphi_1 \,\mathbf{U}\, \varphi_2]$ and $\mathbf{EF}\,\varphi$ operators of CTL [4], respectively.

Another useful feature is the possibility to include in an XTL program other XTL files, typically containing libraries of temporal operators. This allows to reuse existing XTL code and also to provide different implementations of the same temporal logic (see Section 4). For example, the following construct denotes the textual inclusion of an XTL source file implementing the ACTL temporal logic:

```
library actl.xtl end_library
```

An XTL program consists of an expression (the program's body) preceded by an optional list of macro-definitions and library inclusions.

3 Implementation

We developed a model-checker for XTL as part of the CADP protocol engineering toolset. The tool takes as input an XTL program and an LTS model encoded in BCG format, evaluates the program over the LTS and produces the results.

The architecture of the model-checker is shown in Figure 1. First, the XTL program is processed by an auxiliary tool called *expander*, that textually expands the macro-definitions and includes the XTL libraries used in the program. The resulting program (containing "pure" XTL code, i.e, without macro calls) is translated into a C program, which is then compiled and linked with the BCG libraries. Note that the information contained in the BCG file is used also during the static analysis, since the types and functions defined in the source program (exported by the BCG file) can be used in the XTL program. The object file obtained in this way is executed and the results are obtained on the standard UNIX output stream.

The version 1.1 of the XTL model-checker is available on SUN workstations running SUNOS or SOLARIS and PCs running LINUX. The syntax analyzer has been implemented using the SYNTAX[4] compiler generator. The semantics analyzer has been written in LOTOS abstract data types, which are translated into C code using the CÆSAR.ADT compiler of the CADP toolset. The expander, the code generator, and all the interfacing

[4]SYNTAX is a trademark of INRIA.

code with the BCG environment have been written in C. The overall implementation consists of about 27,000 lines of code.

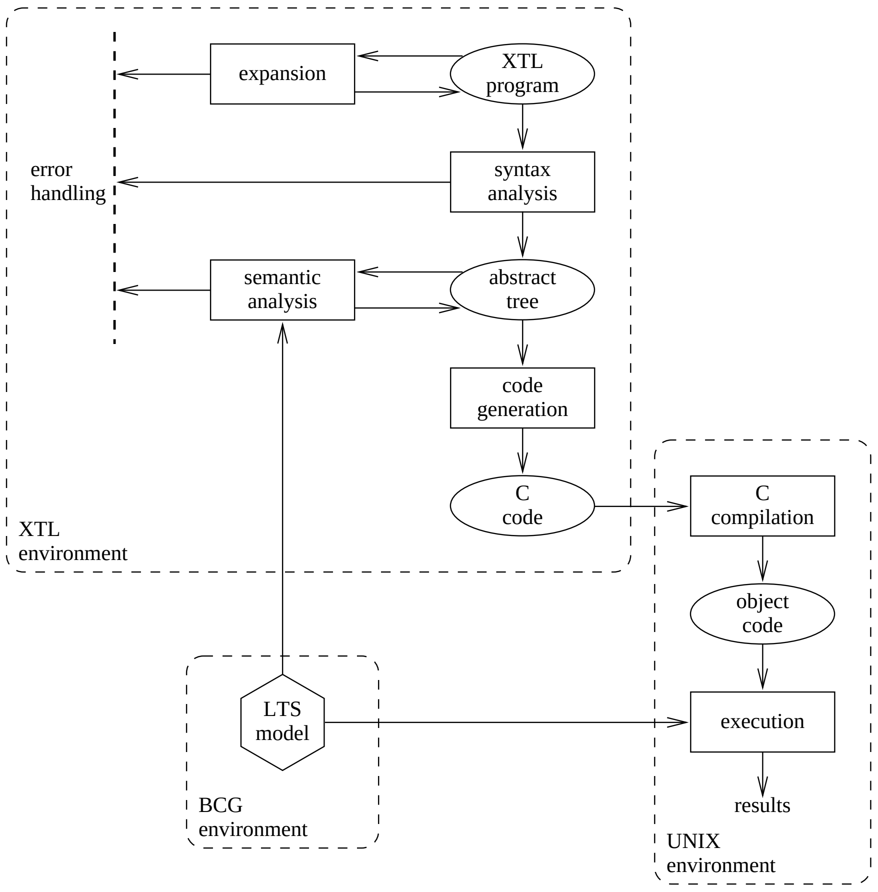


Figure 1: Architecture of the XTL model-checker

Besides developing the model-checker, we also provided XTL libraries implementing the operators of HML [14], CTL [4], LTAC [25], ACTL [23], and the modal μ-calculus [18]. All these operators can be naturally used in conjunction with the built-in XTL data-handling facilities in order to express temporal properties involving data.

4 Applications

An initial version 1.0 of the XTL model-checker has been used to verify several LOTOS descriptions of small size, such as the alternating bit protocol, a leader election algorithm, and various mutual exclusion algorithms. These exercises, together with the experience of using XTL for teaching purposes at the University Joseph Fourier of Grenoble, provided valuable feedback, enabling us to improve the tool from the initial version 1.0 to the current version 1.1. So far, this version of the XTL model-checker has been used to validate three medium-sized industrial case-studies.

Bounded Retransmission Protocol: The BRP protocol has been designed by Philips and is currently used in the remote control devices of television sets. It implements the transmission of (large) data packets over an unreliable communication medium by splitting them in (small) chunks that are sent sequentially. Whenever a chunk is lost, it is retransmitted over the communication medium. If a (fixed) maximal number of retransmissions is reached, the protocol gives up the transmission of the packet, appropriately informing the sender and the receiver. This protocol was proposed by Jan Friso Groote (CWI, Amsterdam) as a verification exercise intended for the comparison of several formal methods.

Starting from a μCRL description given in [13], we produced a LOTOS description of the BRP protocol, for which we identified a set of 21 safety and liveness properties, expressed in XTL using the library of ACTL operators combined with data. These properties have been successfully verified, using the XTL model-checker, on different instances of the protocol, obtained by giving different values to the maximal number of retransmissions and to the packet length [21].

Link Layer of the IEEE-1394 Serial Bus: The IEEE-1394 serial bus ("FireWire") is a high-speed bus particularly adapted to data transmission for multimedia devices connected to computers. This bus, standardized by IEEE, is currently used by numerous constructors, such as AT&T, Canon, Compaq, Hewlett-Packard, IBM, Kodak, Microsoft, Sony, Texas Instruments, etc.

We carried out the validation of the asynchronous part of the link layer protocol of IEEE-1394. Based upon a μCRL description provided by Bas Luttik [20] and upon the IEEE standard [16], we produced an E-LOTOS description of this part of the protocol, which was subsequently translated in LOTOS using the TRAIAN prototype compiler of the CADP toolset. We translated in XTL (using the library of ACTL operators extended with data) the 5 correctness properties of the protocol stated in natural language by Luttik. These properties have been verified, using the XTL tool, on several instances of the protocol, obtained for different numbers of nodes connected to the bus and for various message scenarios. This allowed us to detect and correct a potential deadlock occurring in the protocol after about 50 transitions from the initial state [27].

Cluster File System: CFS is a distributed file system developed by Bull and INRIA on top of the ARIAS shared memory architecture [7]. CFS was designed both to validate the ARIAS system itself and to experiment with distributed applications that use shared files as a programming model.

The validation of the migratory file coherency protocol of CFS (referred to as the CFS protocol in the sequel) has been recently carried out by Charles Pecheur. First, he produced a LOTOS description modelling both the CFS protocol and the ARIAS service primitives used by it. Next, he specified a set of 15 safety, liveness, and coherency properties (expressed as ACTL formulas with data) of the control level (i.e., involving only the calls to CFS synchronization primitives) and of the data level (i.e., taking into account also the access and modification of the data files). Finally, he implemented in XTL a new library of ACTL operators, able to produce diagnostic sequences explaining the truth value of a formula using the XTL tool. These properties have been verified on various scenarios of CFS, obtained for different application configurations on top of the CFS protocol [24].

These experiments confirm the usefulness of the XTL language: temporal properties involving data can be expressed in a natural way using directly the notations of the source program. Although the action-based operators of ACTL are often enough powerful to express the safety and liveness properties encountered in practice, there are situations (e.g., some of the properties of the BRP protocol) that can be handled in an easier way using more powerful constructs, such as regular expressions. These can be implemented in XTL by means of their fixed point characterizations.

5 Conclusion and future work

Formal methods have proved their usefulness in the design of complex, distributed applications. Among these methods, model-checking verification techniques are simple to use and completely automated, although limited to finite-state systems.

We presented in this paper a model-checking environment based on a special language called XTL, dedicated to the description of temporal properties involving data. A model-checker for XTL has been developed, and several widely-used temporal logics like HML, CTL, LTAC, ACTL, and the μ-calculus, have been implemented in XTL. The version 1.1 of the model-checker is available as part of the CADP protocol engineering toolset, and has been successfully used to validate several industrial case-studies [21, 27, 24].

These experiments are encouraging, confirming the advantages of the approach adopted in designing XTL, which allows to combine temporal operators and data-handling constructs. Moreover, since XTL is a programming language, it allows the user to implement new temporal logics or to extend existing ones with new operators. Indeed, Charles Pecheur developed in XTL a new library of ACTL temporal operators, able to produce diagnostic sequences [24].

The work presented here can be extended in several directions. Firstly, our experience shows that additional data types (such as sequences and subtrees of the LTS) are needed in order to facilitate the implementation of temporal operators with diagnostic features. Secondly, there is still room for improving the performances of the model-checker: XTL being a functional language, the code generator should be optimized using appropriate storage allocation techniques for the variables containing sets of states and transitions of the LTS. Finally, the implementation of on-the-fly model-checking algorithms, which do not require to generate entirely the LTS *before* evaluating a temporal formula, can be envisaged along the lines described in [22], by using the OPEN/CÆSAR approach to on-the-fly verification [11].

References

[1] A. Arnold, D. Bégay, and P. Crubillé. *Construction and Analysis of Transition Systems with MEC.* World Scientific, 1994.

[2] A. Arnold and P. Crubillé. A Linear Algorithm to Solve Fixed-Point Equations on Transition Systems. *Information Processing Letters*, 29:57–66, 1988.

[3] A. Bouali, S. Gnesi, and S. Larosa. The Integration Project for the JACK Environment. *Bulletin of the EATCS*, 54:207–223, 1994.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: a Development Environment for Concurrent Systems. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96*, LNCS 1102, pages 398–401, 1996.

[6] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench.* In J. Sifakis, editor, *Automatic Verification of Finite State Systems*, pages 24–37. LNCS 407. 1989.

[7] P. Dechamboux, D. Hagimont, J. Mossiere, and X. Rousset de Pina. The Arias Distributed Shared Memory: an Overview. LNCS 1175, 1996.

[8] A. Dicky. An Algebraic and Algorithmic Method for Analysing Transition Systems. *Theoretical Computer Science*, 46(2-3):285–303, 1986.

[9] J-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox. In R. Alur and T. A. Henzinger, editors, *Proceedings of CAV'96*, LNCS 1102, pages 437–440, 1996.

[10] H. Garavel. Binary Coded Graphs — Definition of the BCG Format (version 1.0). Technical report, INRIA Rhône-Alpes, 1995.

[11] H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of TACAS'98*, LNCS, 1998. Full version available as INRIA Research Report RR-3352.

[12] J. F. Groote and A. Ponse. The Syntax and Semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[13] J. F. Groote and J. C. van de Pol. A Bounded Retransmission Protocol for Large Data Packets. Technical Report Logic Group Preprint Series 100, Utrecht University, 1993.

[14] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32:137–161, 1985.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[16] IEEE. Standard for a High Performance Serial Bus. IEEE Standard 1394-1995, Institution of Electrical and Electronic Engineers, 1995.

[17] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, ISO — OSI, Genève, 1988.

[18] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[19] K. G. Larsen. Efficient Local Correctness Checking. In G. v. Bochmann and D. K. Probst, editors, *Proceedings of CAV'92*, LNCS 663, pages 30–43, 1992.

[20] B. Luttik. Description and Formal Specification of the Link Layer of P1394. In I. Lovrek, editor, *Proceedings of the 2nd COST 247 Int. Workshop on Applied Formal Methods in System Design*, 1997. Also available as CWI Report SEN-R9706.

[21] R. Mateescu. Formal Description and Analysis of a Bounded Retransmission Protocol. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 Int. Workshop on Applied Formal Methods in System Design*, pages 98–113. University of Maribor, Slovenia, 1996. Also available as INRIA Research Report RR-2965.

[22] R. Mateescu. *Vérification des propriétés temporelles des programmes parallèles.* PhD Thesis, Institut National Polytechnique de Grenoble, 1998. To appear.

[23] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems.* In *Proceedings of Semantics of Concurrency*, pages 407–419. LNCS 469. 1990.

[24] C. Pecheur. Advanced Modelling and Verification Techniques Applied to a Cluster File System. Research Report 3416, INRIA Rhoône-Alpes, 1998.

[25] J-P. Queille and J. Sifakis. Fairness and Related Properties in Transition Systems — A Temporal Logic to Deal with Fairness. *Acta Informatica*, 19:195–220, 1983.

[26] J. Rathke and M. Hennessy. Local Model Checking for a Value-Based Modal μ-calculus. Report 5/96, School of Cognitive and Computing Sciences, University of Sussex, 1996.

[27] M. Sighireanu and R. Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS. *Springer Int. Journal on Software Tools for Technology Transfer (STTT)*, 1998. To appear.

New Generation of UPPAAL *

Johan Bengtsson[2] *Kim Larsen*[1] *Fredrik Larsson*[2]
Paul Pettersson[2] *Yi Wang* [2] *Carsten Weise*[1]

[1] BRICS, Dept of Computer Science, Aalborg University, Denmark.
[2] Department of Computer Systems, Uppsala University, Sweden.

Abstract. UPPAAL is a tool-set for the design and analysis of real-time systems. In [6] a relatively complete description of UPPAAL before 1997 has been given. This paper is focused on the most recent developments and also to complement the paper of [6].

1 UPPAAL's Past: the History

The first prototype of UPPAAL, named TAB at the time, was developed at Uppsala University in 1993 by Wang Yi et al. Its theoretical foundation was presented in FORTE94 [11] and the initial design was to check safety properties that can be formalized as simple reachability properties for networks of timed automata. The restriction to this simple class of properties was in sharp contrast to other real-time verification tools at that time, which where developed to check timed bisimularities or formulae of timed modal μ-calculi. However, the ambition of catering for more complicated formulae lead to extremely severe restrictions in the size of systems that could be verified by those tools.

The essential ideas behind TAB were to represent the state space of timed systems by simple constraints and to explore the state space by constraint manipulation. In 1995, Aalborg University joined the development, and shortly after a C++-version with efficient operations on constraints and checks for inclusion between constraints was finalized. TAB was subsequently renamed UPPAAL with UPP standing for Uppsala and AAL for Aalborg.

Since its first release in 1995, UPPAAL has in numerous case-studies proved itself useful in the analysis of safety properties of extremely complicated system descriptions. To our knowledge, UPPAAL has at present over one hundred users in both academia and industry. However, the number of downloads of UPPAAL binary code from the UPPAAL WWW-page is much larger according to the record of our WWW-server.

* UPPAAL is developed in collaboration between the Department of Computer Systems at Uppsala University, Sweden and BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation) at Aalborg University, Denmark. The people involved with the development are Wang Yi, Kim Larsen, Paul Pettersson, Johan Bengtsson, Fredrik Larsson, Kåre J. Kristoffersen, Carsten Weise, Per S. Jensen, and Thomas M. Sørensen.

For a detailed description for UPPAAL before 1997, we refer to [6]. During 1997, UPPAAL has been greatly improved e.g. the verification time for the well known Philips audio protocol [1] is reduced from 304 seconds to 5.5 seconds using the same hardware. In Sections 2 and 3 we report on this evolution and on the most recently added model-checking features such as facilities for checking time-bounded as well as (ordinary) liveness properties.

Right from the beginning UPPAAL has been applied in a number of case studies including an rapidly increasing number of case-studies with industrial collaboration. To meet requirements arising from the case studies, UPPAAL has been extended with various features leading to the current distributed version. In Section 4 we offer a brief summary of recent case-studies undertaken by UPPAAL.

The success of UPPAAL efficiency-wise has lead to a strong demand of a reimplementation of the graphical user interface. In particular, the verification engine of UPPAAL now routinely handles models which are too big to be displayed in full on a single screen: thus the ability to perform editing as well as simulation, while focusing only on a selection of relevant components is highly needed. The present distribution of UPPAAL contains two separated graphical tools: an AUTOGRAPH-based *editor* and a graphical *simulator* implemented in XFORMS and *Motif*. From a users perspective one single graphical interface would be preferable. In Section 5 we report on a new UPPAAL graphical user interface currently under implementation addressing these points.

2 UPPAAL's New Languages

Two major improvements have been made on the modeling and specification languages. First, we have introduced two new types: *bounded* integer and *array* of such integers, to simplify modeling. Second, the verifier has been extended to handle liveness properties in addition to reachability properties.

Bounded integers and arrays of integers Instead of using a default domain derived from the hardware implementation of integers, we now allow the user to specify the domains of each variable. However, if no domain is given by the user, a default domain (currently $[-32768, 32767]$) will be assigned to the variable. When assigning a value to a variable, the value is "wrapped" into the correct domain.

In order to ease the modeling task, the class of integer expressions handled by the verifier have been extended. As shown in Figure 1, the new verifier can handle general expressions over integer variables and constants. To allow more condensed models, arrays of integers and arithmetic if-statements have been added to the language. The syntax used for there constructs is the same as in the C programming language, i.e. `var[5]` denotes the sixth element of the array `var` and `(x<=5:2?3)` is 2 if the value of `x` is at most 5, and 3 otherwise.

$$
\begin{array}{lll}
\mathit{IExp} & \rightarrow & \mathit{Id} \mid \mathit{Id}\ \texttt{[}\ \mathit{IExp}\ \texttt{]} \mid \mathit{Nat} \mid \texttt{-}\ \mathit{IExp} \mid \texttt{(}\ \mathit{IExp}\ \texttt{)} \mid \mathit{IExp}\ \mathit{Op}\ \mathit{IExp} \mid \texttt{(}\ \mathit{IRel}\ \texttt{?}\ \mathit{IExp}\ \texttt{:}\ \mathit{IExp}\ \texttt{)} \\
\mathit{IRel} & \rightarrow & \mathit{IExp}\ \mathit{RelOp}\ \mathit{IExp} \\
\mathit{Op} & \rightarrow & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\mathit{RelOp} & \rightarrow & \texttt{<=} \mid \texttt{>=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>}
\end{array}
$$

Fig. 1. Syntax for integer expressions

Specifying liveness properties In addition to the reachability properties checked by the older UPPAAL versions, all versions above 2.12 are also capable of checking simple liveness properties. The liveness properties that can be checked are of the form $\exists\Box P$ and $\forall\Diamond P$, where P is a "local" property of the same kind as the properties handled by the reachability checker.

The actual checking is done by searching for a time divergent path from the initial state, where P holds in all states (in case of a $\exists\Box$ property), or where $\neg P$ holds in all states (in case of a $\forall\Diamond$ property).

Checking Deadlock-Freedom and Consistency In addition to the updates in the modeling and specification languages, the new version of UPPAAL also contains some features to simplify debugging. During verification the tool reports all inconsistent states (i.e. states where the location invariant is violated when the location is entered) and all deadlocked states (i.e. states where no discrete transition will be possible in the future) encountered.

3 UPPAAL's New Heart

In our previous work, before 1997, we have developed and implemented various techniques for optimizing the space- and time-performance of the reachability engine of UPPAAL [7]. The two major optimizations are an algorithm for compaction of constraints and a control structure analysis technique that identifies and discards states that are not necessary to ensure termination of the reachability algorithm [8]. When combined the two techniques yield significant space savings[1] and (usually) improved time-performance.

During 1997, a large part of the source code of the UPPAAL model checker was rewritten and optimized. Surprisingly, small obvious improvements on the

[1] The space saving on the examples in [8] are between 75% and 94%.

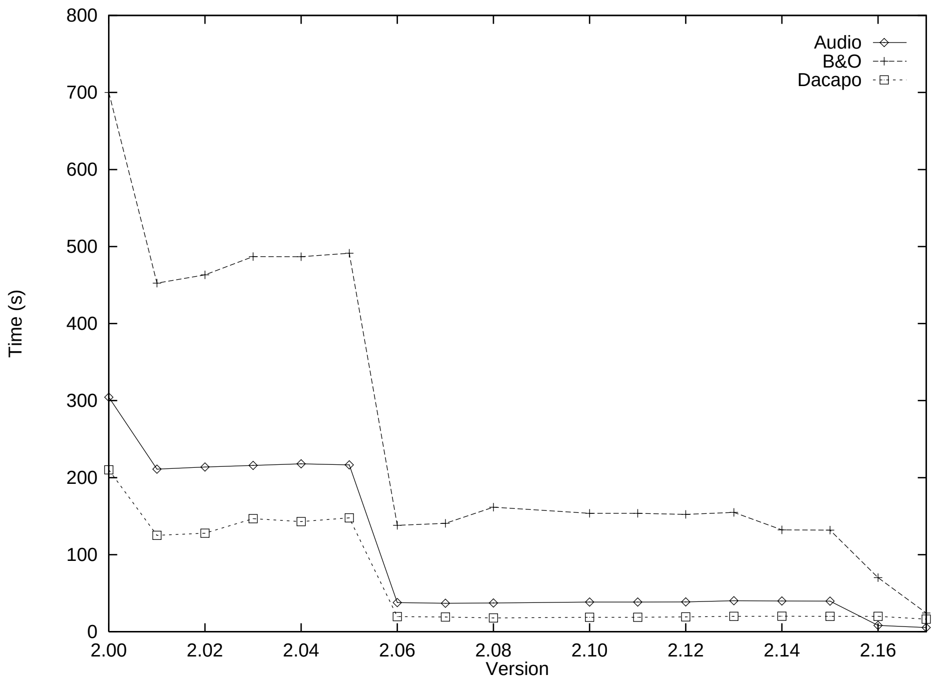


Fig. 2. Time benchmark for UPPAAL version 2.00–2.17.

source code, often yield huge improvements in efficiency. The most widely distributed version of UPPAAL is version 2.02, which is also the version presented in the paper [6]. However, the most efficient version is the current one 2.17.

In Figure 2 and 3, we illustrate the improvement of time and memory usage of UPPAAL from version 2.00[2] to 2.17[3] in terms of three case studies; the Philips audio control protocol with bus collision [1], the B&O protocol [5] and the Dacpo protocol [10]. All versions of UPPAAL used in the test were compiled using GCC version 2.7.2.3 and the benchmark was made on a Pentium-II/333 system with 128 MB of main memory, running RedHat Linux 5.0.

In particular, we notice that for both of the time and space usage diagrams, there is a breaking point in version 2.06 compared with the proceeding version. This is due to a number of of internal improvements in the source code including reimplementation of the main data structure i.e. the passed-list.

In the following, we mention a few recent improvements in the implementation.

[2] Version 2.00 is dated Feb 1997.
[3] Version 2.17 was released in March 1998.

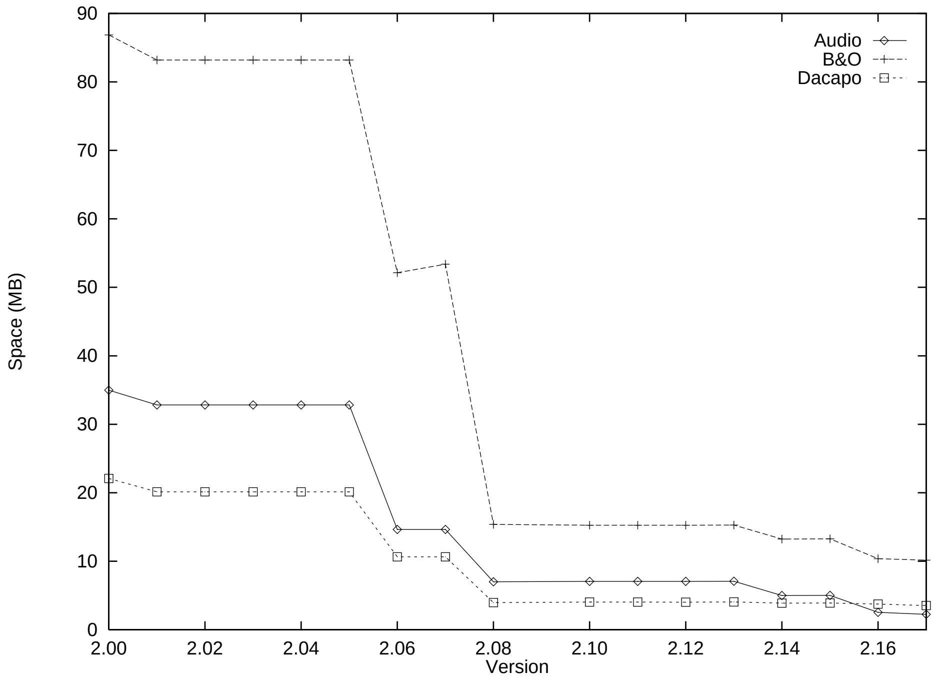


Fig. 3. Space benchmark for UPPAAL version 2.00–2.17.

Improved hash function The most critical data structure in UPPAAL is the so called passed list. It holds all symbolic states visited during the state space exploration. It is mainly to guarantee termination and to avoid repeated searching. It is often the case that a large portion of time usage is spent on searching through the list.

The passed list is implemented as a hash table with symbolic states as entries. In the previous version of the verifier, the hashing was done exclusively on the control nodes of the automata. Now the hash function also takes the values of data variables into account. The hash function assigns a unique integer to every combination of control nodes and the current values of data variables. This is possible because the number of control nodes is finite and all variables have a given domain i.e. a finite number of different values. This integer can be very large, much larger than the size of the hash table, which means that collisions can still occur even though the integer is unique for each combination.

Optimized constraint manipulation UPPAAL represents the symbolic states of a real-time system as constraints over clocks. To keep the constraint manipulation efficient, UPPAAL transforms every constraint system to a canonical form.

This transformation is the most time-consuming of all the operations on the constraints.

There is no way to check if a given constraint system already has this canonical representation which is less expensive as the transformation itself. In the previous version of UPPAAL it happened quite often that the transformation was unnecessary, but in the new version no constraint system already on the canonical form is transformed. This leads to better performance, and is one of the explanations of the big performance leap between version 2.05 and 2.06.

4 UPPAAL's New Applications

UPPAAL is frequently being applied in various case-studies, both in industry and academia. The two main application areas are real-time controllers and real-time protocols, and the purpose is often to model and analyze existing systems. However, the tool has also been applied to support design and analysis of systems under development. In particular, it has been used to support the design and synthesis of a gear controller that will operate in a modern vehicle. In the following we summarize this and some other recent applications of UPPAAL.

Recently H. Bowman et. al. applied UPPAAL to model and automatically verify an existing lip synchronization algorithm [2]. Such algorithms are used to synchronize multiple information streams sent over a communication network, in this case, audio and video streams of a multimedia application. The previously published algorithm specification is modeled and verified in UPPAAL. Interestingly, the verification reveals some errors in the synchronize algorithm, e.g. that deadlock situations may occur before pre-described error states are reached after an error.

Another application of UPPAAL in the context of audio/video protocols is reported in [5]. In this industrial application, UPPAAL is used to model and prove the correctness of a protocol developed by Bang & Olufsen. The protocol, which is highly dependent on real-time, is used to transmit messages between audio/video components over a single bus. Though it was known to be faulty, the error was not found using conventional testing methods. Using UPPAAL, an error-trace is automatically produced which revealed the error, furthermore, a correction is suggested and automatically proved using UPPAAL.

D'Argenio et. al. applied UPPAAL to the bounded retransmission protocol protocol [3, 4]. The protocol was proposed and studied at *COST 247, International Workshop on Applied Formal Methods in System Design.* It is based on the alternating bit protocol, but allows for a bounded number of retransmissions, as it is intended for use over lossy communication channels. It is reported that a number of properties of the protocol were automatically checked with UPPAAL. In particular, it is shown that the correctness of the protocol is dependent on correctly chosen time-out values.

In [10] UPPAAL is used to formally verify the so-called Dacapo protocol, a time division multiple access (TDMA) based protocol intended for local area networks that operate in modern vehicles. The study focused on analyzing the start-up of the protocol and on deriving an upper-time bound for the start-up to complete. It is proved that a network consisting of three or four nodes is guaranteed to eventually become operational and the upper time-bounds for the start-up to complete is also synthesized. Further, the start-up is shown to eventually complete for networks with a clock drift corresponding to 1/10000 between the nodes.

Another application also within the automotive industry is described in [9]. Here UPPAAL is applied to support the *development* of a system, rather than to analyze an *existing* system. The system is a prototype gear controller developed at Mecel AB in a collaboration project with the Department of Computer Systems at Uppsala University. The gear controller implements a gear change algorithm in the control system of a modern vehicle. It is designed to operate in a given surrounding environment and to satisfy a number of informal requirements prescribed by the engineers at Mecel AB. In the development, the simulator of UPPAAL was frequently used to validate the behavior of the intermediate controller descriptions. The final description was verified to satisfy 46 logical properties derived from the informally prescribed requirements.

5 UPPAAL's New Look

Apart from efficiency, the graphical user interface of UPPAAL, which allows easy editing of specifications and visualization of simulation runs, is one of the strong points of UPPAAL. In an upcoming major revision, the graphical interface will be substantially strengthened. This comes together which an extension of UPPAAL's input format, which will help to ease the job of modeling complex systems.

The current distribution of UPPAAL (see [6]) consists of several programs like `checkta` (the syntax checker) and `verifyta` (the modelchecker) which constitute UPPAAL's *engine*, i.e. the algorithmic side of UPPAAL. Further `xuppaal` is a graphical interface using XFORMS, which calls the different programs for the verification and has a built-in graphical interface for visualization of simulation runs as well as an editor for the requirements specification. AUTOGRAPH is used as a graphical editor for UPPAAL specifications, and a special program called `atg2ta` is needed to translate AUTOGRAPH's generic format into UPPAAL's more convenient `.ta` format.

A major disadvantage of this approach is that the look-and-feel of AUTOGRAPH as the graphical editor differs widely from the visualization in the simulator. Therefore UPPAAL98 (see Fig. 4) will have a completely re-designed

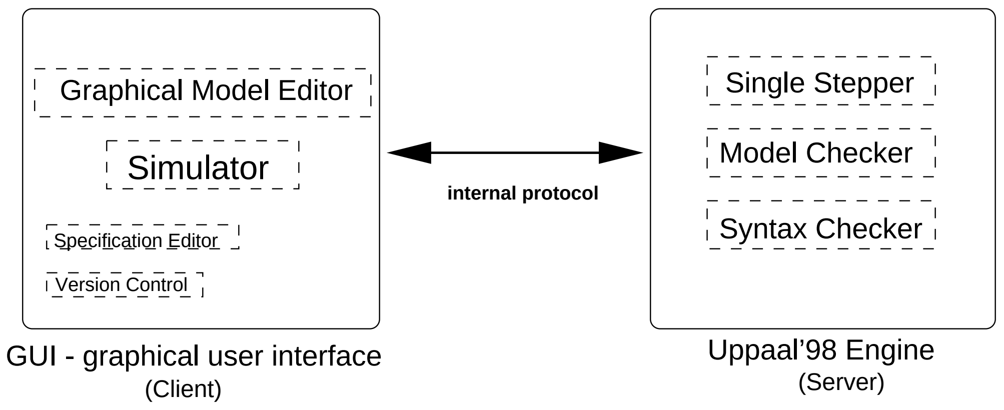


Fig. 4. UPPAAL98.

graphical user interface (GUI) unifying the graphical editor and the simulator. This new version is built as a server/client architecture, with UPPAAL's engine as the server and the GUI as the client. As integral parts of the new GUI, the graphical editor and the simulator share the same look-and-feel, which is mainly inspired by the current comfortable, easy-to-use version of `xuppaal`. Additionally the GUI will also include a specification editor and support for version control and documentation of the models and specifications. The heart of the server, which includes the syntax and the model checker, is a *single stepper* which allows to step through the reachability graph of a system. The single stepper is heavily used by the GUI's simulator. In addition to these improvements, the new approach also solves some inconsistencies between the three parts of UPPAAL's current distribution, which lead to problems in the maintenance and even in the usage.

The new GUI is written in JAVA^{TM}, making it available for all major platforms. The client/server architecture allows UPPAAL98 to be run either completely locally, client and server residing on the same machine, or to use the graphical interface and the Internet to access a host running the server. By this UPPAAL98 can be directly used via the world wide web, and it especially can be used from platforms on which an executable for the server is not available.

The new GUI also extends UPPAAL's modeling language, so that generic processes can be modeled in order to ease re-usability. The new extended format of UPPAAL's language is downward compatible with the current `.ta` format, so that existing examples will still work with UPPAAL98. The graphical information needed by the graphical editor and the simulator are now stored in a new format internal to the new GUI, so that the `.atg` files are no longer be needed. A translator from `.atg` to the new format will be available for downward compatibility.

A major feature of the simulator is the possibility to blind out parts of the system, so that in a simulation of a large system the user can concentrate on the parts he is really interested in.

At the time being, an internal version of the new GUI is up and running, which is implemented in a generic way, using design patterns from object oriented programming. This makes the GUI flexible to changes and future extensions. This version has been implemented by Carsten Lindholst and Peter Lindstrøm, two computer science students, and Carsten Weise. The server side has been implemented by Frederik Larsson. A public version of UPPAAL98 is anticipated to be available in July.

References

1. Johan Bengtsson, David Griffioen, Kåre Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. of 9th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.
2. H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic Verification of a Lip Synchronisation Algorithm using UPPAAL. In *In Proc. of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, 1998.
3. P.R. D'Argenio, J.-P., Katoen, T. Ruys, and J. Tretmans. Modeling and Verifying a Bounded Retransmission Protocol. In *Proc. of COST 247, International Workshop on Applied Formal Methods in System Design*, 1996. Also available as Technical Report CTIT 96-22, University of Twente, July 1996.
4. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the 3rd Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 416–431. Springer–Verlag, April 1997.
5. Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, December 1997.
6. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1997.
7. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL: Status and developments. Number 1254 in Lecture Notes in Computer Science, pages 456–459. Springer–Verlag, June 1997.
8. Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24, December 1997.
9. Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, March 1998.
10. Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, December 1997.
11. Wang Yi, Paul Pettersson, and Mats Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In *Proc. of the 7th International Conference on Formal Description Techniques*, 1994.

The Electronic Tool Integration Platform

Bernhard Steffen[*] Tiziana Margaria[†] Volker Braun[‡]

Extended Abstract

The Electronic Tool Integration platform (ETI) associated to STTT is designed for the interactive experimentation with and coordination of heterogeneous tools. ETI users are supported by an advanced, personalized Online Service guiding experimentation, coordination and simple browsing of the available tool repository according to their degree of experience. In particular, this allows even newcomers to orient themselves in the wealth of existing tools and to identify the most appropriate collection of tools to solve their own application-specific tasks.

The growing complexity of 'real–life' industrial software and hardware systems in fact can no longer be mastered without tool support. Thus many tools have been developed both in academia and in industry, covering different application domains and profiles. Unfortunately, understanding a tool's profile to the point of deciding whether it can be used for a specific application problem is very hard. In fact, looking for an adequate tool, one is typically confronted with a pool of alternatives, none of which matches exactly the expectations, and it is almost impossible to predict the necessary modifications, let alone estimating their cost. Thus in the course of

1. searching for candidate tools, which usually results in a rather accidental collection,
2. installing the tools and getting acquainted with their different interfaces, and
3. comparing the installed tools in the light of the own application profile and intended use

software designers all too often decide to start writing their own tool, as this gives them the reassuring feeling of full control. Consequently, the wheel is developed over and over again, not necessarily with increasing quality. The

[*]Lehrstuhl für Programmiersysteme, Universität Dortmund, Germany
[†]Fakultät für Mathematik und Informatik, Universität Passau, Germany
[‡]Lehrstuhl für Programmiersysteme, Universität Dortmund, Germany

main reason for this unsatisfactory situation is the lack of adequate decision support. In fact, none of the steps above is currently systematized:

1. surfing in the net may sound like a good solution for the first step, but the required patience is unrealistic under the common time pressure conditions, and also the use of search machines usually delivers too scattered results,
2. the acquisition and installation effort depends very much on the specific situation, but, due to the plague of unforeseable problems, it becomes usually much higher than first expected,
3. and a fair comparison is hardly possible because of strongly differing tool profiles, hardware/software constellation, etc..

Even if these problems do not strike, finding ready-to-use solutions for many practical problems would still be out of reach: experience taught for example that checking safety-critical design criteria typically requires the cooperation of *different* analysis and verification techniques, which need to be used in combination in order to overcome inherent methodological bottlenecks.

The Electronic Tool Integration platform (ETI) addresses all these concerns by offering a personalized online service providing systematic support for orientation, experimentation, and combination of all the tool functionalities integrated into the ETI repository. Complex combinations of functionalities taken from different tools can be (semi-) automatically or interactively constructed and tested by online 'meta-programming' in a simple, domain-level specification language tailored for loose specification. In particular, the burden of data format conversion needed to ensure tool interoperability is automatically taken care of within the underlying ETI platform and hidden from the users. Taken together, these features offer an evaluation and coordination support even for application experts with no programming experience.

A complete descritpion of the Electronic Tool Integration platform can be found in the first issue of Springer's International Journal on

Software Tools for Technology Transfer (STTT)

or electronically under:

http://eti.cs.uni-dortmund.de

Hans Malmkvist
Accretia AB
Project manager - LINK

The LINK experiment - A new Swedish technology transfer concept for SMEs

Background

In 1995 the Swedish National Board for Industrial and Technical Development (NUTEK), was assigned by the Department of Industry, to initiate and support a three year programme with the main purpose of establishing new technology transfer organisations supporting SMEs. Late 1996 three proposals and *experiments* were selected and supported by Nutek. The experiments were planned to run in paralell until June 1998.

One of the experiments - ”LINK”, was proposed jointly by three industry organisations and the consultancy Accretia AB. These industry organisations, together representing approximately 3 000 SMEs, are The Association of Swedish Engineering Industries (VI), The Plastics and Chemistries Federation (PoK), The Swedish Graphic Companies' Federation (GFF).
Different RTOs in Sweden, including technical universities, industrial research institutes and Swedish IRC (Innovation Relay Centres) proposed the other experiments. One experiment was dedicated to establishing a new IT-supported network of RTOs located in south west of Sweden. Another experiment was focused on creating a network of technical experts visiting SMEs and developing new methods for analysing technical needs and linking technical questions to other parts of the network. The ambition of Nutek is to integrate the results of these experiments to a new national system for technology transfer to SME.

The LINK project

The following goals were set up for the LINK-project:

- To establish a new channel to technical experts for companies with little or no experience of collaborations with RTO´s.
- To establish a computer supported service organisation for technology transfer to SMEs.
- To establish an easily accessible and personalised central helpdesk
- To practically test the technology transfer service organisation for a limited group of companies

A number of key issues to be addressed were foreseen:

- To survey and map the character of the company questions
- To develop proper functions and features of the call-centre and helpdesk (staff, IT-support, communication etc.)
- To establish agreements with the technology resources (accessibility, security, charge, etc.)
- To find an efficient way to market the new technology transfer concept

The LINK-concept

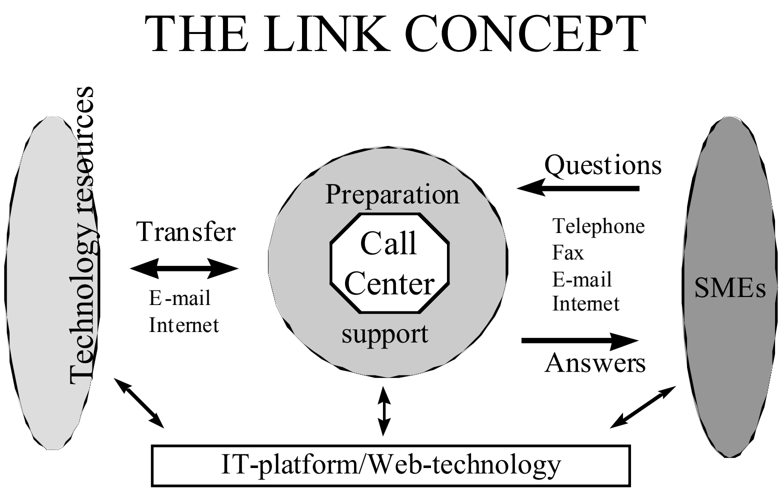
LINK is a new concept for a technology transfer service organisation specially designed for SMEs. The main purpose of LINK is to simplify access to and adoption of new technology. The purpose of LINK is not primarily to ”push” and sell RTOs current ”products” and competence. LINK is trying to support firms to identify new technologies and to give quick access to the relevant competence.

The main features of LINK could be summarised:

- Market driven service organisation for technology transfer to SME
- Technology transfer service organisation independent of technology providers
- Central, personalised helpdesk and call-centre, accessible by phone, fax, e-mail or online Internet
- Advanced, scaleable, Internet-based case handling system
- Helpdesk case preparation supporting organisation
- Agreements with and access to a network of highly skilled experts and RTO´s

The helpdesk and case preparation support

The concept of LINK is illustrated in the figure below:



Company questions are registered at the personalised helpdesk. Helpdesk is staffed and could be reached by phone during office hours. Otherwise questions could be delivered by fax, e-mail or via Internet. In many cases the original question that the company wants to pose is not properly defined. The company usually needs the dialogue with a person with a broad technical experience to be able to formulate the question properly. To support helpdesk a group of technical experts – "preparation support" (currently five people) are closely linked to and easily accessible by the helpdesk.

Case handling and IT-support

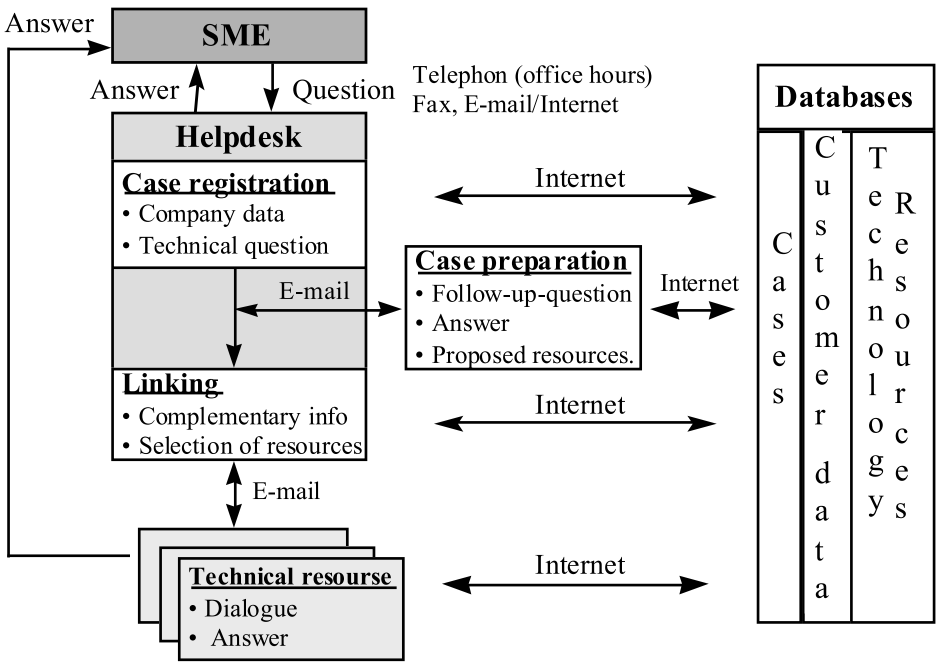
LINK is supported by a modern IT-based case handling system, designed to match the specific needs of the LINK-concept. Key features of the specification were:

- Central databases for cases, customers and technology resource specifications
- Internet TCP/IP communication
- Standard browsers as "client" SW applications
- Full flexibility in location for helpdesk, case preparation support and technical resources
- Full scalability (number of resources and people at the helpdesk location etc.)
- High security (encrypted data) and reliability (web-hotel)
- Minimum cost for system management

Initially a number of different commercially available cases handling systems were evaluated. No systems were found that could fully match the LINK specification. Therefore a new purely web-based system with an Internet/intranet TCP/IP communication and a central SQL-database was designed.

The figure below illustrates the flowchart and communication handled by the LINK-concept:

CASE HANDLING



The IT platform

The dedicated IT and case system includes the following features:

- The system is based on NT-technology. Web-server is Microsoft Internet Information Server 4.0 and database is Microsoft SQL-Server 6.5
- The user-interface is based on HTML. The pages are dynamically generated using Active Server Pages (ASP). ActiveX-components written in C++ are being used for features not supported by ASP, such as attachments and e-mails.
- All users categories connected to the same database, but with different access levels and different user-interfaces.
- Using standard HTML and Internet allows the user to use any browser on any platform connected to the Internet from anywhere in the world, and still have full access to LINK.

- The information in the database is accessible only from the Web-server and is secured by a firewall. All information between the browser and the Web-server is encrypted using SSL.

The pilot project

In November 1997 around 200 companies, all members of the three industry organisations were invited to participate in a pilot test of the LINK-concept. 150 companies were finally registered for the pilot test.

Initially an inquiry was made to find out the most vital technical areas of the pilot companies. The technology resources were recruited accordingly. During the pilot test 20 different technology resources were engaged, including three technical universities, ten research institutes, a few private companies, the Swedish Patent Office and the Swedish EC/R&D-Council. All these resources are communicating with the helpdesk via the web.

Nutek founded the pilot test as a part of the technology transfer experiment and the LINK/helpdesk services were free of charge. The technology providers were also, to a certain level, able to support the project and the companies free of charge. More extensive consulting and engagement by the technology resources were paid directly by the companies.

The experiences of the pilot test have been very positive. The questions and cases have generally been handled at a high pace and the companies have been satisfied with the answers and technical feedback. A broad spectrum of technical questions has been handled by LINK. At the end of March (5 months) approximately 100 cases were registered at the helpdesk. Over 200 individual technical questions were linked to the technology resources. There was a mixture of straightforward technical questions as well as more complex issues related to legislation, standards and development of new production processes. A fairly large number of questions were related to environmental issues.

Examples of questions:

The Association of Swedish Engineering Industries

- Solutions for rapid and contact-free temperature measurement in welders/welding machines (product development)
- The EC Environmental Marking - qualification criterias (product marketing - environmental programme)
- Parameters controlling the quality of weld joints in polythene foil (criteria for design of new welders)
- Smoothing of rough edges in heavy plates after gas cutting (quality enhancement process / work environment)

- Energy recovery in hardening plant for a powder lacquering plant (cost savings and environmental programme)
- Deformation of heavy stainless steel machinery components while finishing surfaces (production problem)
- Replacing brazing with gluing in hot water pipes (production process development).

<u>The Plastics and Chemistries Federation</u>

- Different methods and materials for sealing of electronic microcircuits (product and process development)
- Emissions to air in ejection moulding plant (environmental programme)
- High precision dimensional measurement of plastic parts considering after-mould shrinkage (process tuning)
- Cleaning of hydraulic fluids for plastic mould ejectors (improving maintenance)
- Finding electrically shielding transparent plastics for EMC-proof displays (product development)

<u>The Swedish Graphic Companies' Federation</u>

- Live experiences of fully vegetable print colours in large scale printing (process development /environmental programme)
- Laser cutting of perforations for continuous paper path in form printers (production process development).
- Using Linux operating systems in Internet servers (corporate system development for medium-sized printing firm).
- Methods for direct conversion of medical pictures (magnetic resonance or x-ray) into printable formats (pre-press process development).
- Communication system design - searching comparative development prognoses of server performance and communication rates for different network configurations (strategic process development/investments).

The goal is that the clients should perceive LINK as a support for:

- Simple access to external R&D capabilities
- Quick access to a broad spectrum of technology competence
- Identifying technical know-how and solutions to certain technology problems

However a lot of marketing and internal client support of the LINK concept is still needed. Companies need to better understand the ”product” and to get fully confident that they will achieve fast access to adequate information. There is an obvious ”threshold effect” before companies start to frequently use a new channel to technical information.
The first phase of the pilot experiment will be finalised and evaluated during the autumn of 1998. A second phase is currently being planned.

Tool-supported Software Design and Program Execution for Signal Processing Applications Using Modular Software Components

A. Sicheneder*, A. Bender*, E. Fuchs*, R. Mandl*‡, M. Mendler*, B. Sick*

Abstract

One of the most important tasks for the design and the execution of software in engineering applications is to handle the heavily increasing complexity. Especially high-sophisticated signal processing or control applications like hybrid systems in automated technical production processes contain software-intensive parts. A very successful software engineering solution to master the complexity of the algorithms and the variety of applications is the use of high-level programming systems in which predefined modules are plugged together to fit a particular application problem. This is referred to as "component-based software development." In this article, we present such a component-based tool for the specification and execution of complex signal processing algorithms. In order to optimize the software design process several methods are included to enhance efficiency and security, e.g. intuitive graphical composition facilities at an high abstraction level, reuse of parameterized basic algorithms (software components, i.e. modules), hierarchical software development, automatic data-type verification and other validation facilities. The tool offers different module libraries which are easy to use by application engineers and easy to extend by computer scientists with application-specific algorithms. Hence the tool is a powerful facility for the transfer of knowledge from specialists, who know a lot about very specialized signal processing and control algorithms, to a broad range of users by providing interfaces and the algorithms' black box behaviour. In addition, many user-defined data-types can be handled to provide a tool which is as flexible as possible without jeopardizing efficiency and security. The tool has been used successfully in several industrial applications like controlling and monitoring production processes.

Keywords: signal processing and control applications, graphical software construction, reuse of software components, software quality improvement.

1 Introduction

The development of complex signal processing applications, e.g. for automated production processes, is often difficult and time consuming [1]. A large number of computer-aided software engineering techniques have been developed to solve this problem and to reduce software development costs [2]. In particular, the methodology of *component-based software development* is well known and the basis for many programming tools. An example of a system in which this idea has been persued systematically is the METAFrame system [3]. It is a system-level programming environment for the systematic computer-aided generation, analysis, verification and testing of complex systems from repositories of predefined and reusable components. Examples for coordination tools like METAFrame are discussed in [4]. Another central software engineering technique to master the design of complex systems are *graphical user interfaces.* They are widely used not only in tool coordination frameworks but also, with increasing dedication, within formal methods CASE tools. An example is AUTOFOCUS [5] for the design and analysis of distributed systems. Many other examples can be found, e.g., in [6].

*University of Passau, Faculty for Mathematics and Computer Science, Innstr. 33, 94032 Passau (Germany), email: `{sichened,bender,fuchse,mandl,mendler,sick}@fmi.uni-passau.de`

‡Micro-Epsilon Messtechnik GmbH & Co. KG, Königbacherstr. 15, 94496 Ortenburg

The contribution of the work described in this paper is to implement a new and interesting domain-specific instance of the methodological principles of component-based programming and graphical interfacing. The tool proposed in this paper is a specialized CASE-tool for specifying and executing application-specific algorithms in the area of signal processing and control at an high abstraction level using a graphical composition facility. As such, it is different from both general-purpose CASE frameworks (which are e.g. application-independent) and coordination environments (which support the development process e.g. by mechanisms of software synthesis). The tool presented in this paper has been developed in the context of a concrete application domain. It provides for a very successful transfer of well-known software engineering technology into the area of signal processing and control.

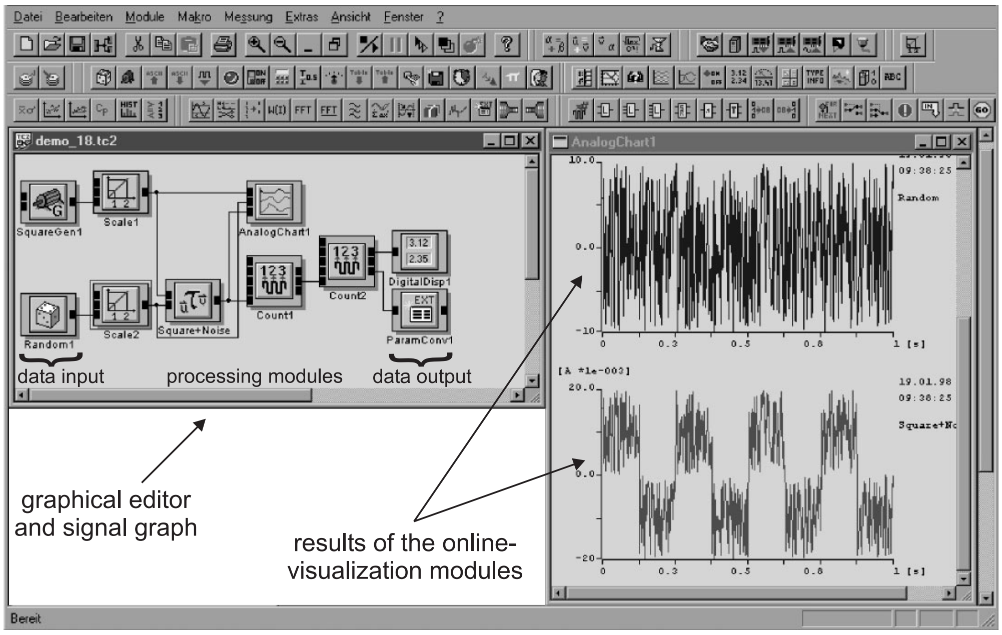


Figure 1: Graphical user interface of the tool (GUI)

On the one hand, the efficiency of the design process is enhanced by a graphical specification facility at an high abstraction level, based (essentially) on dataflow graphs. Parameterized modules serve as reusable basic algorithms such as fast Fourier transforms (FFTs), linear (e.g. PID) controllers, or finite impulse response (FIR) filters. They are encapsulated using an icon-based notation and are connected by means of a graphical editor (see figure 1). The result is a signal graph which represents a complex algorithm. The software components are stored in different application-specific dynamic libraries and can be reused with different instantiations of their parameters in similar applications. On the other hand, security of the software development process (software correctness) can be improved using high-level debugging facilities like graphical data visualization modules for the validation of the program's and modules' functionality. In addition, automatic data-type checking for connections between modules helps to avoid data incompatibilities in the early design phases.

Besides programming specific applications, it is possible to use the tool as an integrated runtime system to execute the complete algorithm. The proposed tool is applicable not only in the offline analysis of previously recorded data sets but also for online supervision and control. The application engineer can introduce graphical instruments like controllers and switches, and visualization modules to build a graphical front-end which is understandable and usable by an operator. In order to avoid undesirable usage of the GUI, editing functions and access to modules can be restricted hierarchically for different user groups.

The tool is implemented in C^{++} on different plattforms (Sun Solaris and Windows 95/NT). A commercial version called ICONNECT has been developed in cooperation with Micro-Epsilon GmbH & Co. KG. Up to now, ICONNECT has been used in several industrial applications where e.g. various physical signals are measured simultanously with different sampling rates, adaptive algorithms are needed, and actuators have to be controlled. Some examples for applications (which consist of up to 300 modules) are online measuring of thin metal or synthetic foils, monitoring the chisel depth of ABS valves, supervision of the thermal conductivity of gas concrete bricks, and controlling the thickness of flat glass for notebook displays. The tool is also used in different research projects at the University of Passau (e.g. tool condition monitoring in turning). As an offline application, the tool is used to build training sets for neural networks by extracting features from huge data sets. A demo version of the tool can be obtained via anonymous ftp from ftp://ftp.uni-passau.de/pub/local/iconnect/files.

2 Description of the tool

2.1 The Components

Figure 2 shows the two main components of the tool. In the upper part of the figure the *development environment* containing the graphical user interface (GUI) and the module libraries are seen. The lower part depicts the *runtime system* with the data input and output for the system.

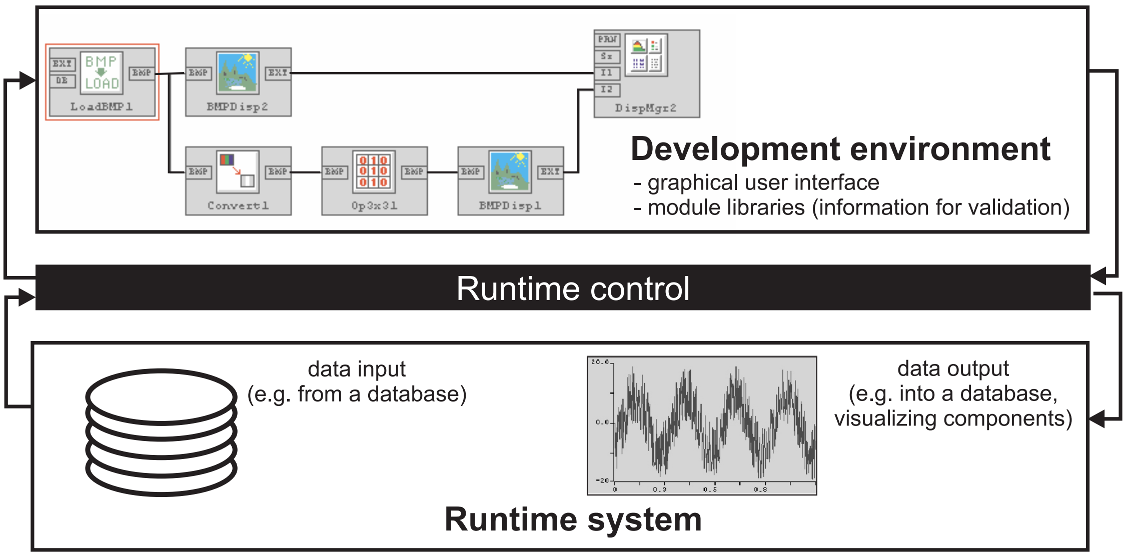


Figure 2: Components of the tool

A software solution for a specific application problem is specified interactively by the application engineer using the GUI. In this process, appropriate modules are selected from the module libraries and connected using the ports of the modules resulting in a software prototype that can be tested by means of the runtime control. Examples for modules realized in the current version of the tool are:

- data inputs and outputs: e.g. drivers for A/D- and D/A-converters, digital interfaces, virtual instruments (switches etc.), displays for offline or online visualization, function generators;
- modules for processing data in the time domain: e.g. linearization of characteristic curves of sensors, arithmetic operations, digital filters like FIR or IIR, function approximation;
- modules for signal transformation into other domains: e.g. frequency analysis, cepstral, or statistical domain;

- controllers: e.g. PID-controllers;
- special modules: e.g. database and file access, communication via computer networks (TCP/IP), function interpreter, C-interpreter;
- modules supporting the debugging process.

The *runtime control* is the central part of the tool with the main task to supervise the actual execution of the specified complete algorithm. The underlying algorithm is described in section 2.3.

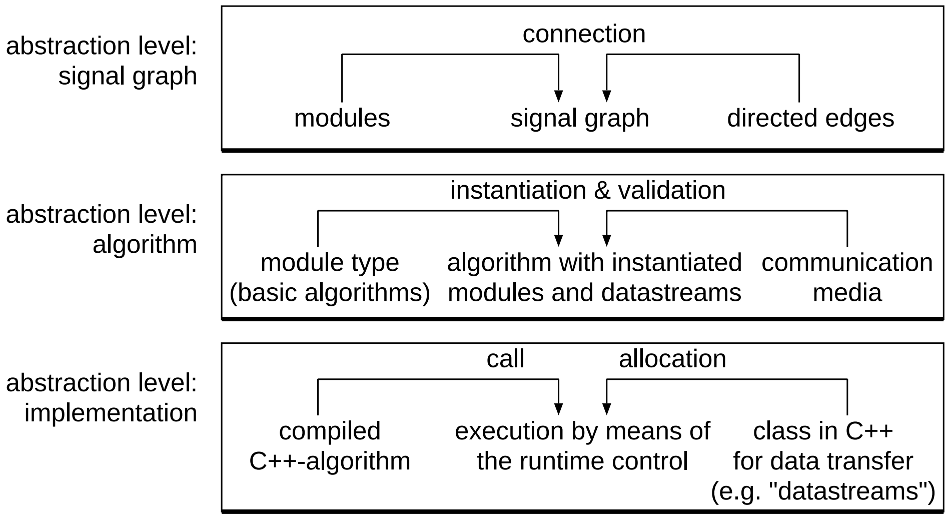

abstraction level: signal graph
connection
modules
signal graph
directed edges
abstraction level: algorithm
instantiation & validation
module type (basic algorithms)
algorithm with instantiated modules and datastreams
communication media
abstraction level: implementation
call
allocation
compiled C++-algorithm
execution by means of the runtime control
class in C++ for data transfer (e.g. "datastreams")

Figure 3: The different abstraction levels

Figure 3 outlines the different abstraction levels of the tool. At the top level is the graphical representation which abstracts from the modules' implementation and allows for an efficient fault detection. The nodes of the signal graph represent modules and the directed edges describe the data (and partially also control) flow between the modules. Some nodes represent data sources (inputs), some represent data sinks (outputs) and others correspond to parameterized basic algorithms. The directed signal graph may contain cycles to implement adaptive algorithms which are typical in signal processing and control applications.

Upon selecting a module, an instance of the algorithm is created and by connecting two modules a type-check is performed and an instance of a communication medium, e.g. a datastream is created. The lower abstraction level in figure 3 comprises the implementation of the modules and the communication media. Both are implemented in C^{++}. While executing the specified signal graph the runtime control calls the compiled algorithm if necessary and allocates communication media.

2.2 Data Model

The communication medium transports measured or simulated data as well as parameter data between the modules. Besides static parameters which are specified in parameter dialogues also dynamic and synchronized parameter adaptations based on the results of previous computations are possible. In this case parameter data are not distinguished from signal data and it is a module's task to ensure the correct interpretation of the data received through a parameter port. Beyond this, the tool provides a flexible data model which allows a variety of data-types to be associated to the ports of the modules. Data in a signal graph are processed blockwise to achieve the following objectives:

- Processing a block of data consisting of several single values reduces the communication costs. The block length is adjustable for each edge to fix the trade-off between communication overhead (long block length to maximze throughput) and time (short block length to minimize reaction time).
- Sometimes input data already are block-oriented, e.g. data from A/D-converters are stored in a buffer and transferred as a block for further processing.
- If the data to be processed are coming from a sensor observing different objects (e.g. quality control in production processes), the data recorded between two objects may be omitted; therefore the measured data of *one* object may be gathered in *one* block, thus leading to a data reduction on the one hand and to a logical grouping of data on the other.
- Many algorithms like FFT or approximation algorithms are not meaningful on single values; therefore this kind of algorithm can easily be served with blocks of data having the right amount of values needed for useful processing, e.g. powers of two for FFTs.

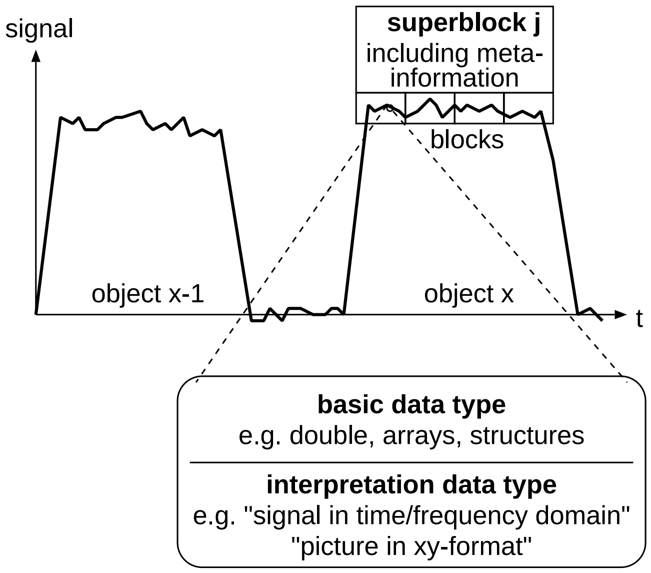


Figure 4: The data model

Blocks are organized in two levels (cf. figure 4). A *superblock* contains a set with equidistantly sampled values of data which belong together semantically (e.g. measured data of a single object). One superblock consists of several blocks each containing several elementary values. Additional meta-information (e.g. sample frequency, time stamps, physical units etc.) is valid for all blocks within a superblock. This meta-information is necessary for some basic algorithms e.g. to process signals with different sampling rates correctly. Figure 4 shows that the supported data types are also organized hierarchically in two levels: the *basic data type* describes the data format of the elementary values in the blocks and the *interpretation data type* describes the context of the data that is associated with the complete block. The type system currently implemented contains basic types such as `int`, `double`, `char`, etc., static and dynamic array types, records, a “special” type for user-defined strucures, and it distinguishes the types of elementary data, blocks, and superblocks. It has a form of parametric polymorphism in the sense of [7] and overloading. The type system is generic and extendable by an arbitrary number of additional primitive types without modification of the runtime control or the graphical user interface. Interpretation data types inform the user or modules operating on the block about the context of the signal (e.g. “signal in time domain,” “picture in xy-format”).

2.3 Runtime Control

The runtime control, which is the central component of the tool (see figure 2), is a special-purpose real-time operating system that ensures the correct execution order of the basic algorithms in a block-oriented and data-driven manner in accordance with the modules' prespecified priority [8]. Note that a simple static module processing order based on the graph's structure does not fulfill this task, because subgraphs may become disabled, depending on the values of particular control data (i.e. realizing the branching of control flow) and because modules have priorities, which may even change dynamically. Another reason is the possible cyclic processing order of subgraphs causing multiple executions of the same module. Therefore, the processing order has to be dynamic; it is based on the module status [9]. We call a module *ready for execution* if all the mandatory information (i.e. data) for the processing of the underlying basic algorithm is available.

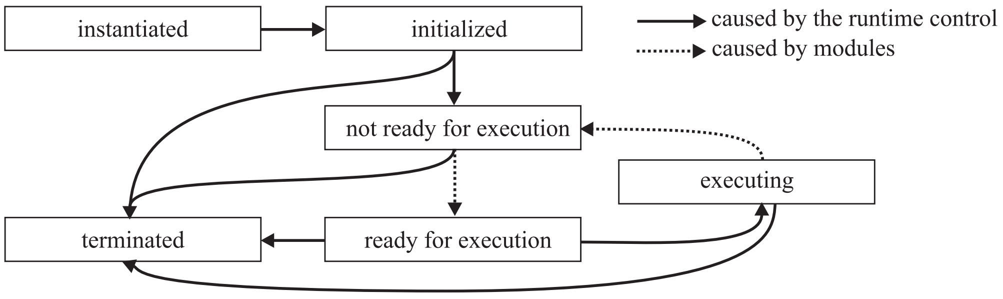


Figure 5: Module states and state transitions

In addition to *ready for execution* the runtime control distinguishes five other module states: *instantiated, initialized, not ready for execution, executing* and *terminated* (see the state transitions caused by the modules or the runtime control in figure 5). The runtime control algorithm is based on these module states and consists of four phases (cf. figure 6): First, instances of modules (from the module libraries) and communication media are generated in the *instantiation phase*. In addition, the values specified in the signal graph are assigned to parameters and macros are replaced by subgraphs. After that, specific initial actions like memory allocation are performed within the *initialization phase*. Furthermore, this phase builds a list containing all source modules ordered by priority. This list is called "source module list" (SL) in figure 6. External applications (e.g. a filter specification tool for FIR or IIR filters) can be executed for initialization purposes. The *execution phase* is divided into two subphases: first, all source modules which are *ready for execution* execute their corresponding algorithm; after each execution of a module, the successor modules (in the signal graph) are checked to find out if they are *ready for execution*. Those which are, are inserted into a waiting list (WL) according to their priority. A module is removed from the SL only if the module looses the feature of being "source," in which case it does not produce data anymore. In the second subphase, modules in WL are processed according to their priority. The generated output is passed to the successor modules and those that become *ready for execution* are inserted into WL using their priority. The module that was executed is removed from WL. Then again the next module is picked from WL and the process repeated. When WL is empty the execution continues with SL. This alternate processing of SL and WL is repeated until SL is empty. The processing is also stopped by an explicit request by a module to end the execution, by an external user request, or after a runtime error. The final *termination phase* which is executed in all cases of termination, closes data files, frees memory etc. Again it is possible to call external applications, e.g. tools for offline visualization of processed data.

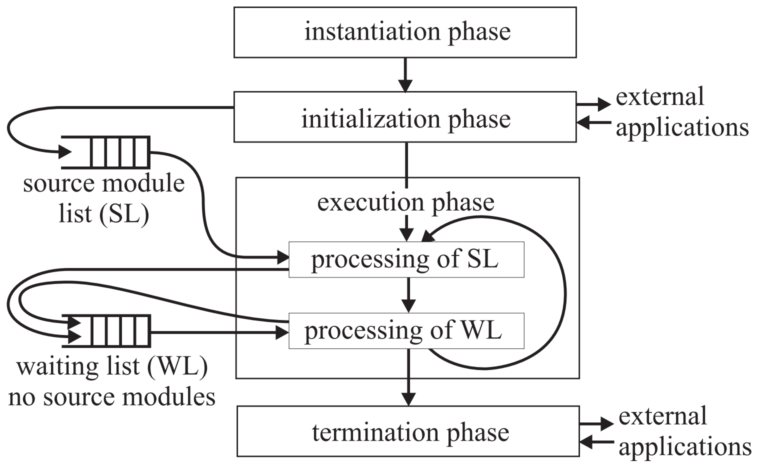


Figure 6: Algorithm of the runtime control

3 Software Development Efficiency and Security

For ordinary general-purpose programming languages the goals of design efficiency and design security are difficult to achieve, for they require a substantial amount of design restrictions which contradict the essential idea of a general-purpose language which is to be universally applicable. In domain-specific programming systems, however, the situation is quite different. In such dedicated systems, like the one considered here, there is sufficient focus to take over from the user a lot of the difficult and error-prone programming tasks, that are common for the given application area. This eliminates unnecessary programming freedom and increases design efficiency and security. Our system adopts this strategy stressing the following particular features:

- Graphical User Interface
- Parameterization and Modularity
- Type Checking
- High-level Data-Flow Model
- Integrated Runtime System and Debugging
- Abstraction and Hierarchy

Although these are mostly well-known software engineering methods, they have not as yet fully found their way into commercial tools for programming signal processing and control applications. Our tool incorporates these features and demonstrates their benefit in a specific application context. In the concrete projects that we have run in cooperation with MicroEpsilon we found that more than 90% reduction in development time — compared to purely C-programmed solutions — can be achieved by using our tool.

Graphical User Interface As mentioned before the efficiency of the software design process is considerably enhanced using the graphical specification feature. The graphical user interface (GUI) is easy to use and its interactive "push-button-programming" style reduces considerably the coding effort as well as it eliminates annoying syntactic errors. Even more important for this specific engineering context is the benefit of the GUI for technology transfer: Through the GUI the tool may be used by application engineers with little or no skills (or interest) in software programming and algorithm design. Through the GUI a large amount of irrelevant complexity may be hidden from the user which is very important from the company's point of view.

Parameterization and Modularity Application engineers are most efficient in developing an appropriate solution in an explorative and evolutionary fashion by modifying previous solutions (subgraphs or single modules) in terms of different parameters, or relatively small changes or extensions to the algorithms (software-reuse). Adjustable parameters support the technology transfer from the software engineer to the application engineer. In our specific application domain parameterization may arise in non-trivial form, in the sense that e.g. the filter coefficients defining a particular FFT module may themselves be computed, interactively with the user, through specialised initialisation modules.

Type Checking To identify potential design errors a type checking algorithm is incorporated into our tool. A static type analysis (see also sec. 2.2) with a simple form of polymorphism [7] ensures correct typing thoughout the signal graph. In this way runtime errors due to inconsistent use of data and modules are excluded already at construction time. There is a notion of an interpretation data type, which is special in that type violations of interpretation data types only produce warnings but do not stop the construction of the signal graph.

High-level Data-Flow Model The representation of the functionality on an high abstraction level is based on a simple dataflow model. This modelling paradigm is superior in many aspects to flow charts, Petri nets, or state diagrams [10]. The concept of dataflow graphs makes programs easier to construct while still preserving their natural understandability. This behavioural abstraction, together with the hierarchical decomposition feature, supports the user in specifying complex applications which result in very large signal graphs. Some commercial applications developed with our industrial partner involve signal graphs with up to 300 nodes, each of which representing a complex module.

Integrated Runtime System and Debugging The integrated runtime system, which is basically a specialised real-time operating system, provides for rapid prototyping and simulation facilities. The tool can also be run as an embedded system in on-line control applications, or used for emulation. The expensive error tracing within the whole graph is dramatically reduced due to the modular character of the specification. Special modules are available for debugging. Examples are data visualization modules, monitoring the result at an output port of any basic algorithm, or the so-called "pause"-module which allows breaks in the algorithm's execution, much like the breakpoints used in C-debuggers. However, here, debugging is done at the module level rather than inside the modules at the algorithmic level.

Abstraction and Hierarchy The tool can be used efficiently on different levels of abstraction by a broad range of people with different programming skills. Access to modules and editing functions is hierarchically restricted for these user groups. Users like the application engineers are not interested in implementation details. They can specify their application on an high abstraction level focusing on the important aspects of their specific signal processing problem. This group of users needs parameterized basic components within the GUI. Algorithm developers, on the other hand, are more familiar with the specific problems of algorithm and software development. They want to implement specific basic components on a lower abstraction level, but they are not interested in such aspects as communication between the modules or prioritized real-time scheduling. A software engineer is only responsible for the algorithms' I/O-behaviour without integration aspects. In this way the tool supports the technology transfer between the computer scientists (developing the tool), the software engineers (programming the modules), the application engineer (using the modules to build an application system), and finally the operator (using the application system).

4 Related Tools in the Area of Signal Processing and Control

Graphical specification based on dataflow diagrams is a well known technique in the area of control engineering, measurement technology, signal or image processing and multimedia applications. Following Schreier [11] we mention the most popular tools in this field. LabVIEW from National Instruments was originally developed as a graphical user interface for instruments [12]. It provides a powerful graphical programming facility for the definition and connection of so-called "virtual instruments." Other tools that support graphical specification are Hewlett Packard's HP Vee [13], DASYLab from Datalog Corp. [14] or in multimedia applications the tool MET^{++} developed at the University of Zürich [2]. Most of these tools, as far as they are specific to the signal processing domain, satisfy *some* but none of them fulfills *all* of the following requirements:

- block-oriented data processing with individually adjustable block length for each edge in the signal graph,
- processing of signals with different sampling rates (e.g. using several sensors),
- synchronisation of different data streams (e.g. from different subgraphs or with different sampling rates),
- processing of signal graphs containing cycles (e.g. for adaptive algorithms),
- parallel or distributed (over a network) execution of several signal graphs,
- obeying of weak real-time requirements (e.g. actions within the GUI such as moving or resizing of windows must not stop the continuous signal processing),
- easy extendability of the system by new modules or new data types without compilation of the main components of the tool,
- clear representation of complex programs by hierarchical decomposition and rectilinear wiring of the egdes.

The proposed tool fulfills all of these requirements and therefore, it is a very powerful tool. There exist other approaches that deal with the (automatic) compilation of graphic dataflow specifications into programs (e.g. C-code) [15] but these do not integrate a runtime system. One of the most popular tools in this area is Ptolemy [16] which was primarily developed for the design and simulation of multiprocessor systems or DSPs. The kernel of the system is the basis of further work (e.g. tool for the simulation of optical communication networks or a tool for the rapid prototyping of special processors [17]) at several international universities and companies.

5 Conclusion and Future Work

The presented modular tool is a good example for technology transfer from academic research to commercial applications following the component-based programming paradigm. The tool offers a graphical specification facility which leads to considerable economic benefits and fulfills the requirements of different user profiles in the following way:vi

- Complex signal processing applications can be implemented and documented in a single working cycle.
- The software development process is safer; yet solutions can be built faster.
- Even very complex applications are understandable and known solutions can easily be adjusted for reuse in new applications.
- Well-tested modules (provided by extendable module libraries) can be reused in specific applications; furthermore new libraries can be created and integrated into the tool on demand by software experts.
- The editing of signal graphs can be disabled to prevent the system from unauthorized use e.g. in a control application.

- Interfaces to databases or networks can be used to analyse data offline.
- Several mechanisms help to uncover and avoid bugs already at an early stage in the specification phase (e.g. type-checking).

In future work we plan to extend the tool in two directions. In one project we aim to replace the current run-time system by a distributed real-time operating system to support a more fine-grained distributed execution of signal graphs. Currently, only complete signal graphs may be run in a distributed fashion (dynamic scheduling). In another project we plan to extend the automatic type checking algorithm to a more powerful static validation method. By enriching the type system in a suitable way, we hope to include the verification of static module parameters as well as of some aspects of reactive and quantitative real-time behaviour.

References

[1] J. Kodosky, J. MacCrisken, and G. Rymar, "Visual Programming Using Structured Data Flow," in *Proceedings of the 1991 IEEE Workshop on Visual Languages*, (Kobe, Japan), pp. 34–39, 1991.

[2] P. Ackermann, *Developing Object-Oriented Multimedia Software.* Heidelberg: dPunkt (Verlag für digitale Technologie GmbH), 1996.

[3] B. Steffen, T. Margaria, A. Claßen, and V. Braun, "The METAFrame'95 environment," in *Proceedings CAV'96* (R. Alur and T. Henzinger, eds.), pp. 450–453, Springer, 1996. LNCS 1102.

[4] A. Claßen, *Component Integration in METAFrame.* PhD thesis, University of Passau, Faculty for Mathematics and Computer Science, 1997.

[5] F. Huber, B. Schätz, and G. Einert, "Consistent Graphical Specification of Distributed Systems," in *Proceedings of FME'97: Industrial Applications and Strengthened Foundations of Formal Methods* (J. Fitzgerald, C. B. Jones, and P. Lucas, eds.), pp. 122–141, September 1997.

[6] B. Steffen, W. Cleaveland, and T. Margaria, eds., *International Journal on Software Tools for Technology Transfer*, vol. 1. Springer, 1997.

[7] R. Milner, "A theory of type polymorphism in programming," *J. Comp. Sys. Sci.*, vol. 17, no. 3, pp. 348–375, 1978.

[8] H. Nömmer, "Spezifikation und Implementierung einer Entwicklungsumgebung für Signalverarbeitungsalgorithmen mit Ablaufsteuerung zur datenflußgetriebenen Bearbeitung auf der Basis parametrisierter Module," diploma thesis, University of Passau, 1997.

[9] H. Nömmer, E. Fuchs, B. Sick, and R. Mandl, "Entwicklung und Ablauf objekt-orientierter Echtzeitsoftware auf der Basis parametrisierter Algorithmenmodule," in *Echtzeit 97*, (Wiesbaden, Germany), 1997.

[10] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," *IEEE Computer*, vol. 15, pp. 26–41, February 1982. (special issue on data flow systems).

[11] P. G. Schreier, "Users adopt new technologies, return to familiar suppliers," *Personal Engineering*, pp. 22–25, January 1997.

[12] L. K. Wells and J. Travis, *Labview for Everyone: Graphical Programming Made Even Easier.* Prentice-Hall, 1996.

[13] Hewlett Packard, product page of HP Vee, *URL: http://www.hp.com/go/hpvee.*

[14] N. Trevarthen and S. Leigh, "16 and 32 Bit Data Acquisition Systems with Multiboard Drivers," *Adept Scientific*, July 1997.

[15] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs.* Boston et al: Kluwer Academic Publishers, 1996.

[16] Official Web-Site of the Ptolemy Project: http://ptolemy.berkeley.edu, *The Almagest: A Manual for Ptolemy.* Vol. I–III.

[17] M. A. Richards, A. J. Gadient, and G. A. Frank, *Rapid Prototyping of Application Specific Signal Processors.* Boston: Kluwer Academic Publishers, 1997.

C-Mix: Making Easily Maintainable C-Programs run FAST

The C-Mix Group[*], DIKU, University of Copenhagen

Abstract

C-Mix is a tool based on state-of-the-art technology that solves the dilemma of whether to write easy-to-understand but slow programs or efficient but incomprehensible programs. C-Mix allows you to get the best of both worlds: you write the easy-to-understand programs, and C-Mix turns them into equivalent, efficient ones. As C-Mix is *fully automatic*, this allows for faster and more reliable maintenance of software systems: system programmers need not spend hours on figuring out and altering the complicated, efficient code.

C-Mix is a *program specializer:* Given a program written in C for solving a general problem, C-Mix generates faster programs that solve more specific instances of the problem. Application areas include model simulators, hardware verification tools, scientific numerical calculations, ray tracers, interpreters for programming languages (Java bytecode interpreters, task-specific interpreters), pattern matchers and operating system routines.

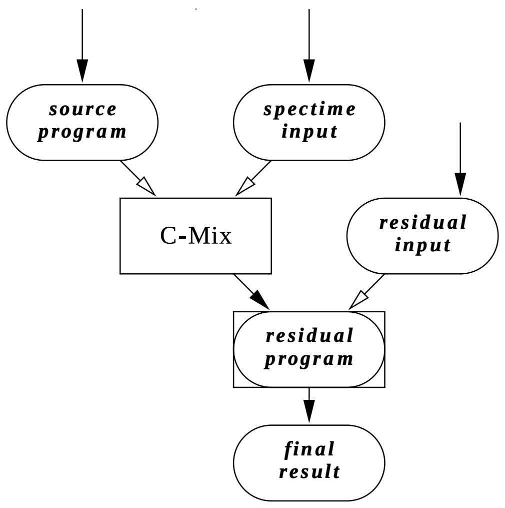
C-Mix currently runs on Unix systems supporting the GNU C compiler, and treats programs strictly conforming to the ISO C standard. Future releases of C-Mix are intended to run on a variety of platforms.

1 Program specialization and partial evaluation

C-Mix performs program specialization by a technique called *partial evaluation* (Jones, Gomard, & Sestoft, 1993). Given a source program and some of its input (the *specialization-time* data), it produces a so-called *residual* or *specialized* program. Running the residual program on the remaining input (the *residual-time* data) yields the same results as running the original program on all of its input; but potentially faster.

We use the word *spectime* to denote values and variables that are present at specialization time. The other values and variables in the program are *residual.*

C-Mix generates specialized versions of functions, unrolls loops, unfolds function calls and pre-computes expressions and control constructs that does not depend on residual data. These transformations are similar to what optimizing compilers do, but since C-Mix takes some of the program's input into account, it can potentially do better. In addition, partial evaluation is based on inter-procedural analyses (including inter-procedural constant propagation), whereas most optimizing compilers only use intra-procedural analyses.



Generality versus efficiency and modularity: One often has a class of similar problems which all must be solved efficiently. One solution is to write many small and efficient programs, one for each. Two disadvantages are that much programming is needed, and maintenance is difficult: a change in outside specifications can require every program to be modified.

Alternatively, one may write a single highly parameterized program able to solve any problem in the class. This has a different disadvantage: *inefficiency.* A highly parameterized program can spend most of its time testing and interpreting parameters, and relatively little in carrying out the computations it is intended to do.

[*]Contact: Arne Glenstrup, Henning Makholm, or Jens Peter Secher: {panic,makholm,jpsecher}@diku.dk

Similar problems arise with highly modular programming. While excellent for documentation, modification, and human usage, inordinately much computation time can be spent passing data back and forth and converting among various internal representations at module interfaces.

To get the best of both worlds: write only one highly parameterized and perhaps inefficient program; and *use a partial evaluator to specialize* it to each interesting setting of the parameters, automatically obtaining as many customized versions as desired. All are faithful to the general program, and the customized versions are often much more efficient. Similarly, partial evaluation can remove most or all the interface code from modularly written programs.

C-Mix supports code that is *strictly conforming* to the ISO C standard. That is, code that depends on the details of data representation on the target platform (*e.g.*, mixes pointers and integers) will not be handled by C-Mix. However, in well-designed code these platform dependencies will usually be isolated in certain source files which can readily be excluded from the set of files that C-Mix specializes.

2 A simple example: specializing `printf` and `power`

Consider a source program containing a simplified implementation of the C library functions for computing the power function and formatting a string. Specializing the statements `v[0] = power(n,x); printf("Power = %d\n", v);` to fixed n=5 will yield the residual statements `v[0] = power_5(x); printf_res(v);`, where functions `power_5` and `printf_res` are defined in the residual program:

Source Program

```
void printf(char *fmt, int *values)
{ /* print formatted data */
  int i, j;
  /* Parse the format string */
  for (i = j = 0; fmt[i] != '\0'; i++) {
    if (fmt[i] != '%')
      putchar(fmt[i]);
    else {
      i++;
      switch (fmt[i]) {
      case 'd': /* %d: output int */
        sys_printf("%d", values[j]);
        j++; break;
      case '%':
        putchar('%'); break;
      default:
        putchar(fmt[i]); break;
  }}}
}

int power(int n, int x)
{ /* Return the nth power of x */
  int pow;

  pow = 1;
  while (n > 0) {
    pow = pow * x; n--;
  }

  return pow;
}
```

Residual Program

```
void printf_res(int *values)
{

  putchar('P');
  putchar('o');
  putchar('w');
  putchar('e');
  putchar('r');
  putchar(' ');
  putchar('=');
  putchar(' ');
  sys_printf("%d", values[0]);
  putchar('\n');






}

int power_5(int x)
{
  int pow;

  pow = 1;
  pow = pow * x; pow = pow * x;
  pow = pow * x; pow = pow * x;
  pow = pow * x;

  return pow;
}
```

Note that the variable `n` has been specialized away: The calls to the specialized versions of `power` and `printf` now only take the residual variable as argument. The loop in the `power` function has disappeared and 5 multiplications remain. The loop in the `printf` function interpreting the format string has also been specialized away leaving only the code with side effects.

This residual program is much faster than the original general one, since the testing and updating of the loop variables has been eliminated.

3 A more subtle example: binary search

Binary search is an algorithm for detecting whether a sorted array contains a given element `x`. The problem is solved by keeping track of a range within the array in which `x` must lie, if it is there at all. Initially the range is the entire array, and in each iteration the range is halved. By comparing the middle element of the range to `x`, it is decided whether to continue looking in the upper or lower half. The process continues until `x` is found or the range becomes empty. The time to search an array of size n is $O(log(n))$.

Instead of representing the range by its lower and upper values as in the "classical" version, the range is represented by its lower value `low` and an increment `mid`, thus *separating* the computations for the position of the range and its size. As usual, we ensure that the increment takes only powers of 2 as values.

```
int bsearch(int x, int *a)
{
    int mid = 512;
#pragma residual: bsearch::low
    int low = -1;

    if (a[511] < x)
        low = 1000 - 512;
    while (mid != 1) {
        mid = mid / 2;
        if (a[low + mid] < x)
            low += mid;
    }
    if (low + 1 >= 1000 || a[low+1] != x)
        return -1;
    return low + 1;
}
```

Specialization with respect to "no" spectime input may seem useless, but notice that the size of the array (1000) is "hard-coded" into the program. Thus, some data is present in the function already at specialization time.

The variable `low` can take 1000 different values, depending on the value of `x` and the array contents, so specialization with respect to this variable is likely to produce an enormous amount of residual code. To avoid this we can insert the C-Mix directive `residual` in the source program, which prevents C-Mix from trying to keep track of `low`'s values at specialization time. Note that this does not make `mid` residual, exactly because `mid` does not depend on `low` like in the "classical" implementation of the binary search algorithm.

Running this example through C-Mix we find that the residual program is small, and the division calculations involving `mid` have all been specialized away:

```
int bsearch(int x, int *a)
{
    int low;

    low = -1;
    if (a[511] < x)       low  = 488;
    if (a[low + 256] < x) low += 256;
    if (a[low + 128] < x) low += 128;
    if (a[low + 64] < x)  low += 64;
    if (a[low + 32] < x)  low += 32;
    if (a[low + 16] < x)  low += 16;
    if (a[low + 8] < x)   low += 8;
    if (a[low + 4] < x)   low += 4;
    if (a[low + 2] < x)   low += 2;
    if (a[low + 1] < x)   low += 1;
    if (low + 1 >= 1000 || a[low + 1] != x) return -1;
    else                                    return low + 1;
}
```

Had we specialized the program without the `residual` directive, the result would have been a rather large program (around 114 Kb), because the function is specialized with respect to many different values of the two spectime variables. The first few lines of the residual `bsearch` function look like this:

```
int bsearch(int v1, int *(v2))
{
  if (((v2)[511]) < (v1)) {
    if (((v2)[744]) < (v1)) {
      if (((v2)[872]) < (v1)) {
        if (((v2)[936]) < (v1)) {
          if (((v2)[968]) < (v1)) {
            if (((v2)[984]) < (v1)) {
              if (((v2)[992]) < (v1)) {
                if (((v2)[996]) < (v1)) {
                  if (((v2)[998]) < (v1)) {
                    if (((v2)[999]) < (v1)) {
                      return -1;
                    } else {
                      if ((0) || (((v2)[999]) != (v1))) {
                        return -1;
                      } else {
                        return 999;
                      }
                    }
                  } else {
                    if (((v2)[997]) < (v1)) {
                      if ((0) || (((v2)[998]) != (v1))) {
                        return -1;
                      } else {
                        return 998;
                      }
                    }
```

The table below shows the run-times of the various versions of binary search we have seen. The function was called 1000 times. The runtime is shown in CPU user seconds, and the code size is the size of the object file measured by `size`. The speedup is the ratio between the running time for the original and the specialized program; similar for the code size blowup. The programs were compiled with the Gnu C-compiler `gcc` with option `-O2`, and the programs were executed on an HP9000/735.

Program	Runtime (sec)			Code size (bytes)		
	Orig	Spec	Speedup	Orig	Spec	Blowup
bsearch2 (residual `low`)	7.7	5.2	1.5	96	200	2
bsearch1 (spectime `low`)	7.7	4.6	1.7	96	24520	255

The cost of forcing `low` to be residual is a increase in running time by 0.6 seconds (or 13 percent), which is acceptable since the specialized program is 100 times smaller! Careful inspection of the programs and other experiments show that the size of the residual program in the first example is proportional to the size of the array, whereas it is proportional to the logarithm of the size in the second example.

This experiment shows both a strength and a weakness of partial evaluation. It is pleasing that we can achieve a good speedup despite the modest code blowup, but it requires some insight to discover that the variable `low` should be residualized. However, once the `residual` directive has been added, the program can be *automatically* specialized whenever the initial value of `mid` and the array size changes.

Finally it is worth mentioning that it is also possible to specialize binary search with respect to a known array. In that case even higher speedups can be expected. Experiments show speedups of 2.8 and 3.7 and code blow-ups of 385 and 363 (Andersen, 1994). Code blowup of this size is quite acceptable, if it results in such good speedups, and the table is not too big.

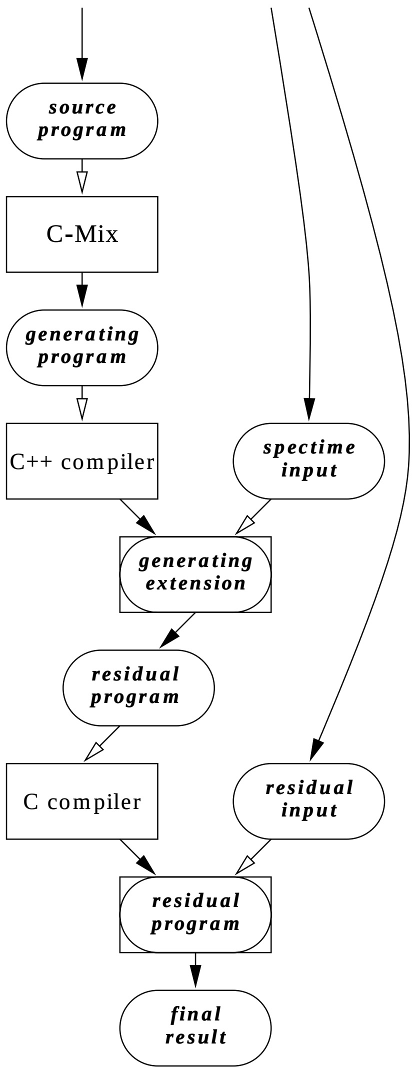
4 Tool structure

C-Mix works together with existing C and C++ compilers when producing the residual program. The C-Mix core system produces from the source program a C++ program text called the *generating extension.* When the generating extension is run (after having been compiled with a C++ compiler) it reads in the specialization-time inputs and emits the final C source for the residual program.

There are several advantages of this approach. The first is that since the spectime actions are performed by the generating extension, C-Mix does not itself have do know how to execute them—that is left for the C++ compiler to decide. (Many earlier partial evaluators were "monolithic" and needed in effect to contain an interpreter for the source language to be able to perform specialization-time actions).

Second, the user of C-Mix can link the generating extension together with object code of his own. That means that the user can provide functions that can be called at specialization time, without C-Mix having to analyse and understand their source code. This is particularly convenient if the spectime input has a high-level format that have to be parsed before it can be used. C-Mix performs rather badly when faced with the output of parser generators such as `yacc`—but since none of the parser code is supposed to be present in the residual program anyway, the parser can be compiled as-is and linked into the generating extension without confusing C-Mix.

After having run the generating extension the finished residual program has to be compiled with a normal C compiler. Our experience is that the transformations done by C-Mix works well together with optimizing C compilers. Rather than performing optimizations which the compiler would have done in any case, partial evaluation seems to open up new possibilities for compiler optimization. This is mainly because the control structure of the residual program is often simpler than that of the source program, and because, generally, the residual program contains less possibilities of aliasing.



5 Conclusion

C-Mix is a partial evaluator for specializing C programs. It is automatic and handles all C programs that are strictly conforming to the ISO C standard. Using C-Mix one can obtain *efficient* programs from larger but *easy-to-read* programs, and in this way reduce the slow and error-prone job of maintaining highly efficient but highly unreadable software.

C-Mix is available free of charge from DIKU. The project's home page is http://www.diku.dk/research-groups/topps/activities/cmix.html

References

Andersen, L. (1994). *Program analysis and specialization for the C programming language.* Unpublished doctoral dissertation, DIKU, University of Copenhagen. (DIKU report 94/19)

Jones, N. D., Gomard, C. K., & Sestoft, P. (1993). *Partial evaluation and automatic program generation.* Prentice-Hall.

Rapid Prototyping with APICES

Ansgar Bredenfeld
GMD
Institute for System Design Technology
D-53754 Sankt Augustin, Germany
bredenfeld@gmd.de
http://set.gmd.de/APICES

APICES is a tool for very rapid development of software prototypes and applications. It offers special support for technical applications dealing with network-like structures. Network-like structures are modelled with predefined object-oriented building blocks so-called graph pattern*.*
Software development with APICES starts with an object-oriented model of the application. This application model is the input for our code generators. They generate a core of the application consisting of a class library (C++) with an optional interface to an object-oriented database. This core is automatically embedded in the script language Tcl and may be extended with application code in C++ or Tcl.

Many technical applications deal with network-like data. Some examples are schematic editors (block diagrams), simulators (component networks), digital signal processing tools (signal and data flow graphs), hardware/software co-design tools (process networks, control data flow graphs), or workflow applications (task graphs). Our tool is tailored to support rapid prototyping of such applications.
APICES is based on state-of-the-art object-oriented modelling constructs - object types, inheritance, data encapsulation, typed attributes, different types of relationships (association, aggregation). Each modelling construct has a set of generic manipulation methods, e.g. each relationship type has a set of access methods which allows to manipulate, access and navigate through the elements of a relationship. To this extend, APICES is comparable to off-the-shelf modelling tools with code generation capability.

1. Graph Pattern

In addition to object-oriented modelling constructs, APICES offers re-usable building blocks to model complex network-like structures. These building blocks are called *graph pattern* and are the specific strength of our tool. Graph pattern allow very rapid prototyping of technical applications. We support various variants of graph pattern. They cover simple, flat network structures as well as more complex hierarchical network structures. Each graph pattern variant has a set of methods which are usually needed to construct, destruct, access, navigate or transform network structures. Graph pattern methods offer functionality for composition, for connectivity, for hierarchy handling, and for simulation.

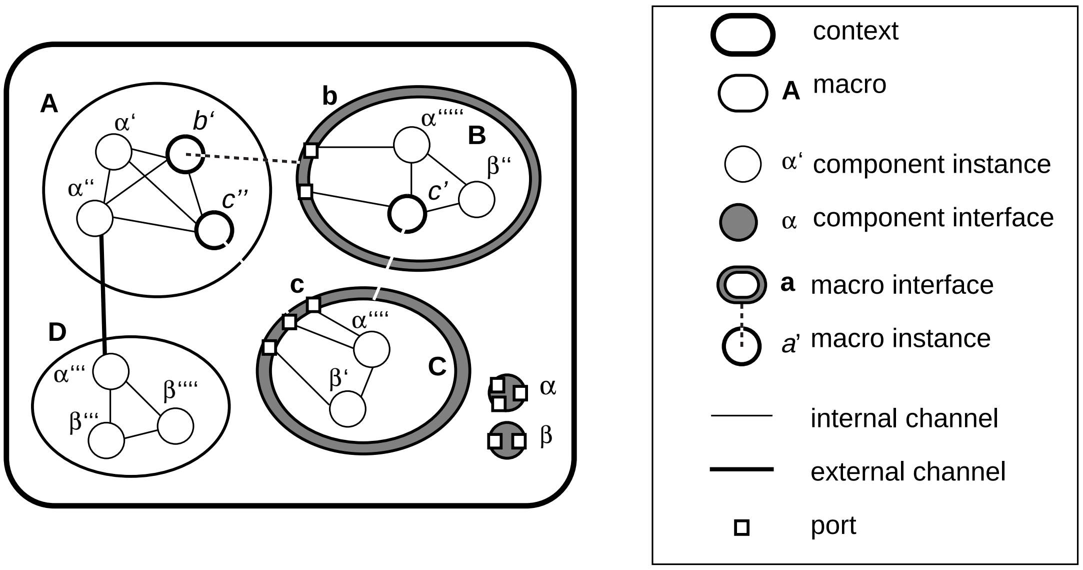


Figure 1: Elements of Graph Pattern

The basic elements of graph pattern are *components*. Components have a list of directed ports and a behaviour. *Ports* are elements of the interface of a component. Ports may be inputs, outputs or bidirectional. Ports are connected with each other by *(multi-)channels*.

- Flat network structures

The most simple graph pattern variant offers components needed to model flat, non hierarchical network structures. An example is a netlist without hierarchy which is the core structure of many simulators. Components which describe interfaces are called *component interfaces*. They are type descriptions (prototype components, template components). Component interfaces are terminal elements, i.e., they can not be further decomposed. *Component instances* are occurrences of component interfaces. The component interface determines the structure and the behaviour of a component instance. All instances of an interface have the same port structure. In addition, the behaviour of an instance is given by the interface. Component instances are connected at their ports with channels.

- Hierarchical network structures

A more complex graph pattern variant provides elements to model hierarchical network structures. Hierarchy is modelled by a macro mechanism. *Macros* aggregate and encapsulate several component instances. They contain the component instances with their ports and the channel connections between them. Macros are associated with two specialized components - *macro interfaces* and *macro instances* - to construct hierarchies. The *macro interface* is a component which describes the external (formal) port structure of the macro. The macro interface defines the structure and the behaviour of a macro instance. *Macro instances* are occurrences of macro interfaces. Hierarchical structures are constructed by recursively creating macro instances within macros. All macros, which belong to a hierarchy, are aggregated by a *context*.
Figure 1 shows an example of graph pattern elements and their contextual relationships. Ports of component instances and macro instances are omitted to improve legibility.

2. APICES tools

The graphical model editor (figure 2) allows to specify and document the core model of an application. Graph pattern are used to model network-like structures typically needed by a technical application. Graph pattern are reused by binding template object types of a graph pattern variant to the object types of the application.
The code generators of APICES are able to generate the following implementations from this high-level application model.

• C++ class library

The model of the application is compiled to a C++ class library. The class library contains a C++ class hierarchy implementing the model and access methods which are needed to handle objects, attributes, relationships and the elements of graph pattern. This library is the core of the application and can directly be linked to existing or new C++ programs. Optionally, an interface to an object-oriented database is generated.

• Extended script language Tcl

The methods of the generated C++ library are automatically embedded in the script language Tcl. This embedding offers all generated methods for interactive and interpreted access. This is an important feature for rapid prototyping and allows to develop algorithms first in Tcl and then migrate to C++ if necessary.

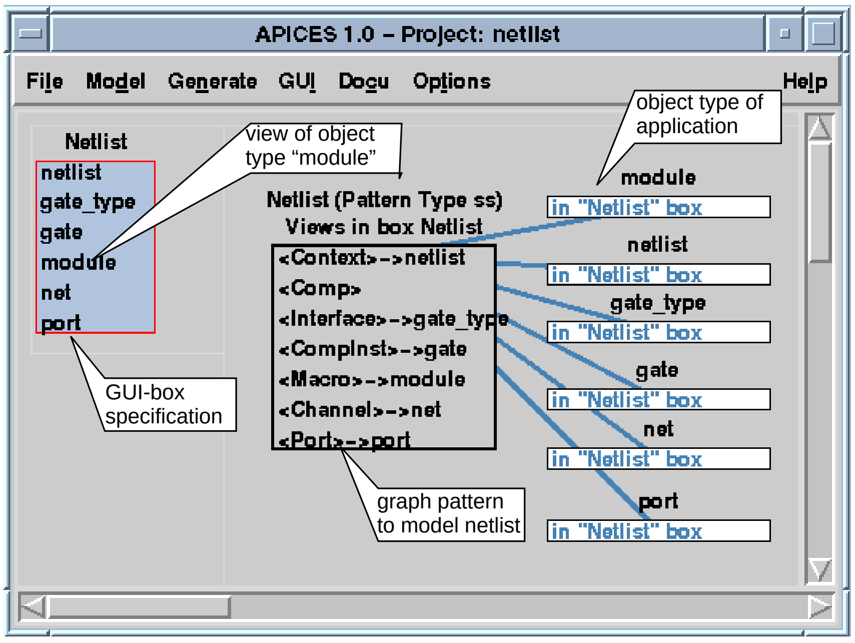


Figure 2: Application model of a simple tool operating on netlists including the GUI specification of the tool

- Graphical interface

The model editor not only allows to specify the model of the application but also a graphical user interface for the modelled application. The user interface is primarily used during rapid prototyping for test and debug purposes. Nevertheless, it can be used as graphical end-user interface for an application.
The left part of figure 2 shows a specification of a *box*. A box is a rectangular part of the canvas at the user interface. Each box is able to display several objects as *object views*. The mapping between object types and object views is done via the model editor. The visualization of object views is configurable. If graph pattern variants have to be visualized, a predefined default view for each element of a graph pattern is generated.

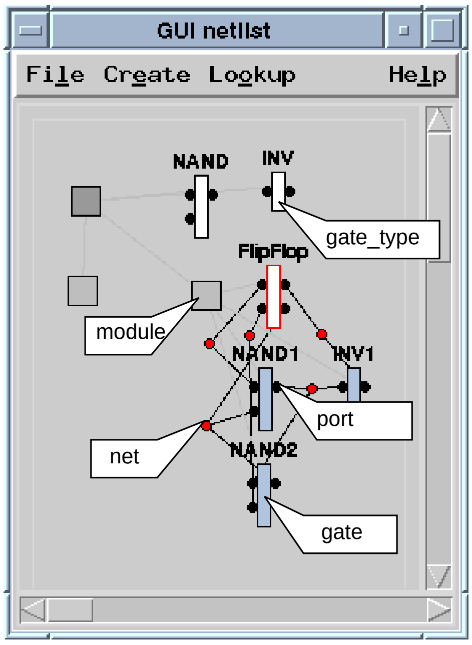


Figure 3: Application GUI generated from model in figure 2 showing instances of graph pattern elements.

The generated graphical user interface offers context sensitive menus for all displayed objects and the elements of graph pattern. In addition, it provides a trace mechanism which allows to record user interactions. This allows to “replay” user interactions and offers convenient support for debugging even in a very complex object population. We offer a simple mechanism to add application specific extensions in C++ or Tcl to the rapid prototype.
An example of the default graphical user interface generated from the model in figure 2 is depicted in figure 3.

3. APICES application

APICES raises productivity of application development in the technical domain. Originally, it has been a research prototype to offer automated support for the construction and integration of electronic CAD tools [1][2] and digital signal processing tools [3]. The following sections sketch how APICES can be used and what its advantages are.

• Modelling and documentation

The model editor of APICES supports early phases in the application development process, since it allows to specify the core structure of an application graphically and in an intuitive way. Graph pattern are powerful generic components which are designed for reusability. The support of network-like structures makes modelling of technical applications easy.
The HTML-documentation generator allows to browse the application model, its user-defined comments and the specification of generated methods. The documentation serves as reference in a development team and allows to track model evolution in a convenient manner.

• Rapid prototyping and application development

APICES supports very rapid prototyping by generating an application core and a corresponding prototype from the application model. Since the generated prototype consists of a C++ library embedded in the script language Tcl, all generated methods of the application core are offered at an interpreted interface. Prototyping of algorithms can be done by accessing the application core via Tcl-procedures.
The generated graphical user interface of the prototype offers demonstrators after a very short time. They can be used to discuss design alternatives or model modifications. Changes in the application model are cheap because the effort to align the prototype is minimal. The generative approach followed by APICES allows very short re-design cycles and fosters incremental application design.

APICES is running under Solaris and Linux. Further actual information is available from the web (http://set.gmd.de/APICES).

4. References

[1] Ansgar Bredenfeld, Raul Camposano, “Tool Integration and Construction Using Generated Graph-Based Design Representations”, in *Proceedings of the 32nd ACM/IEEE Design Automation Conference (DAC’95),* pp. 94-99, 1995

[2] Ansgar Bredenfeld, “Automatisierte Integration von Entwurfswerkzeugen für integrierte Schaltungen”, Dissertation, *GMD-Bericht Nr. 273,* München-Wien, R. Oldenbourg Verlag, 1996, ISBN 3-486-24087-0 (in German)

[3] Ansgar Bredenfeld, “APICES - Rapid Application Development with Graph Pattern", in *Proceedings of the 9th IEEE International Workshop on Rapid System Prototyping (RSP'98),* Leuven, Belgium, pp. 25-30, June 3-5, 1998

PERFORMANCE MEASUREMENT METHODOLOGY-AND-TOOL FOR COMPUTER SYSTEMS WITH MIGRATING APPLIED SOFTWARE

S.L. Frenkel

The Institute of Informatics problems, Russian Academy of Sciences, Moscow, Russia
Fax: (095) 310-70-50 , E-mail: slf@dep34.ipian.msk.ru

1.Introduction

Software measurements are essential for any computer system (CS) development projects. Requirements to measurement depend on the CS development technique/scenario. One of widespread components of current CS design philosophy is reuse. In accordance to [Laforme96], reuse activity can be represented in terms "donors" that build assets with reuse and "clients" that integrate those reusable assets. From this point of view, "migration" of programs (e.g., of some libraries or applications) from a machine where they were developed, (considered often as an instrumental machine (IM))) into some "target" environment("target system" (TS)) , is a particular instance of the reuse activity.

As for current practices in computer systems development and engineering, both reuse and migration are closely associated with hardware/software (HW/SW) co-design methodology, where also "migrating software" exists copiously. The main goal of (HW/SW) co-design is to develop a system with an optimum trade-off between performance and design cost, trying to achieve balance of its software and hardware components. This partition can be performed basing on the rigorous synthesis methods [Gupta95], or on the designer's experience. HW/SW codesign methodologies, in fact, today also are used in the practice of universal computing system design. For example, many different systems include successfully used emulators to run applications on platforms for which they were not initially targeted [Sites92]. In this case, designers have to look for trade-off software/hardware solutions, e.g., whether to provide the performance by the expensive translation techniques or through the cache memory enhancement. (Note, that today the term "migration" is also concerned with some special multimedia issues, that, however, is outside the scope of this paper).

In any cases of reuse and HW/SW activity, a system designer should be able to estimate and predict the possible performance level at the early steps of the TS design. This means that he should use a metric to evaluate both software reused (migrated) and compilers of the TS from the point of view of the performance. Such metric could tell him, for example, if one reuse-based technique is more or less effective than another, or whether to put a particular block in hardware or software has to be based on the metrics of interest for the entire system as well as above thirds party software-and- hardware components. As the analytical approaches to performance evaluation (based, as a rule, on the Markov chain model) have high computational complexity, so-called informal "characterization" [Saavedra89] is used widely for the system optimization. It measures the workload of the some accessible (at the given design stage) system's prototype to understand the possible impact of its characteristics on the performance of the whole target system. Usually, such measures are either some temporal characteristics or frequencies of some events (e.g., function calling, operations executed, opcode appearance etc.).

From the programmer's viewpoint, such "characterization" is a behavioral characterization of a migrated software simulated on an instrumental environment. In fact, relatively to the instrumental machine we could interpret this software as some applications set ("domain") which must be implemented at the target machine. In this case, the performance is defined relatively to a given workload, which is determined by the specified applied software system. So, among software characteristics it is desirable to have some metrics concerning the performance-determining properties of software application domain (AD). However, as follows from literature analysis [Gong95], there are no software metrics which could be used to make accurate predictions concerning behavioral characteristics of software from the point of view of speed-up of software reused-or-migrated on the TS. The point is that, no suitable mathematical models to express the relationships between structural characteristics of source codes and some temporal characteristics of program execution, because of the ambiguity of these characteristics impact on the compilers functionality (on the target platform). In traditional approaches to the machine characterization [Saavedra89], there is no the unique definition of "performance" itself. In fact, performance is a vector of temporal parameters, which are set of measurements giving the utilization either of the major system resources (and the amount overlap in CPU and I/O utilization) or other times for the separate events in the system (e.g., duration of primitive operations of some high-level language [Saavedra89]). Thereby, in spite of different performance providing problem definitions[Saavedra89, Gong95, Edward96], the sums of products of program operations frequencies values and expected (or measured) times of their execution on the TS are used as a performance metric. As an immediate effect of the lack of the model, the possible increasing of the CS design cost may be pointed out, because optimization results obtained with some benchmarks measuring may be not representative (from the point of view of time execution) for one or other programs from an applied domain specified or from some program's class. As a first step to overcome this problem, the rigorous model for based on profiling/tracing performance estimation should be developed.

This paper represents one approach to general mathematical model of performance, which integrates some software characteristics, and some temporal characteristic measured at the TS prototype (physical or virtual), as well as a performance evaluation tool corresponding to this model. The approach is mostly intended for CSs designed for various specific AD, e.g. for example, for DSP image processing programs , or for some application software system migrated (like MS-Office). It was shown that this model is generic for the well-known performance characteristics, justifying, in particular, the using of so popular software characteristics as operations frequencies. Also some principal aspects of the frequencies measurement will be considered as well as the measurement tool implementation .

2.Performance Evaluation Model

As mentioned above, the characterization is performed in terms of the sum of the frequencies-and-times products. Therefore, in order to model the performance evaluation we should understand whether these sums reflect some performance measure as a time characteristic for a class of programs with respect to the some application domain.

Let $P= \{(P_1,ID_1), (P_2,ID_2), \dots ,(P_s\ ID_s)\}$ be a sequence of programs P_i , $i=1,\dots s$, with input data suites $\{ID_i\ )$ from some AD, $O_1,.\ O_m$ are the set of "basic operations" in terms of which the program are represented (that is, in terms of some software specification language, e.g., VHDL, C, or some subroutines mapping the vertices/edges of corresponding program data/flow graph [Gong95]), $\tau_1,..\ \tau_m$ are the times of each O_i execution, which are independent random values relatively to P (but they are constants during each program execution). The randomness of-the times (over set $\{ P_i \}$) is due to cache miss, context switching etc.
Let's consider the sum $\sum f_i*\tau_i$, i=1, M, where M is the number of the basic operations in the sequence P (that is, length of the sequence).
Theorem. Let T be expected time of execution of sequence P.
Then, under above conditions for, TS we have for T:

$$T/M \xrightarrow{P} \sum_{i=1}^{M} f_i * \tau_i \qquad (1)$$

where $\rightarrow^P$ is the convergence in probability, (The proof see in [Fr98]).

Suppose the programs sequence P is united program (with the segments P_i) with the corresponding united input data set $\{ID_i\}$. Then, the sum of the weighted (by the frequencies) basic operation execution times is proportionally with the average time of execution of the some large program from specific AD, as, from (1) follows $T/M \approx \sum f_i * \tau_i$. Obviously, if this program had some "characteristic" properties relatively to the programs from AD, T could be considered as a natural performance measure definition, if, at least, P includes all semantic elements of AD (e.g., all image processing operation).

Furthermore, let's consider AD as a general population, and P as a corresponding statistical sample [Pollard77]. Then, to pretend to be a characteristic program, the size of P should provide the suitable (from the statistical point of view) confidence intervals for all measurements (f_i , τ_i, $f_i * \tau_i$). In this context, T reflects the expected time of the program execution over AD, and, therefore, it is relevant characteristic of the performance. By this means, there is the explicit monotone dependence between T and both f_i and τ_i . Correspondingly, this monotonous dependence on each of $f_i*\tau_I$ determines the possibility to use the percentage terms $f_i*\tau_i/\sum f_i*\tau_I$ as a guide to locate bottlenecks, knowing the parts of CS hardware/ software system which are responsible for the corresponding instruction groups. The techniques to use similar data is well-known [Bashr97]. However, to use well of above percentages to variants ordering, we have to provide $\{ f_i \}$ estimation with enough accuracy (in a statistical sense), otherwise it would be wrong results of comparisons of design versions (e.g. because of insufficiently narrow confidence intervals for f_i values).

So, mentioned above model (with its interpretation) demonstrates that traditional characterization techniques based on both the frequencies and the times of basic operations measurements can represent the real performance of CS, if we take into account the requirements to programs sample, which, in fact, are like to requirements for test patterns for testing of system specification [Howden86].

3. Tool for Windows NT applications characterization

Our experience of using of mentioned above performance measurement model regards to RISC system that must execute the x86 Windows NT applications. The use of (f_i ,τ_i) data for the performance evaluation to improve the applications execution time at this RISC platform has been examined. This goal had to be basically achieved by the improving of the x86 instructions emulating ways. One of the specific aspects of this investigation was the studying of the possibility to provide above requirements to software measurement, i.e. to provide suitable confidence intervals for the frequencies measured in so complicated environment as Windows NT.

We had to solve some methodological problems to separate system's and applications calls to provide AD statistics collection, as well as the problems of various conditional probabilities definition. These problems have been solved by using of both appropriate statistics collection techniques and some mathematical decisions. The frequencies of all possible x86 instructions types have been measured by the special program (altogether eleven types have been extracted, e.g. vector of addressing forms, vector of prefixes size etc. All these types can be related to AD semantic classes (section 2)). The measurements have been grouped in the vectors, which represent the characteristic of applications migrated in accordance with these types. Each vector includes the number of bits, corresponding to the number of values of the characteristic. For example, the vector ***Register***: {al,cl,dl,bl,ah,ch,dh,bh,ax,cx,dx,bx,sp,bp,si,di,ea,ec,eb,ed,es,cs,ss,ds,fs,gs}, contains the frequencies of using of corresponding registers under workload considered.

Let's consider briefly the ways of measurement of these frequencies for AD on the background of system's programs. There are two types of system's calls having an impact on the frequencies estimation. First, this is various dialog boxes, which should not migrate to target CS. The instructions of this component may be excluded from the statistics collected by the stopping of corresponding collection ("Suspend" regime). Secondly, there is influence of various "invisible" functions like Winglon. As showed the analysis, the best way to smooth the such "noise" influence is to combine of statistics, gathered for each of (P_i, ID_i) (Section 2) to calculate above frequencies from the unified sample. In fact, this is the way to achieve result closely approximating the Bayesian frequencies estimations procedure.

It is easy to see that a natural model for the above frequencies measurement is the polynomial trials [Pollard77]. Correspondingly, the confidence ellipsoid for parameters of the polynomial model can be used as an accuracy characteristic:

$$(m_1-M*p_1)^2/m_1+(m_2-M*p_2)^2/m_2+ ..+(m_1-M*p_N)^2/m_N \le \chi^2(N-1,t)$$
$$p_1+p_2+...p_N=1$$

where N is the number of mentioned above characteristic vector bits, all p_i are the frequencies of these bits in P, $\chi^2(N-1,t)$ is chi-squared criterion with N-1 degrees of freedom, and t is a given significance level (e.g., 5%) [Pollard77], m_i is the measured number each of bit in P (under given $\{ID_i\}$.

The results of our investigations show the possibility to estimate above frequencies with a good accuracy (a huge statistical materials there is in [Fr96]). As mentioned above, one from significant requirements to the data obtained in the framework of the model is to be separable to make a decision during variants comparison, that reduced to the requirement to provide suitable confidence intervals for the $f_i*\tau_i/\sum f_i*\tau_i$ (but not for the frequencies and times only). As can be seen from figure 1, the distances between values are very significant for the various x86 Windows NT applications (e.g., between MS-Office and Mathcad ones), that points out the good possibility for the variants separation (see next page).

Mentioned above program has been implemented using of the single-step instruction execution mode (i.e. using the single-step trap flag). Current mnemonics-or-opcodes diagrams are displayed on the screen during the statistics collection, and collected values of opcode/mnemonics can be indicated by the mouse click in the corresponding diagram's point. Besides the frequencies, this program calculates the various histograms of used memory size as well as basic blocks length, branches displacement size, and - statistics of using of various resources (memory area, registers etc.).

4. Some results and conclusions

Both the mathematical model of performance evaluation for CS with a software migrated and tool for such measurements have been suggested and investigated in this work This model is a generic for many of recently suggested software characterization models [Saavedra89, Gong95, Edward96] as it is based on the $f_i*\tau_i$ product. To investigate experimentally the possible ways to the software migrated characterization, the x86 Windows NT environment has been selected as a prototype of instrumental machine. Of course, this is a particular case from the HS/WS co-design's viewpoint, but we relied on it because on the one hand, this is the multi-threaded software(this is very important for the today's high-reliable and high-performance systems), and on the other hand, because the possibility to use well-known tools to support the data collection. Then, as there are closed application classes within the well-known operating systems ("closed" e.g., in the sense of unified implementation rules (Microsoft Office etc.)), it is possible to determine the conditions of input data and behavior integration.

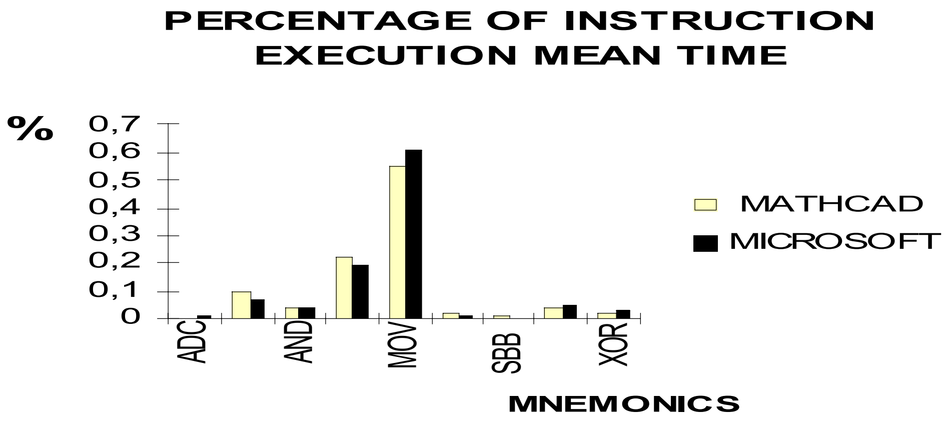


Fig. 1 Diagram of $f_i * \tau_i / \sum f_i * \tau_i$ data for five upper (in terms of this percentage) x86 instructions emulated on the RISC platform.

RISC processor has been used as a hardware prototype. It should be improved (from the performance viewpoint) both by the instruction/data cache size choice and by the control software debugging, matching the above Windows NT application migrated to this RISC platform.

To solve the above applied software/target hardware optimization problem in terms of minimization of average time of migrated software execution (e.g., instrumental machine instructions emulation time), it is suffice to have the tool which allows to combine both IM and TS characteristics groups, that is, frequencies of IM operations and times of their executions at the TS prototype. The main requirement to the combining is to provide the ordering of variants in accordance to this combined value. This ordering may be based on the $f_i * \tau_i$ product. To estimate these values a statistically-justified method to measure of applied programs within the given operational environment has been suggested and investigated. The conceptual model of this environment is the "abstract machine under given workload". The workload is considered as a set of applied functional tasks (AFT) (e.g., "open file", "bitmap transformation" etc.), which characterize (both statistically and semantically) corresponding AD. The choice of the tasks for the test set can be provided in terms of statistical series, corresponding to frequencies values of each migrated operation for each of AFT (maximal, minimal, median etc.).

Acknowledgment

I would like to thank Dmitry L. Petrov and Boris E. Prutkovsky, the Institute of Informatics Problems (IPIRAN) Members, for their efforts to implement software to evaluate above model. Also I am very grateful to Victor N.Zakharov, IPIRAN Deputy Directory for his comments on earlier drafts.

References

[Pollard77] J.H. Pollard, A Handbook of Numerical and Statistical Techniques, Cambridge University Press, 1977.
[Howden86] W.E.Howden, Functional Program Testing and Analysis, McGraw-Hill, 1986.
[Saavedra89] R. H.Saavedra-Barrera et al. Machine characterization Based on an Abstract High-Level Language Machine, *IEEE Trans. On Comp.* Vol 38, No 12, December 989, pp. 1659-1679.
[Sites92] R L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson, “Binary Translation”, *Digital Technical Journal,* Vol. 4, No. 4, 1992
[[Gupta95] Co-Synthesis of Hardware and Software for Digital Embedded Systems, *Kluwer Academic Publisher*, 1995.
[Bash97] D.Bash and M. Zagar, ATLAS, *IEEE Design & Test,* July-September, 1997, pp. 104-112
[Gong95] J. Gong et al, Software Estimation Using of Generic-Processor Model, *Proc. of EDAC’95,* 1995.
[Edwards97] M.D.Edwards et al, Acceleration of software algorithm using hardware/software co-design, *Journal of Systems Architecture* 42 (1996/1997), pp.697-707
[Laforme96] D. Laforme and M. E. Stropky, Mechnism for Effectively Applying Domain Engeneering in Reuse Activities, Army Reuse Center CACI, INC. -USA, 1996.
[Fr96] S.L. Frenkel, D.L.Petrov, x86 Applications Characterization and x86 Interpreter Optimization, Project report , Contract 795-34-è, IPIRAN .
[Fr98] S.L.Frenkel, One Model for Simulation-Based Approach to Computer System Performance Evaluation, *Proceedings of 3rd EUOROSIM CONGRESS*, Vol.3, pp.599-502, Helsinki, April 14-15,1998.