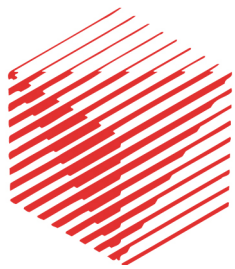
European Research Consortium
for Informatics and Mathematics
ERCIM
www.ercim.org



7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'02)

University of Málaga, Spain
July 12-13, 2002

Rance Cleaveland, Hubert Garavel (Eds.)

Available as Technical Report ITI-2002-5
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Local Organizing Commitee:
María del Mar Gallardo, Pablo López,
Jesús Martínez, Pedro Merino

Foreword

The aim of the FMICS workshops is to provide a forum for researchers who are interested in the development and application of formal methods in industry. In particular, these workshops are intended to bring together scientists who are active in the area of formal methods and interested in exchanging their experiences in the industrial usage of these methods. These workshops also strive to promote research and development for the improvement of formal methods and tools for industrial applications. Topics include, but are not restricted to:

- Tools for the design and development of formal descriptions
- Verification and validation of complex, distributed, real-time systems and embedded systems
- Verification and validation methods that aim at circumventing shortcomings of existing methods in respect to their industrial applicability
- Formal methods based conformance, interoperability and performance testing
- Case studies and project reports on formal methods related projects with industrial participation (e.g. safety critical systems, mobile systems, object-based distributed systems)
- Application of formal methods in standardization and industrial forums

Previous workshops of the ERCIM working group on Formal Methods for Industrial Critical Systems were held in Oxford (March 1996), Cesena (July 1997), Amsterdam (May 1998), Trento (July 1999), Berlin (April 2000), and Paris (July 2001).

This year's workshop is organized at the University of Málaga, immediately after the ICALP 2002 conference. It includes five sessions of regular contributions. We are also pleased to welcome three invited presentations: Andreas Podelski, who discusses abstraction for software model checking, Andrew D. Gordon, who investigates in authenticity types for cryptographic protocols and Wang Yi, who addresses the issue of synthetizing verified real time software.

The proceedings of FMICS 02 are published both physically, as a technical report of the University of Málaga, and electronically, in the ENTCS series (*Electronic Notes in Theoretical Computer Science*).

We wish to thank the members of the programme committee and the additional reviewers for their careful evaluation of the submitted papers (13 papers have been selected out of 22 submitted). We are very grateful to the local organizers at the University of Málaga, and especially Pedro Merino, for their excellent assistance during the workshop preparation.

Finally, we would like to thank ERCIM and ICALP for their financial and organizational support of FMICS 02. Our reviewing process benefited from the METAFrame Online Conference Service (courtesy of METAFrame Technologies, which we would like to thank also for their technical support in setting and running the service).

Rance Cleaveland, Hubert Garavel

June 2002

Keywords: Formal Methods, Formal Description Techniques, Modelling, Specification, Verification, Prototyping, Testing, Software Development, Protocols, Safety Critical Software, Abstractions, Model Checking, Real Time.

Further information about the FMICS working group: **http://www.inrialpes.fr/vasy/fmics**

Programme Committee

- **T. Arts** (Ericsson, S)
- **M. Bernardo** (Univ. of Urbino, I)
- **R. Cleaveland**, **co-chair** (SUNY and Reactive Systems, USA)
- **W.J. Fokkink** (CWI, NL)
- **H. Garavel**, **co-chair** (INRIA Rhone-Alpes, F)
- **S. Gnesi** (CNR/IEI Pisa, I)
- **P. Godefroid** (Bell Labs, USA)
- **H. Hermanns** (Univ. Twente, NL)
- **T. Margaria** (METAFrame Technologies, D)
- **P. Merino Gómez**, **local organization chair** (Univ. Málaga, E)
- **I. Schieferdecker** (GMD Berlin, D)
- **S. Schneider** (Royal Holloway, University of London, UK)
- **M. Sighireanu** (University of Paris-7 Jussieu, F)
- **R. de Simone** (INRIA Sophia Antipolis, F)
- **U. Ultes-Nitsche** (University of Southampton, UK)
- **A. Valmari** (Tampere University of Technology, Fi)
- **W. Visser** (RIACS/NASA Ames, USA)

Additional Reviewers

- Bahareh Badban
- Clara Benac Earle
- Tommaso Bolognesi
- Antonio Cerone
- Kousha Etessami
- Alessandro Fantechi
- Natalia Ioustinova
- Frederic Lang
- Izak van Langevelde
- Michael Leuschel
- Pablo Lopez
- Cecilia Mascolo
- Mieke Massink
- Radu Mateescu
- Simona Orzan
- Jun Pang
- Laurence Pierre
- Simon St James
- Laurent Thery
- Mikko Tiusanen

Local Organizing Committee

Software Engineering Group, University of Málaga

- M. del Mar Gallardo
- P. López
- J. Martínez
- P. Merino, **local organization chair**

Table of Contents

INVITED PRESENTATION

SESSION 4 - INDUSTRIAL CASE STUDIES II

SESSION 5 - MODEL CHECKING

Andreas Podelski (Max-Planck-Institut, Germany)

Abstraction for Software Model Checking

based on joint work with Tom Ball and Sriram Rajamani (Microsoft Research, Redmond WA, USA), and Andrei Rybalchenko (Max-Planck-Institut, Germany)

A program with variables over unbounded data domains (e.g. integers) generates an infinite-state transition system. Thus, one approach to extend model checking to software works by mapping an infinite-state transition system to a finite one. We explain the limitations of that approach and show how one can go beyond. We present a whole spectrum of abstractions with different precision/cost trade-off's. In order to automatize software model checking, we combine the automated process of abstraction with an automated process to make the abstraction more and more precise (namely until the property can be proven in the abstract). We present new ideas on parametrizing such an `abstraction refinement' process.

References

T. Ball, A. Podelski, and S. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In Proceedings of TACAS 2002, LNCS 2280, pages 158-172.

Predicate Abstraction and Refinement for Model Checking VHDL State Machines

Mustapha Bourahla

Computer Science Department, University of Biskra
BP 145 RP, Biskra, Algeria, 07000
Email: mbourahla@hotmail.com

Mohamed Benmohamed

Computer Science Department, University of Constantine
Constantine, Algeria, 25000

Abstract

In this paper we present an automatic combination of abstraction-refinement by which we translate a VHDL model describing a state system to an initial equivalent abstract system described by SMV to explore its state space to verify CTL properties. We present the method implemented to compute automatically abstractions using decision procedures. This method can handle different kinds of infinite state systems including systems composed of concurrent components and it can be extended for more complex VHDL concepts. Abstract models may admit spurious counterexamples (false negative results) which are executions at the abstract level with no corresponding executions at the concrete level. We devise a new algorithm which analyzes such counterexamples and refine the abstract model correspondingly by eliminating gradually the false negative results. We illustrate our approach on an example and we confirm its effectiveness on a large design.

1 Introduction

The main idea of abstract interpretation of digital systems, is to interpret the behavior of a system in a different abstracted (and therefore simplified) system with fewer states for handling the state explosion problem in applying model checking to large industrial designs. An abstraction can be seen as a relation between two systems. On one hand, the original system has the complete description of its behavior, whereas its abstraction preserves some of that behavior and abstracts the rest. The verification task is then performed in the abstracted system. There are two types of abstractions: exact abstractions are those where the result of the verification in the abstract system implies an

equivalent result in the concrete system. In the case of conservative abstractions, on the other hand, only certain results in the verification of the abstract system can be implied in the original system.

Verification by abstraction appears to be promising for reasoning about control intensive designs in which control is finite but the data part is infinite or very large [9] [10]. Abstract models are usually provided manually, and theorem proving is used to check that the provided abstract mapping preserves the properties. Recently, novel techniques based on abstract interpretation have been proposed in the context of the verification of temporal properties where theorem proving is used to compute automatically finite abstractions [2] [7] [8]. These techniques are quite effective, but require heavy use of theorem proving and decision procedures. There are methods/tools that compute an abstract system from the text of a finite state program and an abstraction relation [6]. It should be realized that it is important to avoid the construction of the concrete model which represents the semantics of the considered program before generating the abstract system. Otherwise, one would have to store the concrete system which might be too large. The produced abstract system is usually smaller than the concrete one, and hence is much simpler to model-check.

Verification by abstraction can also be applied to infinite state systems as shown in [12]. However, in all these approaches the verifier has to provide the abstract system and an important amount of user intervention is required to prove that the abstract system simulates the concrete one. What is needed is a method to automatically compute an abstract system for a given infinite state system and an abstraction relation. A method that achieves this for a restricted form of abstraction functions, namely those induced by a set of predicates on the concrete states, is given in [14]. This method has, however, the drawback that it generates an abstract graph rather than the text of an abstract program with the consequence that one can neither apply further abstractions nor techniques for avoiding the state explosion problem as, for example, partial-order techniques. There is another method [3] based on elimination during the construction of abstract systems. Then, to construct an abstract transition of a concrete transition starting from the universal relation, which relates every abstract state to every abstract state, this method eliminates pairs of abstract states such that after elimination of a pair the obtained transition is still an abstraction of the concrete transition. This method is too complex because the number of transitions of the universal relation is exponential in the number of variables. This method was combined with other techniques based on partitioning the set of abstract variables, using substitutions but this partitioning leads to a more non-deterministic abstract system and then more spurious counterexamples.

The drawback of using abstraction followed by model checking as a verification and analysis technology consists in the fact that abstractions are approximations of the original systems that induce false negative results. For

instance, a model checker may exhibit an error trace that corresponds to an execution of the abstract program that violates the desired properties. However, this error trace may not correspond to an execution trace in the concrete program. This situation indicates that the abstraction is too coarse, and that the results of model checking the abstract system are not conclusive. That is too many details were abstracted and the abstraction needs to be refined.

We propose a method for the automatic construction of predicate abstractions extracted from VHDL models to abstract infinite transition systems such that the abstract model by construction simulates the concrete system. These systems can be composed of concurrent components. But the process of constructing the abstract system does not depend on whether the computational model is synchronous or asynchronous, i.e., interleaving based. In general, our technique computes an upper approximation of the original system. Thus, when a specification is true in the abstract model, it will also be true in the concrete design. However, if the specification is false in the abstract model, the counterexample may be the result of some behavior in the approximation which is not present in the original model. When this happens, it is necessary to refine the abstraction. Our method differs from that in [3], so that only the last behavior which caused the spurious counterexample is eliminated. Clarke [6] has presented other technique in another framework of abstraction based on abstraction from a concrete model, but when a spurious counterexample is present, to get a refined abstract model his method eliminates all the non reachable states in the spurious counterexample by using comparison with the behavior of the concrete model. This also costs time and it is not necessary.

The VHDL models are written using a subset of the language [19] and a certain modeling style taken from the most synthesis tools. The VHDL model states are represented symbolically and the abstract state is a conjunction of one of these states and truth assignment to the abstract Boolean variables. The false negative results will be gradually eliminated by an automatic process called refinement which uses information obtained from spurious counterexamples. The verification methodology is based on abstraction followed by model checking and refinement. If there is no possible refinement, the system will report counterexamples by mapping each step in the trace to the concrete domain. We have used an example to explain our method which is implemented to automatically construct the abstract systems.

This paper is organized as follows: Section 2 presents modeling style of transition systems with VHDL. In Section 3, we present the framework of predicate abstraction used by our algorithm of abstraction presented in Section 4. Section 5 presents the algorithm of refinement. An overview of the tool and analysis of results are presented in Section 6. At the end a conclusion is given.

2 Modeling Transition Systems with VHDL

Hardware Description Languages (HDLs), most notably VHDL, have gained considerable popularity in the specification of hardware designs. VHDL supports process level parallelism. It employs constructs with complicated semantics to achieve concurrency, communication and synchronization among the processes [19]. VHDL constructs such as signal assignment statements and wait statements facilitate deterministic inter process communication and coordination. One can exploit these features of VHDL to write succinct behavioral descriptions.

Definition 2.1 (transition system). A transition system M is a tuple $M = (S, V, T, I)$, where

- S is a set of system states
- V is a set of system variables of any type
- T is a set of system transitions, each transition is associated with a guard expression and a set of action expressions over the set V
- I is a set of initial states

VHDL is a language particularly adapted to the description of transition systems because of its high level syntax (instructions if ... then ... else, case ... when) which allows direct translations of traditional graphic representations like graphs and diagrams. With the VHDL syntax, we can name the states, the signals, etc. This gives us a clear and readable descriptions. For abstraction we have chosen a subset of VHDL to describe transition systems. A transition system can be composed of one behavioral component or many concurrent components. Each component is described by one process. The variables of the transition system can be of any type: Boolean, bit, integer, real, etc., and the states are represented symbolically. A directive is introduced to write the CTL formulas to be checked over the VHDL model. We illustrate our verification approach on the well known algorithm that computes the GCD (Great Common Divider) of two natural numbers x and y. The transitions are of the form condition/action, with the meaning that the transition takes place if condition is true, and then action is executed. The VHDL model of GCD is shown in Figure 1 ("<=" is the VHDL assignment operator) .

```
entity GCD is port(clk :  in bit; x, y :  in integer;
    start :  inout bit; z :  out integer);
end entity GCD;
architecture Behavior of GCD is
    Type State is (S0, S1, S2);
    Signal S : State := S0; Signal xp, yp :  natural;
begin
    process begin wait until Clk = '1';
        case S is
```

```
          when S0 =>
            if start = '0' then S <= S0 end if;
            if start = '1' then xp <= x; yp <= y; S <= S1; end if;
          when S1 =>
            if xp < yp then yp <= yp - xp; S <= S1; end if;
            if xp > yp then xp <= xp - yp; S <= S1; end if;
            if xp = yp then Z <= xp; S <= S2; end if;
          when S2 => Start <= '0'; S <= S0
        end case;
    end process;
    -- $ AG(Start = '1' -> AF(xp = yp))
end behavior;
```

Fig. 1. VHDL model of GCD and property specification

3 Framework of Predicate Abstractions

Predicate abstraction consists of using predicates over concrete variables as Boolean abstract variables [14]. It can be defined in the framework of abstract interpretation using Galois connections.

Definition 3.1 (Abstraction by Galois Connection). Let S_c and S_a represent the concrete and abstract state domains respectively. A Galois connection [2] from S_c to S_a is a pair of functions $\alpha : 2^{S_c} \rightarrow 2^{S_a}$ and $\gamma : 2^{S_a} \rightarrow 2^{S_c}$ such that:

- α and γ are total and monotonic.
- $\forall X \in 2^{S_c}, \gamma o \alpha(X) \supseteq X$, and
- $\forall X \in 2^{S_a}, \alpha o \gamma(X) \supseteq X$.

Theorem 3.2 *(Relation between connection and simulation). Let R_c and R_a represent the transition relations of M^c and M^a respectively. If (α, γ) is a connection between S_c and S_a and $\forall S' \in 2^{S_a}$, then $\alpha(Pre(R_c, \gamma(S'))) \subseteq Pre(R_a, S')$ then $M^c \preceq M^a$, where Pre is the pre image function.*

If P is a predicate over concrete variables, a predicate abstraction can be expressed as a Galois connection [14] as follows:

$$\alpha(P) = \bigwedge\{B^a | P \Rightarrow \gamma(B_a)\} = P^a,$$

where B^a is any Boolean expression over the set $\{B_1, ..., B_k\}$ which is the set of abstract variables corresponding to the set of concrete predicates $\{\phi_1, ..., \phi_k\}$. γ is defined as a substitution function, that is, $\gamma(P^a) = P^a[\phi_1/B_1, ..., \phi_k/B_k]$, where each Boolean variable B_i is substituted by its corresponding correct predicate ϕ_i. Thus, the abstraction of a concrete set of states represented by a predicate P over concrete variables is defined as the smallest Boolean formula P^a over the abstract variables B_i, that is, an over approximation of P. For computing the most precise Boolean abstraction with respect to a set

of predicates, for systems where the transition relation is given as a relational predicate, an efficient enumeration of all Boolean combinations B^a to test the assertion $P \Rightarrow \gamma(B^a)$ should be specified. This will abstract systems where the transition relation is given as a predicate. Each implication $P \Rightarrow \gamma(B^a)$ is submitted to the decision procedure to test its validity. Notice that any approximation of P^a is a valid abstraction of P.

Thus, in order to compute for a concrete system M, an abstract system M^a, it is sufficient to abstract the initial state I by computing $\alpha(I)$, and to abstract each transition $t \in T$ as follows:

$$t^a = \alpha(t) = \alpha(action_t(V, V')) = \bigwedge\{(B^a, B^{a'}) | \vdash post[t](\gamma(B^a)) \Rightarrow \gamma(B^{a'})\},$$

that is, the pair $(B^a, B^{a'})$ characterizing the abstraction of the set of possible predecessors by t and the abstraction of the set of possible successors by t, where post expresses the strongest post condition by a transition t of a predicate P over the state variables of V, it is defined as follows:

$$post[t](P) = \exists V'.action_t(V', V) \wedge P(V'),$$

where $action_t(V', V)$ is defined as the relation between the current state and next state, that is the expression:

$$(s = s_i) \wedge guard \wedge \bigwedge_{i=1}^{l} (v_i^{'} \Leftarrow e_i) \wedge (next(s) = s_j)$$

The preservation of properties expressed in temporal logic is established via equivalences and preorders between the concrete and abstract models.

Theorem 3.3 *(weak preservation). Let M be a concrete system, and let M^a be a predicate abstraction of M using any set of predicates. We have $M^a \models \alpha(\phi) \Rightarrow M \models \phi$, for each temporal formula ϕ.*

Proof. All the executions of M are executions of M^a, then if a property holds along all execution paths of M^a, it holds in all execution paths of M. This means that M^a simulates M, because the following holds for each transition t of M:

$\forall P.post[t](P) \Rightarrow \gamma(post[\alpha(t)](\alpha(P)))$. □

This theorem indicates that when a property is established in the abstract system, its corresponding concrete property holds in the concrete system. However, nothing can be concluded when the property does not hold in the abstract system. Strong preservation results can be applied in this case under some conditions.

Theorem 3.4 *(strong preservation). Let M be a concrete system, and let M^a be a predicate abstraction of M using any set of predicates that includes all the literals appearing in the guards of M and in the property ϕ. If M^a is deterministic, we have $M^a \models \alpha(\phi) \Leftrightarrow M \models \phi$, M^a and M are equivalent.*

You can find its proof in [14]. The strong preservation result allows us

to avoid false negative results by mapping abstract error traces to concrete executions violating the property. However, the condition for strong preservation requires that M^a be deterministic. This is usually not the case. Each abstract state is then a conjunction of a subset of the set of Boolean variables which are the codes of the finite abstract domain. The concretization of an abstract state is a set of concrete states that can be represented as a predicate. We have used these notions of predicate abstractions to automatically abstract transition systems described with VHDL. The next section presents the algorithm and illustration on the example of GCD.

4 Automatic Construction of Predicate Abstractions

The algorithm uses decision procedures for the automatic construction of a predicate abstraction of a concrete, infinite state system described as a transition system with VHDL. The abstraction of a concrete system $M = (S, V, T = \{t_1, ..., t_n\}, I)$ is an abstract system $M^a = (S, V^a, T^a = \{t_1^a, ..., t_n^a\}, I^a)$ such that:

- V^a is the set $\{B_1, ..., B_k\}$
- T^a is a set of abstract transitions.
- I^a is the abstract initial state computed as $\alpha(I)$.

The abstraction algorithm consists in computing I^a and for each concrete transition t defined as $(s = s_i) \wedge guard \wedge action \wedge (next(s) = s_j)$ a corresponding abstract transition t^a defined as $(s = s_i) \wedge guard^a \wedge action^a \wedge (next(s) = s_j)$

Algorithm 1 *Abstraction*
Step 1: Define the abstraction function α using the predicates in the transition guards and the CTL formula. The function γ is the corresponding substitution function.
Step 2: For each guard, the abstract guard ($guard^a$) is computed as $\alpha(guard)$. When using the literals of the guards as abstract Boolean variables, $\alpha(guard)$ is an exact abstraction, where each literal of guard is substituted syntactically by its corresponding abstract Boolean variable.
Step 3: Construction of a list L of all the Boolean expressions B^a of the form $\bigwedge(B_i/\neg B_i)$ using the abstract variables.
Step 4: The action assignments of each transition will be abstracted to a Boolean expression composed of maximum number of abstract variables and it should validate the implication:

$$post[t](true) \Rightarrow \gamma(\textit{the abstract Boolean expression}).$$

The "abstract Boolean expression" is a conjunction of all the Boolean expressions B^a taken from the list L, where the implication $post[t](true) \Rightarrow \gamma(B^a)$ is valid. This means that for each abstract variable B_i in this expression, the strongest post condition by t of any arbitrary state is in $\gamma(B_i)$ or in $\neg\gamma(B_i)$, that is, in ϕ_i or in $\neg\phi_i$. If the abstract variable is not in the expression this means that is not deterministic.
Step 5: The variable S is not abstracted since it is of finite type.

4.1 Illustration on the Example

We use predicates over concrete variables which are extracted from the VHDL model, as Boolean abstract variables. The transition table generated after the parse of the VHDL model (Figure 1), is shown on Table 1.

N	Present State	Guard	Action	Next State	$Guard^a$	$Action^a$
1	S0	Start = 0	Empty	S0	B1	Empty
2	S0	Start = 1	xp := x; yp := y	S1	$\neg B1$	Empty
3	S1	xp < yp	yp := yp - xp	S1	B2	Empty
4	S1	xp > yp	xp := xp - yp	S1	B3	Empty
5	S1	xp = yp	z := xp	S2	$\neg B2 \wedge \neg B3$	Empty
6	S2	True	Start := 0	S0	true	B1 := true

Table 1
Transition table of the GCD

The columns $Guard^a$ and $Action^a$ are filled in after the abstraction. First, we compute the abstract initial state. The VHDL model contains one initialization statement ($S := S0$), this will not be abstracted. Then, the abstract initial states are any state verifying the formula ($S = S0$). Second, we compute the abstract guards of all the transitions along with the specification predicates. The set of predicates presented in the column Guard with the set of predicates generated from the CTL formulas presented by the directive `--$` (in this case, the set is $\{Start =' 1', xp = yp\}$), will be the entry to the abstraction algorithm for producing the set of abstract Boolean variables and the equivalent abstract predicate of each concrete predicate using a decision procedure. The algorithm will take the predicates in the Guard column one by one and it will try to express them with the already constructed abstract variables. If it is not possible, it decomposes the predicate to simpler predicates (by simpler, we mean removing the Boolean connectors) and then, it associates a new abstract variable to one of them which is not already associated and then it will retry the process until the predicate is completely expressed with the constructed abstract variables. We will repeat the process until all the predicates can be expressed with abstract variables.

If the set of abstract variables for our example is $\{B1$ for $(start = 0)$, $B2$ for $(xp < yp)$, and $B3$ for $(xp > yp)\}$, this means that the abstraction function α is defined by the predicate $(B1 \leftrightarrow (start = 0)) \wedge (B2 \leftrightarrow (xp < yp)) \wedge (B3 \leftrightarrow (xp > yp))$, then we need to represent all the guard predicates with the minimum of abstract variables using calls to a decision procedure. The abstract guard of transition number 3 (see Table 1), for instance, is $\neg B2 \wedge \neg B3$ because the implication $((xp = yp) \Rightarrow \gamma(\neg B2 \wedge \neg B3))$ is checked to be valid.

Third, we compute the abstraction for each assignment in the action of each transition. The assignments are in the Action column. We need to realize a conjunction of the maximum number of abstract variables (or their

negations) to abstract these assignments such that, the following implication "assignment of transition *action* $\Rightarrow$ γ(conjunction of the maximum number of Boolean abstract variables)" should be valid. These implications will be checked by calls to a decision procedure. The abstraction of an action is the conjunction of all the abstractions of its assignments. After the construction of all the abstract predicates, a translation program will generate the equivalent SMV module (Figure 2). The SMV system [20] is a tool for checking finite state systems against specifications in the temporal logic CTL.

```
Module main
VAR
   B1 : boolean; B2 : boolean; B3 : boolean;
   S : {S0, S1, S2};
INIT
   S = S0
TRANS
   (S = S0 & B1 & next(S) = S0) | (S = S0 & !B1 & next(S) = S1) |
   (S = S1 & B2 & next(S) = S1) | (S = S1 & B3 & next(S) = S1) |
   (S = S1 & !B2 & !B3 & next(S) = S2) |
   (S = S2 & next(B1) & next(S) = S0)
INVAR
   (B2 & !B3) | (!B2 & B3) | (!B2 & !B3)
SPEC
   AG(!B1 -> AF(!B2 & !B3))
```

Fig. 2. SMV Abstract model

4.2 Invariant Generation

In the SMV module (Figure 2), there is an invariant. The invariant is a formula representing a set of states and each state reachable in the system, is in this set. The invariant formula is to make consistence between the abstract Boolean variables already generated so that to not get a state in which there will not be a formula making no sense, and then avoiding the system to reach useless states. By example there will not be any concrete state verifying the formula $B2 \wedge B3$ (its equivalent in the concrete domain is "$(xp > yp) \wedge (yp > xp)$"). The idea of the following algorithm is to check if not $(B2 \wedge B3)$ is a tautology. If yes, this combination will be removed from the invariant formula.

Algorithm 2 *Invariant Generation*

Consider B is the set of abstract Boolean variables
Consider P is the set of the equivalent predicates
Consider P^S is the set of subsets from P, where predicates of each element from P^S, are using the same subset of concrete variables
Consider B^S is a set of subsets from B, where each $B_j \in B$ is an abstract of one of P_i^S from P^S

Invariant $\leftarrow$ *true*
for each B_j^S *from* B^S *and if* B_j^S *contains more than one element do*
  for (every conjunction C *of all* B_j *or not* B_j *in* B_j^S*) do*
    if not valid(not $\gamma(C)$*) then*
      Invariant $\leftarrow$ *Invariant* $\wedge C$
    end if
  end for
end for

The algorithm first, searches the abstract Boolean variables that are abstractions of concrete predicates using the same concrete variables and grouping them in clusters of variables. Then the algorithm will try to check all the conjunctions composed of these clusters of variables. If the negation of the equivalent conjunction in the concrete domain, is checked to be a tautology, it will not be inserted in the invariant formula. We should remark that the invariant can be true (empty).

4.3 Model Checking the Abstract Model

Once an abstract system is constructed, the SMV model checking system is used to explore its state-space. The advantage of model checking over other verification techniques is its ability to generate counterexamples when a property is violated. The error trace is a sequence of states and transitions starting from the initial state of the system leading to a state violating the property. Error traces of an abstract system can be mapped to executions of a concrete system since each abstract transition corresponds to a single concrete one. Figure 3 shows an error trace which is a spurious loop counter example violating the property specified.

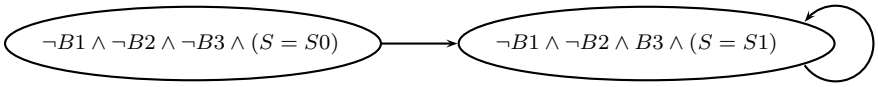


Fig. 3. Error trace

The simulation of the error trace on the concrete system indicates that it does not correspond to an execution of the concrete system. However, this does not rule out the possibility that the property is violated. In the next section, we present an algorithm to show how model checking can guide the automatic refinement of an abstract system until the property is verified or a counterexample corresponding to a concrete execution violating the property is generated.

5 Automatic Refinement of Abstractions

The specification above is not satisfied by the initial abstract system already constructed. The system SMV produced an error trace indicating the violation

of this specification (Figure 3). By analysis of this error trace, we understand that the abstract system is executing a trace which can not be executed in the concrete system. The abstract system is too abstract and it needs to be refined. Effectively, the transition number 4 (see Table 1) is defined by the formula $(S = S1 \wedge B3 \wedge next(S) = S1)$. The Boolean variable $B3$ can get the next value true or false which is not deterministic, but in the concrete system the next value of $(xp > yp)$ will eventually get the value false. Figure 4 shows the different counterexamples (A, B and C) generated after each step in the refinement process of this module until the satisfaction of the specified liveness property.

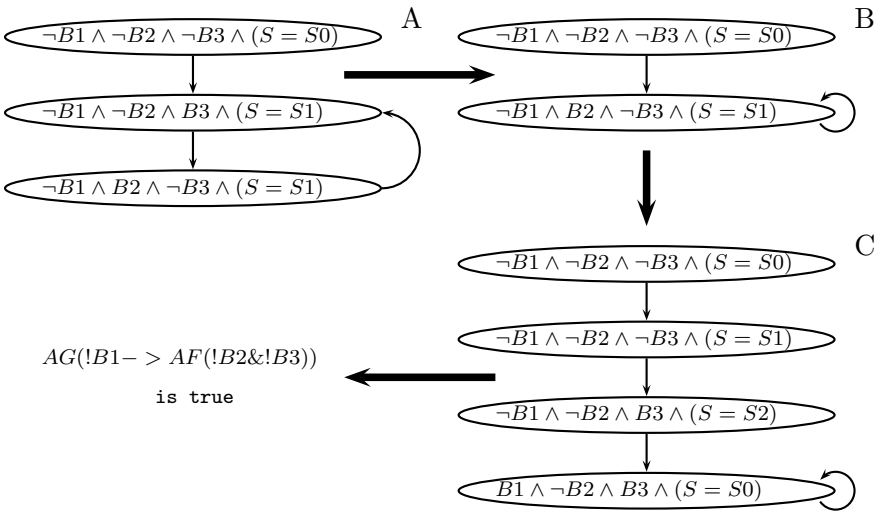


Fig. 4. Error traces generated by model checking different levels of refinement

Thus, we have model checked four abstract models produced gradually by refinement from the initial abstract model (Figure 2), until the property is verified. In the following and before presenting our algorithm of refinement, we will explain our method of refinement on this example. We take the formula of the last transition t^e in the error trace (see Figure 3)

$$t^e \equiv (S = S1) \wedge \neg B1 \wedge \neg B2 \wedge B3 \wedge \neg next(B1) \wedge \neg next(B2) \wedge next(B3) \wedge (next(S) = S1)$$

By mapping to the abstract transition system already we have (the initial abstract model), the equivalent abstract transition t^a, is (see Figure 2)

$$t^a \equiv (S = S1) \wedge B3 \wedge (next(S) = S1)$$

This transition formula should verify the equality $t^e \wedge t^a = t^e$ to be considered. In our approach we take only the abstract variables that are used in the abstraction of predicates composed of concrete variables used by the action of the transition t^c, which has t^a as its abstract transition (in this case they are $B2$ and $B3$). Then, we take one abstract variable (for example, $B3$). The predicate $next(B3)$ is in t^e (because, $t^e \wedge next(B3) = t^e$) and it is not in t^a, because $t^a \wedge next(B3) \neq t^a$. Then we should check the validity of

$action_{t^c} \Rightarrow \gamma(B3)$. In other words the following implication should be valid $(xp = xp - yp) \Rightarrow (xp > yp)$. But, the decision procedure does not valid this, and because it causes an error in the model, we may use its negation to avoid it and the transition will be written like the following for the new refined abstract model.

$$(S = S1 \wedge B3 \wedge \neg next(B3) \wedge next(S) = S1)$$

When we model check this modified abstract module, we get another error trace (Figure 4 A) and the formula of the last transition t^e in the new error trace is

$$t^e \equiv (S = S1 \wedge \neg B1 \wedge \neg B2 \wedge B3 \wedge \neg next(B1) \wedge next(B2) \wedge \neg next(B3) \wedge next(S) = S1)$$

The equivalent transition in the current abstract model t^a, is

$$t^a \equiv (S = S1 \wedge B3 \wedge \neg next(B3) \wedge next(S) = S1)$$

This is the abstract transition already modified. Now we apply the same rule as above and the abstract variable $B2$ will be taken. The predicate $next(B2)$ is not occurring in the current transition so we have to validate $(xp = xp - yp) \Rightarrow (yp > xp)$ which is also invalid and it can be eliminated from the abstract system. The transition of the second refined abstract model should be written now like this

$$(S = S1 \wedge B3 \wedge \neg next(B2) \wedge \neg next(B3) \wedge next(S) = S1).$$

There still an error (Figure 4 B), and its trace gives the formula of the last transition

$$t^e \equiv (S = S1 \wedge B2 \wedge \neg B3 \wedge next(B2) \wedge \neg next(B3) \wedge next(S) = S1)$$

This formula modifies by the same way, the transition

$$t^a \equiv (S = S1 \wedge B2 \wedge next(S) = S1) \text{ to be}$$
$$(S = S1 \wedge B2 \wedge \neg next(B2) \wedge next(S) = S1)$$

Now the error trace (Figure 4 C) gives the formula

$$t^e \equiv (S = S0 \wedge B1 \wedge \neg B2 \wedge B3 \wedge next(B1) \wedge \neg next(B2) \wedge next(B3) \wedge S = S0)$$

The equivalent transition in the current abstract system is

$$t^a \equiv (S = S0 \wedge B1 \wedge S = S0).$$

There is no action for this transition, so we can take one of the abstract variables $B1$, $B2$, or $B3$. For example, we take $B1$ and the new transition of the refined abstract model is (the implication $true \Rightarrow \gamma(B1)$ is not valid)

$$(S = S0 \wedge B1 \wedge \neg next(B1) \wedge S = S0).$$

Therefore, this last refined abstract model verifies the specification.

5.1 Refinement Algorithm

After this execution of refinement process on the example, we present our detailed algorithm. This algorithm will be called every time we get an error trace path after model checking an abstract model. The algorithm does not introduce new predicates. It refines the transitions without adding states, so this method of refinement is for liveness and reachability properties.

Algorithm 3 *Refinement*

Let P to be the path of the abstract error trace
While P is not empty do
  Let t^e to be the formula of the last transition in the abstract error trace
  and t^a is the corresponding transition in the abstract model,
  which is verifying the equality $t^a \wedge t^e = t^a$
  Let A to be the set of abstract variables of concrete predicates using
  concrete variables occurring in the action of the equivalent concrete
  transition t^c of t^a
  while A is not empty do
    take v from A
    if $(next(v) \wedge t^e = t^e)$ and $(next(v) \wedge t^a \neq t^a)$ and not
      Valid($action_{t^c} \Rightarrow \gamma(v)$) then
      Refine the abstract model by changing t^a to be $t^a \wedge \neg next(v)$
      and return
    elseif $(\neg next(v) \wedge t^e = t^e)$ and $(\neg next(v) \wedge t^a \neq t^a)$ and not
        Valid($action_{t^c} \Rightarrow \neg\gamma(v)$) then
        Refine the abstract model by changing t^a to be $t^a \wedge next(v)$
        and return
    end if
  end while
  P := P - last abstract state
end while
There is no possible refinement then, output the concrete counterexample after mapping the abstract error trace.

Thus, the main idea of this algorithm is to search the non deterministic abstract variables in the last abstract transition in the error trace. Then it takes these variables one by one to negate their next value which gives a refined abstract model (it is an under approximation). If such variables don't exist it will go backward until the first transition. At the end if there are no deterministic variables, the concrete counterexample will be produced using the function γ.

6 Overview and Experiments

Figure 5 shows an overview of the tool implementing our methodology based on abstraction - model checking - refinement which is dedicated to the verification of infinite state systems.

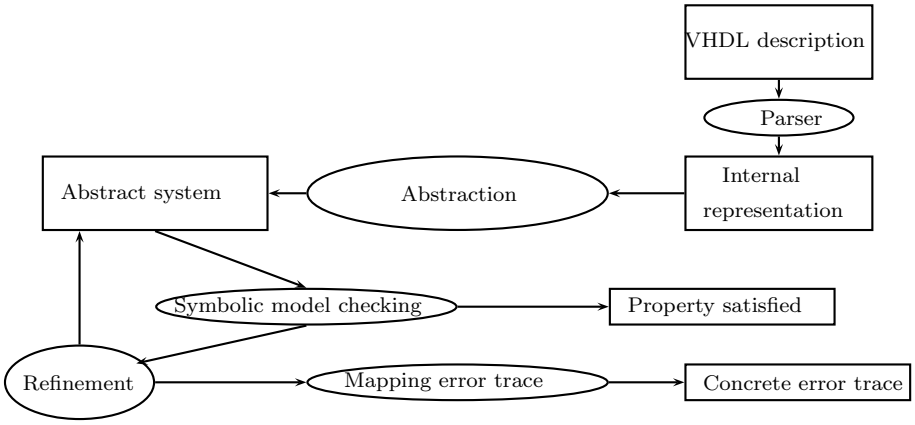


Fig. 5. Overview of the tool

We have implemented this tool under the operating system Windows and we have used the decision procedure of the system SVC (Stanford Validity Checker) [1] [16] to prove theorems. After the execution of the abstraction process we call the system SMV to model check the produced abstract model. If there is a counterexample we call the refinement process as explained above to produce a refined new abstract model in the case if the counterexample is spurious.

In addition to the GCD example, we have used this system to verify the mutual exclusion property (this is a safety property but no refinement was needed) in the Bakery protocol, and by which we have tested the inter-process communication with VHDL and its equivalent in the system SMV. The Bakery protocol is composed of many parallel components each one is represented by a VHDL process. We have also verified the ATM (Asynchronous Transfer Mode) switch [5]. This is relatively a large design and it uses many components. The table below shows our experiments with the three designs. The table shows the number of abstract variables used. It shows the number of implications generated and proved for each abstraction. Also the number of refinements, and the global time of verification.

Case	# of abstract variables	# of calls to decision procedure	# of refinements	Time (s)
GCD	3	18	4	2
Bakery	3	33	0	1.5
ATM	17	254	8	35

Table 2
Experiment results

7 Conclusion

We have presented a novel abstraction refinement methodology for symbolic model checking VHDL models describing state machines, which can be infinite transition systems. The methodology which is implemented by a completely automatic tool, consists of an algorithm for the automatic construction of predicate abstraction by which we translate a VHDL model to an equivalent abstract SMV model, and an efficient algorithm for automatically refining a coarse abstraction when model checking the abstract system fails. This refinement algorithm eliminates gradually the spurious paths in the error trace. The construction of the initial abstract system and the refinement process use many calls to the decision procedure. The choice of non deterministic abstract variables in the refinement algorithm has big effect on its performance and good heuristics for their selection will approve it.

References

[1] C. Barret, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, Formal Methods in Computer-Aided Design (FMCAD '96), volume 1196 of Lecture Notes in Computer Science, pages 187-201, Palo Alto, CA, November 1996. Springer-Verlag.

[2] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In CAV'92, pages 251-263, 1992.

[3] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [HV98], pages 319-331, 1998.

[4] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In R. Alur and T.A. Henzinger, editors, 8th International Conference on Computer Aided Verification, volume 1102 of Lecture Notes in Computer Science, pages 323-335. Springer-Verlag, 1996.

[5] T. Chaney, J.A. Fingerhut, M. Flucke, and J. Turner. Design of a gigabit ATM switching system. Technical Report WUCS-96-07. Computer Science Department, Washington university, St. Louis, Missouri, 1996.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Emerson and A. P. Sistla, editors, Computer-Aided Verification, Lecture Notes in Computer Science. Springer-Verlag, 2000.

[7] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. ACM Transactions on Programming Languages and Systems, 16(5)1512-1542, 1994.

[8] M.A. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In Hu and Vardi, pages 293-304, 1998.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4th ACM symp. of Prog. Lang. Pages 238-252. ACM Press, 1977.

[10] D. Dams. Abstract interpretation and partition refinement for model checking. PhD thesis, Technical University of Eindhoven, 1996.

[11] D. Dams, R. Gerth, G. Dohmen, R. Herrmann, P. Kelb, and H. Pargmann. Model checking using adaptive state and data abstraction. In D.L. Dill, editor, CAV'94, pages 455-467. Springer-Verlag, Berlin, 1994. LNCS 818.

[12] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In Halbwachs and Peled [HP99], pages 160-171, 1999.

[13] J. Dingel and Th. Filkorn. Model checking for infinite state systems using data abstraction. In P. Wolper, editor, Computer Aided Verification, volume 939 of LNCS, pages 54-69. Springer-Verlag, 1995.

[14] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In Computer Aided Verification, volume 1254 of Lecture Notes in Computer Science, 1997.

[15] R.P. Kurshan. Computer-Aided Verification of coordinating processes. Princeton university press, Princeton, NJ, 1994.

[16] J.R. Levitt. Formal verification techniques for digital systems. PhD thesis, Stanford university, December 1998.

[17] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design. 6(1), 1995.

[18] D. E. Long. Model Checking, Abstraction and Compositional Reasoning. PhD thesis, Carnegie Mellon, 1993.

[19] R.E. Harr, and A.G. Stanculescu. Applications of VHDL to circuit design. Kluwer Academic Publishers, 1991.

[20] K.L. McMillan. Symbolic model checking. Kluwer Academic Publishers, Boston, 1993.

[21] H. Saidi and N. Shankar. Abstract and model check while you prove. In Halbwaks and Peled [HP99], pages 443-454, 1999.

A Tool for Abstraction in Model Checking

María del Mar Gallardo, Jesús Martínez
Pedro Merino, Ernesto Pimentel

Dpto. de Lenguajes y Ciencias de la Computación
University of Málaga, 29071 Málaga, Spain

Abstract

Abstraction methods have become one of the most interesting topics in the automatic verification of software systems because they can reduce the state space to be explored and allow model checking of more complex systems. Nevertheless, there is a lack of tools actually supporting this technique. One direction for abstracting a system is to transform its formal description (its model) into a simpler version specified in the same language, thus skipping the construction of a specific (model checking) tool for the abstract model. The abstraction of the model should be followed by the abstraction of the temporal formulas to be checked. This paper presents αSpin, a tool for the integration of several abstraction approaches (for models and formulas) into the well known model checker Spin. In particular, αSpin integrates two dual approaches, the classic abstraction method, based on under-approximating properties, and an alternative approach, proposed by the authors, where abstraction provides an over-approximation of the formulas. [2]

Key words: Model Checking, Abstraction, Tools, Spin

1 Introduction

Computer based verification methods, such as model checking [1], have become realistic techniques to be used in the development of critical systems. However, model checking is only fruitful when a *useful* model of a system is available. By *useful* we mean an abstract representation of the system, containing only the details which ensure that satisfaction (non-satisfaction) of certain properties provides us with information about the actual behavior of the system. Models describing an excess of details may produce the state explosion problem, which could prevent the use of current tools to fully analyze them. This problem affects both the symbolic method, and explicit model checking; both of them employ ideas of abstract interpretation [5] to construct abstract state spaces or models [2,6,4,13]. Whereas most proposals to implement abstraction focus

[2] Work supported by projects TIC99-1083-C02-01 and TIC2001-2705-C03-02
[1] Email: gallardo,jmcruz,pedro,ernesto@lcc.uma.es

on the symbolic approach, there is a great demand for tools for the second approach. This paper presents a method to extend explicit model checkers with abstraction. Although our technique can be applied to different tools, we describe αSpin, an implementation on top of Spin [16,17].

Extending a model checker with automatic abstraction should improve some of the classical steps enumerated by Clarke et al. in [2]: a) defining one abstraction function suitable for the temporal property to be verified, b) constructing the abstract model, and c) relating the verification results to the behavior of the original (concrete) model.

As regards step (a), we propose the use of an abstraction library with previously defined abstraction functions that can be used depending on the properties to be analyzed [10]. This idea is also employed in [9] and [13].

Our method to construct the abstract model (b) is based on syntactic transformation of the model and the formulas. This allows us to reuse the same tool (Spin) to verify the abstract model. The approach also gives us a framework to reason about correctness of the transformation, as discussed in [10]. Finally, we based the transformation on the use of XML [19] in order to be as independent as possible of the actual modelling language [12].

Regarding the relation of results (step c), the classic method to abstract temporal logic is based on defining an abstract satisfability relation which under-approximates the standard one [2,6]. As a consequence, it is suitable to check whether a temporal formula is true for all execution paths (satisfaction of universal properties). We introduce a new approach based on over-approximation of temporal formulas [11], which can be employed to ensure that it is impossible for any path in the abstract model to satisfy a formula (refutation of existential formulas). Our experience suggests that, given a model, both dual methods can be efficiently employed in the verification work. We have thus integrated both techniques in our current implementation, αSpin. Furthermore, the implementation also allows us to explore how to obtain more benefits from the combination of both approaches in the same formula.

Abstraction by syntactic transformation is also supported by other tools, but they are mainly oriented to programming languages, and not to formal description techniques. Furthermore, these tools produce the abstract model in a different language, thus lacking the advantage of reusing the model checker. Tools like FeaVer [18], Bandera [9] or the first version of JPF [15] are considered abstraction tools for Spin because they produce (extract) PROMELA code from the source code (C, Java). We believe that αSpin is complementary to these tools because they deal with different problems. In model extraction, the major aim seems to be how to "remove" great amounts of code to obtain the PROMELA model. In our case, we start with a relatively simple model, and our work focuses on incrementally applying abstraction to the initial and the new PROMELA models. See [7] for an overview of abstraction techniques and associated tools.

Process ::= [`active`[“[”`NumberOfInstances`“]”]]	`proctype` *ProcessTypeID* {*Decl*; *InstSeq*}
InstSeq ::= [*l* :] *Inst*{; [*l* :] *Inst*}*	*Inst* ::= *Basic*|*Jump*|*If*|*Do*|*Atomic*|*D_Step*
Basic ::= *BExp* | *Assign* | *Input* | *Output* | *Rendez*	*Jump* ::= `goto l` | `break`
If ::= `if` *BranchSeq* `fi`	*Do* ::= `do` *BranchSeq* `od`
Atomic ::= `atomic` “{” *InstSeq* “}”	*D_Step* ::= `d_step` “{” *InstSeq* “}”
Input ::= *ChannelId* ? *ExpSeq*	*Output* ::= *ChannelId* ! *ExpSeq*
Rendez ::= *Input* | *Output*	*Branch* ::= :: *Inst*
BranchSeq ::= *Branch*{*Branch*}*	

Fig. 1. Part of Promela Syntax

The paper is organized as follows. Section 2 contains some preliminary background on Spin and its input languages. Sections 3 and 4 present the theoretical basis to support correct abstraction by transformation of promela and temporal logic, respectively. In Section 5, we explain how to use αSpin with a previously published lift controller system as the case study [8]. Section 6 is devoted to the implementation details of αSpin. In the last section we discuss the main contributions of the work, and outline some future works.

2 PROMELA, LTL and SPIN

In the last few years, Spin has become one of the most employed model checkers in both academic and industrial areas [16,17]. It supports the verification of usual safety properties (like deadlock absence) in systems written in the modelling language promela, as well as the analysis of complex requirements expressed with Linear Temporal Logic (ltl). It is also used as the platform to try new powerful algorithms to attack the state explosion problem.

2.1 Modelling with promela

promela is a language designed for describing systems composed of concurrent asynchronous communicating processes. A promela model $P = Proc_1 || \ldots || Proc_n$ consists of a finite set of processes, global and local channels, and global and local variables. Processes communicate via message passing through channels. Communication may be asynchronous using channels as bounded buffers, and synchronous using channels with size zero. Global channels and variables determine the environment in which processes run, while local channels and variables establish the internal local state of processes.

promela is a non-deterministic language that borrows some concepts and syntax elements from Dijkstra's guarded command language, Hoare's CSP language and C programming language (see Fig. 1). A promela process is defined as a sequence of possibly labelled sentences preceded by the declarative part (see example in Fig. 2). *Basic* sentences in promela are those that produce a definite effect over the model state; in other words, the assignments, the instructions for sending (receiving) messages to (from) channels and the Boolean expressions, *BExp*, that include tests over variables and contents

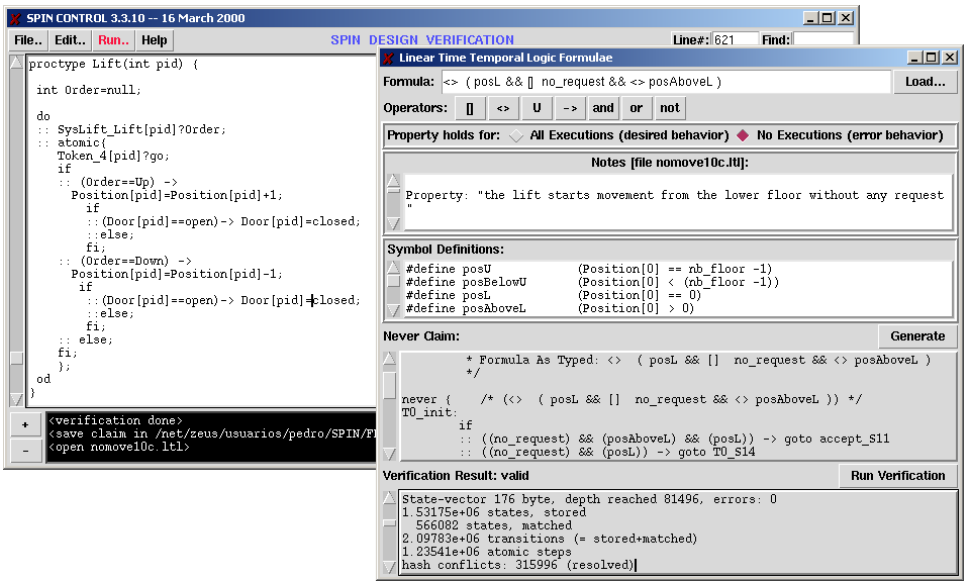


Fig. 2. Lift system model and LTL formula in XSpin

of channels. In addition, PROMELA has other non-basic sentences like the non-deterministic *If* and *Do* sentences.

2.2 Temporal logic

Spin verifies LTL formulas against PROMELA models. Well-formed formulas of linear temporal logic (LTL) are inductively constructed from a set of atomic propositions (in PROMELA, propositions are tests over data, channels or labels), the standard Boolean operators, and the *temporal operators*: *always* "□", *eventually* "◇" , *next* "O", and *until* "U". Formulas are interpreted with respect to model state sequences $t_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$. Each sequence expresses a possible model execution from state s_i. The use of *temporal operators* permits construction of formulas that depend on the current and future states of a configuration sequence. The semantics of LTL is shown in Fig. 3 where p is a proposition, and f and g are temporal formulas. For the sake of convenience, we assume that all formulas are in negation normal form, that is, negations only appear in propositions. Note that we have not included a rule defining the satisfaction of a negated formula. Instead, we treat the evaluation of negated propositions independently of their corresponding non-negated ones. The reason for this will be explained in Section 4. The two last rules in Fig. 3 define the semantics of the universal and existential temporal formulas. There, M represents the set of execution traces produced by the model.

2.3 Spin

By default, given a LTL formula, Spin translates it into an automata that represents an undesirable behavior (which is claimed to be impossible). Then, verification consists of an exhaustive exploration of the state space searching

$t_i \models p$	*iff* $s_i \models p$
$t_i \models \Box f$	*iff* $\forall j \geq i.t_j \models f$
$t_i \models \Diamond f$	*iff* $\exists j \geq i.t_j \models f$
$t_i \models Of$	*iff* $t_{i+1} \models f$
$t_i \models fUg$	*iff* $\exists k \geq i.\forall j.i \leq j < k.t_j \models f, t_k \models g$
$M \models \forall f$	*iff* $\forall t.t \models f$
$M \models \exists f$	*iff* $\exists t.t \models f$

Fig. 3. LTL Semantics

for executions that satisfy the automata. If such an execution exists, then the tool reports it as a counterexample for the property. If the model is explored and a counterexample is not found, then the model satisfies the LTL property as a *universal property*. The same verification scheme can be employed to check whether a formula cannot be satisfied by any path (*refutation of existential properties*). These two ways of using LTL are presented in a user friendly interface called XSpin, as shown in Fig. 2. The first case corresponds to marking "All Executions" and the second one to "No Executions".

Although there are many real examples where the verification can be done with standard exhaustive verification, Spin also implements optimization techniques to deal with complex systems. *Partial order reduction* replaces several interleaved sequences of events (sentences) by only one that represents the whole set. *State compression* reduces the use of memory by compressing the representation of the states without losing information. *Bit-state hashing* represents states as bits in a hash table, so in many cases the analysis is only partial. Our new tool preserves these optimization techniques.

3 Abstracting PROMELA

The first step for realizing abstract model checking is to reduce the model to be analyzed. In [10], we described a method based on the source-to-source transformation to abstract PROMELA models. The main idea in this work is that for abstracting models it suffices to replace the original type definitions (data and basic operations) by simpler ones, in such a way that the control part of the program (high level operations like non-determinism selection and loops, co-routines and so on) remain unchanged. From a practical point of view, this observation is very important, because it allows us to isolate the program points that must be changed when abstracting a model independently of the complexity of the language constructions. In addition, this modular vision facilitates the definition of abstractions, the analysis of the correctness of the abstraction and even the implementation.

In the rest of the section we summarize the theoretical background supporting the source-to-source transformation method. Let *State* denote the set of system states. We define functions $effect : Basic \times State \rightarrow State$ and $test : BExp \times State \rightarrow \{false, true\}$ which describe the effect of executing a

basic sentence and a test in a given state, respectively. The semantic function $G(-, \mathit{effect}, \mathit{test}) : \text{PROMELA} \rightarrow \wp(State^{\omega})$ associates each model M with the set of traces $G(M, \mathit{effect}, \mathit{test})$ representing all possible executions of M, in which functions *effect* and *test* are used when executing a basic sentence or a Boolean expression. Note that functions *test* and *effect* represent the standard implementation of the model data types.

In order to simplify the analysis of properties over $G(M, \mathit{effect}, \mathit{test})$, we must choose an adequate set of reduced states $(State^{\alpha}, \leq^{\alpha})$ and an abstraction function $\alpha : State \rightarrow State^{\alpha}$ which transforms concrete states into their abstractions. Each abstract data is intended to represent a set of concrete states sharing some characteristic which is abstracted. $(State^{\alpha}, \leq^{\alpha})$ is usually a complete lattice, and the partial order $\leq^{\alpha}$ represents the degree of precision of each abstract state. Finally, to obtain the abstract behaviour of the model we must also define abstract versions of *effect* and *test*, that is, functions $\mathit{effect}^{\alpha} : Basic \times State^{\alpha} \rightarrow State^{\alpha}$ and $\mathit{test}^{\alpha} : BExp \times State^{\alpha} \rightarrow \{false, true\}$, giving the proper meaning to the basic PROMELA sentences when executed over abstract states. Given the previous discussion, $G(M, \mathit{effect}^{\alpha}, \mathit{test}^{\alpha}) \in \wp((State^{\alpha})^{\omega})$ defines an abstract behavior, easier to be analyzed, for the same model M.

For instance, Fig. 2 shows an excerpt of a PROMELA model that represents the behavior of a lift (extracted from [8]). In order to simplify the exposition, we assume that system states are given by the value of the variable `Position[pid]` that is an integer number between the values $0..\mathit{nb\_floor} - 1$. Variable `Position[pid]` always stores the current floor for the lift identified by `pid`. To reduce the model size, consider the poset $(\texttt{FLOORS}, \leq^{\alpha})$ illustrated in Fig. 4 and the abstraction function $\alpha : [0..\mathit{nb\_floor} - 1] \rightarrow \texttt{FLOORS}$ defined as $\alpha(0) = \mathit{Lower}, \alpha(\mathit{nb\_floor} - 1) = \mathit{Upper}$ and $\forall 0 < j < \mathit{nb\_floor} - 1, \alpha(j) = \mathit{Middle}$. The use of the partial order $\leq^{\alpha}$ allows us to include the notion of approximation in the abstract domain `FLOORS`: the abstract value *noUpper* approximates any floor different from the *Upper* one, thus *noUpper* is an abstract value less precise than both *Lower* and *Middle*. Value *Unknown* is the least precise abstract data since it represents any floor. Finally, value $\perp$ is used to represent illegal values.

The redefinition of states involves the redefinition of the effect of basic sentences. The table in Fig. 4 shows a definition of the abstract effect.

3.1 Correctness

Given an abstraction function α, it is clear that functions effect^{α} and test^{α} may be arbitrarily defined. However the interest of the approach is in preserving some correction properties between $G(M, \mathit{effect}, \mathit{test})$ and $G(M, \mathit{effect}^{\alpha}, \mathit{test}^{\alpha})$. In [10] there is an exhaustive study of the correctness conditions that test^{α} and effect^{α} must verify for $G(M, \mathit{effect}^{\alpha}, \mathit{test}^{\alpha})$ to be a correct over-approximation of $G(M, \mathit{effect}, \mathit{test})$.

s^α	$effect^\alpha(\texttt{i=i+1}, s^\alpha)$	$effect^\alpha(\texttt{i=i-1}, s^\alpha)$
Lower	*Middle*	$\bot$
Middle	*noLower*	*noUpper*
Upper	$\bot$	*Middle*
noLower	*noLower*	*noUpper*
noUpper	*noLower*	*noUpper*
Unknown	*noLower*	*noUpper*

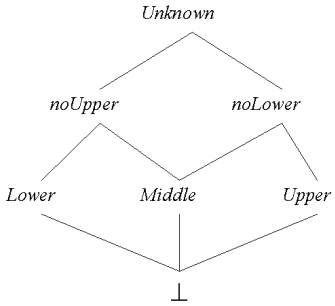


Fig. 4. Part of the abstract effect and the lattice for FLOORS

Correctness conditions guarantee that the reduced/abstract model simulates the original one in the sense that for each non-deadlocked trace $t = s_0 \rightarrow s_1 \rightarrow \ldots \in G(M, effect, test)$ there exists a non-deadlocked abstract trace $t^\alpha = s_0^\alpha \rightarrow s_1^\alpha \rightarrow \ldots \in G(M, effect^\alpha, test^\alpha)$ that over-approximates it, which is denoted by $\alpha(t) \leq^\alpha t^\alpha$, where $\alpha(t)$ represents the abstract trace $\alpha(s_0) \rightarrow \alpha(s_1) \rightarrow \ldots$ and we define $\alpha(t) \leq^\alpha t^\alpha$ as the relation $\forall i \geq 0.\alpha(s_i) \leq^\alpha s_i^\alpha$. Note that we explicitly exclude deadlocked traces because the abstraction process may modify this safety property of the system. For instance, the concrete trace

$$t = 0 \overset{i=i+1}{\rightarrow} 1 \overset{i=i+1}{\rightarrow} 2 \overset{i=i-1}{\rightarrow} 1 \overset{i=i-1}{\rightarrow} 0 \overset{skip}{\rightarrow} \ldots$$

could be approximated by the abstract trace

$$t^\alpha = Lower \overset{i=i+1}{\rightarrow} Middle \overset{i=i+1}{\rightarrow} noLower \overset{i=i-1}{\rightarrow} noUpper \overset{i=i-1}{\rightarrow} noUpper \overset{skip}{\rightarrow} \ldots$$

We have labelled each transition with the basic instruction executed. Note that we have used the table in Fig. 4 to realize each abstract transition.

We implement the loss of information when executing abstract operations using specific abstract constants instead of sets of constants as employed in [9]: for example, when the value *Middle* is incremented, we use the value *noUpper* instead of the set $\{Lower, Middle\}$. When applied to abstract tests, this means that function $test^\alpha$ always produces a safe result, that is, it returns *true* iff in some concrete execution the value *true* may be returned. Thus, given $p \in BExp$, and $s^\alpha \in State^\alpha$, function $test^\alpha$ is defined as

$$test^\alpha(p, s^\alpha) = \bigvee_{\{s \in State.\alpha(s) \leq^\alpha s^\alpha\}} test(p, s) \qquad (Over)$$

In addition, in the following section, we will make use of the abstracted constants to implement the over-approximation method for evaluating temporal formulas.

3.2 Syntactic transformation of Promela

The syntactic transformation of a PROMELA model M to obtain a new model M^α is based on replacing each basic instruction in M by a standard PROMELA code that implements $test^\alpha$ and $effect^\alpha$. Then the verification is carried out by only executing standard PROMELA instructions. This approach corresponds to implementing a verifier for $G(M^\alpha, test, effect)$.

For instance, the next code shows `FLR_INC`, a possible implementation of abstract increment `i = i + 1` defined in Fig. 4.

```
#define FLR_INCR(v) (((x==Lower))->Middle:
                     (((x==Middle))->noLower:
                     (((x==noUpper))->noLower:
                     (((x==noLower))->noLower:
                     (((x==Unknown))->noLower: (ILLEGAL))))))
```

In this code, the constant `ILLEGAL` is employed to represent $\perp$. The code in Fig. 2 is now replaced by the following one, that illustrates the use of the abstract instruction ($effect^{\alpha}$) to increase the variable `Position[]`.

```
proctype Lift(int pid){
int Order=null;
  do
  ...
  :: SysLift_Lift[pid]?Order;
     if
     :: (Order==Up) -> FLR_INCR(Position[pid]);
  ...
}
```

The same method is employed to implement $test^{\alpha}$. For example, the next definition contains the implementation of `FLR_EQ` (abstract test for (i==j))

```
#define FLR_EQs(x,y)        ( (x==Lower && y==Lower) || (x==Upper && y==Upper) )
#define FLR_EQw(x,y)        (((x==Upper)&&(y==noLower)) || ((x==noLower)&&(y==Upper)) ||
                            ((x==Lower)&&(y==noUpper))  || ((x==noUpper)&&(y==Lower)) ||
                            ((x==Middle)&&(y==noUpper)) || ((x==noUpper)&&(y==Middle)) ||
                            ((x==Middle)&&(y==noLower)) || ((x==noLower)&&(y==Middle)) ||
                            ((x==Unknown)) || ((y==Unknown)))
#define FLR_EQ(x,y) (FLR_EQw(x,y) || FLR_EQs(x,y))
```

Function `FLR_EQ` verifies the correctness conditions (studied in [10]) necessary for the abstract model to correctly simulate the original one. Informally, `FLR_EQ(x,y)` is true when $a == b$ holds for some concrete data a and b abstracted by x and y, respectively, as defined in ($Over$) equation. This explains why `FLR_EQ(Upper,noLower)` returns true. The reason for defining `FLR_EQ` using two macros (`FLR_EQs` and `FLR_EQw`) will be explained below.

The user has to select the variables to be abstracted and the abstraction to be applied (α), and then the transformation is automatically performed.

4 Abstracting Temporal Logic

Once the model has been reduced using the method described in the previous section, the following step is to define the satisfaction of a temporal formula over the abstract model (which is called the abstract satisfaction) and to relate it with the satisfaction the formula over the original one.

Atomic propositions in LTL formulas regarding PROMELA models are Boolean expressions. Thus, considering the standard notion of satisfiability given in Fig. 3, and following the same idea used for abstracting the model, we may assert that in order to define the abstract satisfaction of a temporal formula

it suffices to define the abstract satisfaction of the atomic propositions. One clear possibility is to use the function $test^\alpha$, as defined in the previous section (*Over*), to evaluate the atomic propositions. Using $test^\alpha$ leads us to construct the so-called *over-approximation* method for abstracting temporal formulas, which has been studied at length in [11]. An alternative possibility is to use the following function $test_c^\alpha$ to evaluate the atomic propositions. Given $p \in BExp$ and $s^\alpha \in State^\alpha$, $test_c^\alpha$ is defined as

$$test_c^\alpha(p, s^\alpha) = \bigwedge_{\{s \in State. \alpha(s) \leq^\alpha s^\alpha\}} test(p, s) \qquad (Under)$$

Classic papers integrating model checking and abstraction [2,6] use function $test_c^\alpha$ to evaluate temporal formulas. Function $test^\alpha$ incorporates the dual method that may be of interest in some occasions, as discussed in the next section. Now, we summarize the main theoretical results concerning the relation between the abstract satisfaction of temporal formulas over the abstract model (using both the classic and the over-approximated method) and the satisfaction over the concrete model.

In the rest of the section, we write:

(i) $s \models p$ when $test(p, s)$ holds,

(ii) $s^\alpha \models^\alpha p$ when $test^\alpha(p, s^\alpha)$ holds, and

(iii) $s^\alpha \models_c^\alpha p$ when $test_c^\alpha(p, s^\alpha)$ holds.

We also extend $\models$, $\models^\alpha$ and $\models_c^\alpha$ to abstract traces defining the meaning of temporal operators as in Fig. 3. Note that, for instance, when we write $M^\alpha \models_c^\alpha \forall f$, we mean that $\forall t^\alpha \in G(M, effect^\alpha, test^\alpha).t^\alpha \models_c^\alpha f$, and so on.

The following theorem presents two direct results of the previous definitions. In the theorem, we assume that $G(M, effect^\alpha, test^\alpha)$ is a correct over-approximation of model $G(M, effect, test)$ in the sense described in the previous section, and that the original model is deadlock-free.

Theorem 4.1 *Given a temporal formula f*

$$M^\alpha \models_c^\alpha \forall f \Rightarrow M \models \forall f$$

$$M^\alpha \not\models^\alpha \exists f \Rightarrow M \not\models \exists f$$

The first result corresponds to the classic weak preservation of universal properties studied in [6]. Using the classic methodology, the satisfaction of universal properties is directly preserved from the abstract to the concrete model. The second result is the dual preservation result. Using the over-approximation method, the refutation of existential properties is directly preserved from the abstract to the concrete model. Note that these results are not equivalent because they deal with negation using non-standard and dual approaches. Given a proposition p and an abstract state s^α, using definition (*Under*), it is possible that for the classic method neither $test_c^\alpha(p, s^\alpha)$ nor $test_c^\alpha(\neg p, s^\alpha)$ hold. In contrast, due to definition (*Over*) for the over-

approximation method, it is possible that both $test^{\alpha}(p, s^{\alpha})$ and $test^{\alpha}(\neg p, s^{\alpha})$ hold. This is why we skipped the negation rule from Fig. 3. Thus, considering that formula $\neg f$ is in negation normal form, we have that $M^{\alpha} \not\models_{c}^{\alpha} \forall f \not\Rightarrow M^{\alpha} \models_{c}^{\alpha} \exists \neg f$, and, in addition, $M^{\alpha} \models^{\alpha} \forall f \not\Rightarrow M^{\alpha} \not\models^{\alpha} \exists \neg f$.

4.1 Syntactic transformation of LTL

The syntactic transformation of temporal formulas is straightforward on the basis of the previous discussion. The first step consists of writing the formula in negative normal form (if necessary). Then the propositions are automatically replaced by the abstract implementation of the *test*, depending on the method to be employed. For the implementation of the over-approximation method, propositions are defined using the same definition of $test^{\alpha}$ employed to transform the model. But the implementation of $test_{c}^{\alpha}$ must be more restrictive than $test^{\alpha}$ in order to ensure the criterium defined above ($Under$). A definition like

```
#define FLR_EQs(x,y) ((x==Lower&&y==Lower)||(x==Upper&&y==Upper))
```

implements the (classic) abstract test for (i==j). Informally, `FLR_EQs(x,y)` is true when $a == b$ holds for every concrete data a and b abstracted by x and y, respectively, as defined in ($Under$). Note that `FLR_EQ` uses the two macros `FLR_EQs and FLR_EQw` in order to consider the cases where only some concrete states satisfy (i==j).

5 Using αSpin: A case study

In this section, we describe the main functionality that αSpin adds to Spin/XSpin Our case study is a variant of the PROMELA code for an elevator controller presented in [8]. In this section, we show how to employ the dual approaches for the refutation and verification of temporal properties.

The first experiment is to discard errors by refutation (with the over approximation method). The second one consists of checking a desired universal property with the classic method.

5.1 The model

The original specification considers a controller system to manage n lifts, and our aim is to verify that the same control structure also works for only one lift. Following the rules about how to construct suitable models for verification, we have made a set of changes to work with one lift (see Fig. 5). The input to the system is modelled by a process that produces user requests from inside the lift (internal requests). The lift is represented with the `Lift()` process that receives orders to move `up`, `down` and `stop`, thus updating the `Position` of each one. The control part receives the inputs and sends the orders to the `Lift()` process. This part is divided into several processes (`SysLift()`,

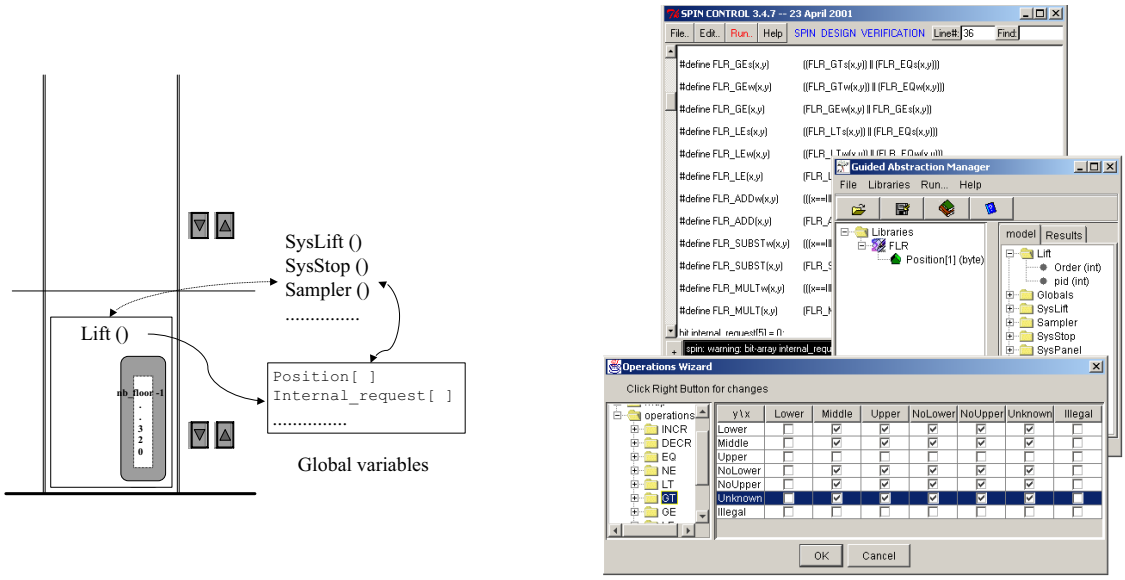


Fig. 5. a) Scheme of the lift system b) One view in αSpin

`SysStop()`, and `Sampler()`) that communicate via rendezvous channels and global variables. The main variable to control the flow in every process is the global variable `Position`, that always stores the current floor for the lift. The global array `internal_request[nb_floor]` stores the pending requests to move to specific floors, `nb_floor` being the actual number of floors in the system. The code in Fig. 2 shows the updating of this variable in the `Lift()` process depending on the order from the control part (`Up`, `Down`, `Stop`).

5.2 Discarding errors

One critical property to check whether the control system works properly is the absence of movement in the absence of requests. The property `NoMove` says that *"the lift never starts the movement without any request"*. If we want to check the property when the lift is on the lower floor, we can encode it as the temporal formula

```
NoMove: <> (posL && no_request && <> posAboveL)
```

and then we can use Spin to verify that there are no executions satisfying the formula (done in Fig. 2), where the propositions `posL` and `posAboveL` represent whether the lift is currently on or above the lower floor, and `no_request` represents that there are no users for the lift. These propositions are defined according to their interpretation standard or abstract as defined in Section 4.

The main problem in verifying the concrete model (with standard meaning for propositions) is that the verification time is highly dependent on the number of floors, and it is not scalable when this parameter is increased to high values. Fortunately, propositions in the formula `NoMove` give us a guide on how to abstract. As the evaluation of these propositions mainly

relies on the value of the variable `Position[]`, and this variable is used as a counter, we could employ the `FLOORS` abstraction to reduce the state space to be visited. However, the use of `FLOORS` implies that the global array `internal_request[nb_floor]` has to be abstracted by an array with only three components. This information is suggested by the abstraction tool by analyzing the structure of the model, and it can also be guessed by the user from the output like the one in Fig. 5. The GUI gives information about the variables contained within the model (name, type and context: global or local), the available templates in the abstraction library suitable for the variables in the temporal formula and the current binding of variables to abstraction functions (`Position[1]` will be abstracted using FLR). When the abstraction functions have been selected, αSpin performs the syntactic transformation of the model depending on the user's choice. When the choice is `Property holds for No Executions (error behaviour)`, as shown in Fig. 6, the code is produced to employ the over-approximation of the formula. The macros `FLR_EQ, FLR_GT, FLR_NE, ..` implement this over-approximation, $test^{\alpha}$, as described before. As shown in Fig. 6 the error is not present either in the abstract model or, using Theorem 4.1, in the concrete model.

The benefits of using the abstract formula to discard this error are summarized in Fig. 8. The formula employed to check movement is the previous one extended to also consider departure for upper and middle floors. The expected number of visited states is greatly reduced compared to the concrete model (see Fig. 8). Furthermore, the variation of the number of states is linear with respect to the number of floors.

5.3 Verification of a desired behaviour

After discarding the key critical error behaviors, we proceed by verifying that the lift system works to provide the intended service. The property `Move` says that *"the lift always starts the movement to the requested floor"*

The version of the property as a desirable behaviour could be as follows:

```
Move: [] ((reqL && posU ) -> <> posBelowU) && ((reqU && posL) ->
          <>posAboveL) && ((reqM && noPosM)-> <> posM))
```

where the propositions `reqL`,`reqU` and `reqM` represent requests from *Lower*, *Upper* and *Middle* floors, respectively. Propositions `posU`, `posBelowU`, `posL`, `posAboveL`, and `noPosM` represent whether the lift is currently at, above or below, a specific floor. Again, these propositions are defined according to the interpretation standard or non-standard, depending on how the verification is to be performed (with concrete or abstract model).

The model is transformed (automatically) in the same way that the refutation case, but when selecting `Property holds for All Executions (desired behaviour)`, the formula is transformed (automatically) to employ the classic method. Note that in Fig. 7 the propositions in the formula are defined using the macros `FLR_EQs, FLR_GTs, FLR_NEs, ...` that implement the classic

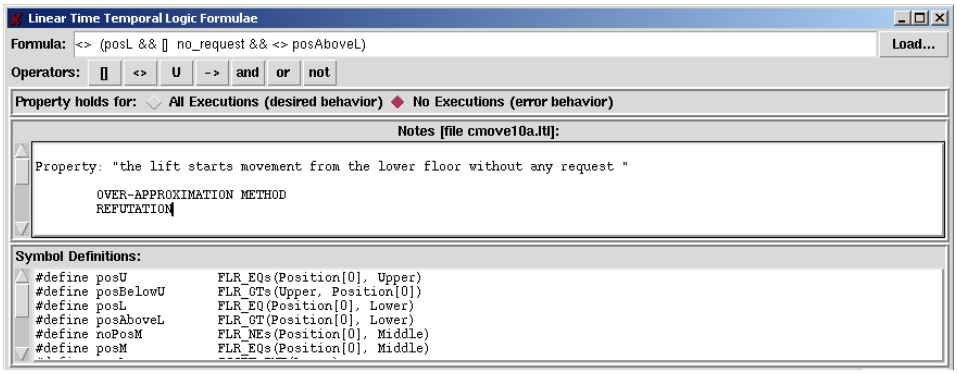


Fig. 6. Refutation of erroneous behaviours

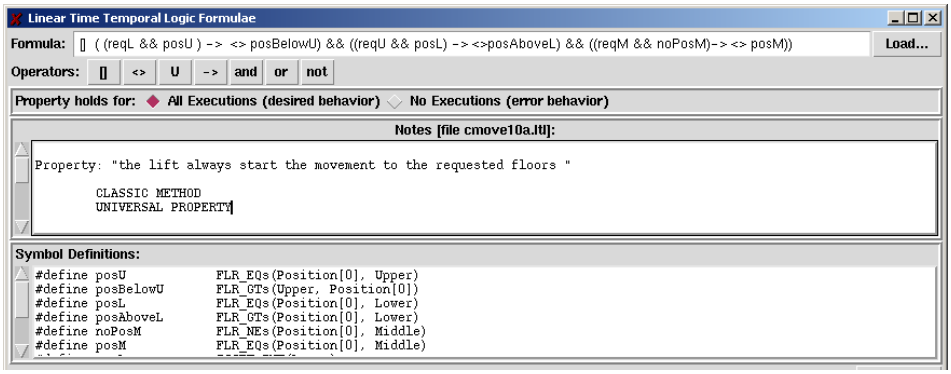


Fig. 7. Verification of desired behaviours

under-approximation $test_c^{\alpha}$. Now the verification result "valid" in the abstract formula ensures that the concrete model satisfies the property. The benefits of verifying with this method are shown in Fig. 8.

6 Implementation

The main design criteria in our abstraction tool is to obtain as much independence with respect to particular modelling languages and model checkers as possible. So we consider XML as the unique internal representation to perform the abstraction by transformation as shown in Fig. 9. The actual modelling language can be translated into this representation by a front-end module (steps 1 and 2 in Fig. 9) and the final abstracted model for the model checker can be produced by a specific back-end module (steps 6 and 7). Furthermore, if we use the same internal notation for both models and abstraction functions, we can concentrate efforts in developing reusable techniques and uniform tools for transformation based abstraction.

In addition to practical reasons like the use of browsers and other user-friendly presentation tools, the development of XML oriented tools is supported by a number of more technical reasons (see [12]). As every model checker uses a particular input, from the point of view of the modelling language, each one has a specific parser and additional support to convert the model

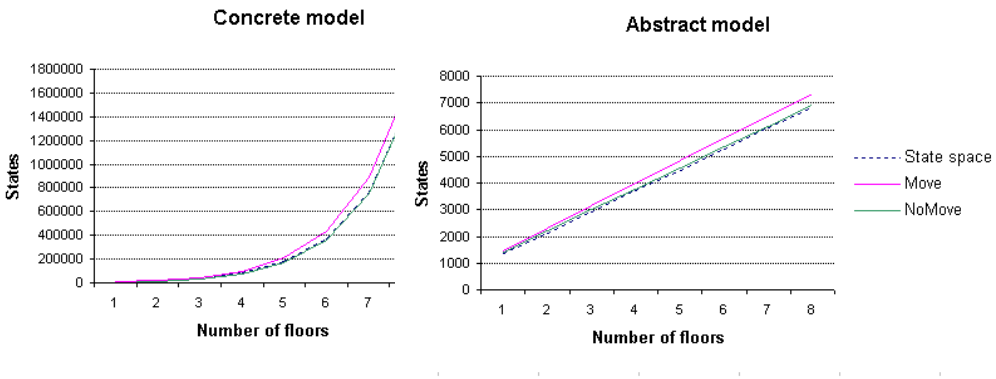


Fig. 8. Verification results

specification into a suitable internal data structure for the model checking phase. Unfortunately, it is not a common practice to have access to this internal representation, because model checking tools are source-closed or not flexible enough for implementing data transformation or manipulation via a set of APIs, as required in abstraction. Even in cases of open-source projects like Spin, most of the work to perform abstraction cannot be directly reused for other model checkers. In addition, the XML representation of the model facilitates traditional tasks in abstraction tools, such as finding relationships among variables or locating the points where a particular variable is employed.

As regards abstraction functions, XML is a powerful means to represent the mapping between concrete and abstract data and abstract operations, including details such as the type of the operands, associativity rules, etc (see Fig. 10). Furthermore, the whole abstraction library can be defined as an XML repository.

The current implementation is composed of the modules shown in Fig. 9. Most of them have been completed, and we are now working on the abstraction prover, that will assist in generating new correct abstraction functions to be included in the library.

7 Concluding Remarks

The main contribution of this paper is the presentation of a tool to perform abstraction by syntactic transformation in the context of explicit model checking. We have presented the actual state of αSpin, a tool that integrates the classic method for abstraction and our over-approximation method. Documentation and current and future versions of αSpin can be found at [20].

The theoretical approach that support the transformation gives us a safe framework to extend implementation preserving the relation between the results in the abstract and the concrete models (and formulas). For example, we have implemented a method to check *existential properties* (those that are true for at least one path) [6]. To do that, the model has to be transformed

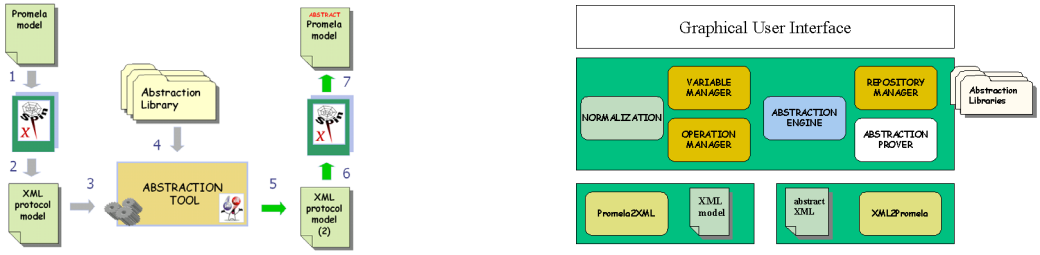


Fig. 9. Architecture of XML based abstraction and modules of αSpin

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE abstractionRepository (View Source for full doctype...)>
- <abstractionRepository>
  - <library name="FLR" concreteDomain="INT">
    - <map>
      - <abstractValue name="Lower" infoLevel="1">
        - <expression type="EQ">
          + <left>
          - <right>
              <const value="min" />
            </right>
          </expression>
        </abstractValue>
      - <abstractValue name="Upper" infoLevel="1">
        - <expression type="EQ">
          + <left>
          - <right>
              <const value="max" />
            </right>
          </expression>
        </abstractValue>
        <abstractValue name="Middle" infoLevel="1" />
        <abstractValue name="NoLower" infoLevel="2" moreImprecise="Middle Upper" />
        <abstractValue name="NoUpper" infoLevel="2" moreImprecise="Lower Middle" />
        <abstractValue name="Unknown" infoLevel="3" moreImprecise="NoUpper NoLower" />
        <abstractValue name="Illegal" infoLevel="4" />
      </map>
```

```
    - <operation type="INCR">
      - <variant type="WEAK" checked="NO">
        - <option>
            <test lval="Lower" effect="Middle" />
          </option>
        - <option>
            <test lval="Middle" effect="NoLower" />
          </option>
        - <option>
            <test lval="NoUpper" effect="NoLower" />
          </option>
        - <option>
            <test lval="NoLower" effect="NoLower" />
          </option>
        - <option>
            <test lval="Unknown" effect="NoLower" />
          </option>
          <default return="Illegal" />
        </variant>
      </operation>
    + <operation type="DECR">
    + <operation type="EQ">
    + <operation type="NE">
    + <operation type="LT">
    + <operation type="GT">
    + <operation type="GE">
    + <operation type="LE">
    + <operation type="ADD">
    + <operation type="SUBST">
    + <operation type="MULT">
    </library>
```

Fig. 10. Part of the XML based abstraction library

using the most constrained versions of abstract *effect* and *test* ($effect_c^\alpha$ and $test_c^\alpha$), and the formula is transformed using $test_c^\alpha$. We are now extending the theoretical framework to give support to new transformations (a related work can be found in [4]).

Other interesting contributions are the use of abstraction libraries and the use of XML to support the abstraction process. The library should be employed to store new functions that are revealed as useful in the verification experiences. It is even possible to give a taxonomy to these functions to make their use easier [14]. Again, XML is a good candidate to store this information.

Our future work is to add strategies to automatically analyze the correctness of abstraction functions using PVS. Another line of work is to improve counterexample analysis [3].

Acknowledgements We would like to thank the referees for their helpful and constructive comments.

References

[1] E.M. Clarke, E. A. Emerson and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications, *ACM TOPLAS*, 8(2),(1986).

[2] E.M. Clarke, O. Grumberg and D.E. Long. Model Checking and Abstraction, *ACM TOPLAS*, 16(5), (1994), 1512–1245.

[3] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-guided Abstraction Refinement, *Proc. of the 12th CAV*, LNCS-1855, pp. 154-169, (2000).

[4] R. Cleaveland, S.P. Iyer and D. Yankelevich. Optimality and abstraction in model checking. In A. Mycroft, editor, *Static Analysis Symposium*, LNCS-983, pp. 51-63, (1995)

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in *Conf. Record of the 4th ACM Symp. on POPL*, pp. 238-252, (1977).

[6] D. Dams, R. Gerth and O. Grumberg. Abstract Interpretation of Reactive Systems, *ACM TOPLAS*, 19(2), (1997), 253–291.

[7] D. Dams. Abstraction in Software Model Checking: Principles and Practice, in *9th Int. SPIN Workshop. Model Checking Software*, LNCS-2318, pp. 14-21, 2002.

[8] G. Duval and T. Cattel. From Architecture down to Implementation of Safe Process Control Applications. Design, Verification and Simulation. In *Proc. of the 13th Annual Hawaii Int. Conf. on System Sciences (HICSS 30)* (1997).

[9] M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Pasareau, W. Visser, H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. *Proc. of ICSE 2001*, 2001.

[10] M.M. Gallardo and P. Merino. A Framework for Automatic Construction of Abstract PROMELA Models. In *Theoretical and Practical Aspects of Spin Model Checking*, LNCS-1680, pp. 184-199, (1999).

[11] M.M. Gallardo, P. Merino and E. Pimentel. Verifying Abstract LTL Properties on Concurrent Systems *Proc. of the 6th World Conference on Integrated Design & Process Technology*. (2002). To appear.

[12] M.M. Gallardo, J. Martinez, P. Merino and E. Rosales, Using XML to implement Abstraction for Model Checking. In *Proc. of ACM Symposium on Applied Computing*, pp. 1021-1025 (2002).

[13] S. Graf. Verification of a distributed Cache Memory by using abstractions. In Dill, D., (Ed.) *Computer Aided Verification* , LNCS-818, pp. 207-219, (1994).

[14] S. Graf. Personal Communication. 2002.

[15] Havelund K., Pressburger T., Model Checking Java Programs using Java Path Finder. *Software Tools for Technology Transfer (STTT)* 2(4):366-381, (2000).

[16] G.J. Holzmann. Design and Validation of Comp. Protocolos. Prent-Hall, (1991).

[17] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. on SE*, 23(5), (1997).

[18] G. J. Holzmann and M. H. Smith. A Practical Method for the Verification of Event Driven Systems. In *Proc. of ICSE99*, pp. 597-608, (1999).

[19] W3Consortium. Extensible Markup Language (XML) 1.0 (Second Edition), available in: *http://www.w3.org/XML/*, (2000).

[20] αSpin project. University of Málaga. http://www.lcc.uma.es/˜ gisum/fmse/tools

Validation and automatic test generation on UML models: the AGATHA approach

David Lugato - Céline Bigot - Yannick Valot

CEA/LIST/DTSI/SLA
CEA Saclay – Bat. 451
91191 Gif sur Yvette Cedex

{david.lugato, celine.bigot, yannick.valot}@cea.fr

Abstract

The related economic goals of test generation are quite important for software industry. Manufacturers ever seeking to increase their productivity need to avoid malfunctions at the time of system specification: the later the defaults are detected, the greater the cost is. Consequently, the development of techniques and tools able to efficiently support engineers who are in charge of elaborating the specification constitutes a major challenge whose fallout concerns not only sectors of critical applications but also all those where poor conception could be extremely harmful to the brand image of a product.

This article describes the design and implementation of a set of tools allowing software developers to validate UML (the Unified Modeling Language) specifications. This toolset belongs to the AGATHA environment, which is an automated test generator, developed at CEA/LIST.

The AGATHA toolset is designed to validate specifications of communicating concurrent units described using an EIOLTS formalism (Extended Input Output Labeled Transition System). The goal of the work described in this paper is to provide an interface between UML and an EIOLTS formalism giving the possibility to use AGATHA on UML specifications.

In this paper we describe first the translation of UML models into the EIOLTS formalism, and the translation of the results of the behavior analysis, provided by AGATHA, back into UML. Then we present the AGATHA toolset; we particularly focus on how AGATHA overcomes several problems of combinatorial explosion. We expose the concept of symbolic calculus and detection of redundant paths, which are the main principles of AGATHA's kernel. This kernel properly computes all the symbolic behaviors of a system specified in EIOLTS and automatically generates tests by way of constraint solving. Eventually we apply our method to an example and explain the different results that are computed.

Keywords : UML specification, automated test generation, symbolic calculus.

1. Introduction

Formal methods allow system analysis and test generation from specifications. This provides an early feedback on a system's behavior. The economic goal of this specification analysis step is considerable, as it simultaneously reduces cost and time of validation, while increasing system reliability. But these formal techniques are generally quite complex: this is why such techniques have not, at this time, penetrated the industrial domain. Therefore, it is crucial to provide tools in which these techniques are automated.

It is also well known that the difficulty of analyzing a system depends on the "quality" of the specification. That's why it's crucial to observe a few rules while specifying a system. Because general UML models still have a lot of points with variable, open or undefined semantics [1], formal analysis requires respecting modeling rules and some UML specialization. These specializations are attached or dedicated to the European project AIT-WOODDES [2].

Methods and tools have been developed to analyze systems using their specification (in order to prevent unexpected behaviors) and to generate tests (to guarantee the fitness of the implementation to the model). Tools such as AGATHA generate test sets allowing to validate that the software implementation is conformant to its specification (black box testing). As it also generates a symbolic execution tree, AGATHA allows deep investigation into the system's behaviors. To produce these results AGATHA has to deal with combinatorial explosion. We will see in the second part of this paper how AGATHA overcomes this problem. The AGATHA toolset is a melting-pot of different techniques, as in [3]. The kernel is based on symbolic calculus, detection of interleaving, constraint solving, rewriting procedures, polyhedral calculus... Like [4], the AGATHA toolset generates a set of tests for UML statecharts, but it does not need test requirements to compute an exhaustive symbolic path coverage. Note there are also several differences on the UML semantics used in the Hartmann's tool and in the tool presented here.

The first step of our work is to develop an interface between UML and the A-EIOLTS (AGATHA Extended Input Output Labeled Transition System) language used by AGATHA, being especially careful to respect the peculiarities of the semantics of each language. We implemented the resulting translation algorithms in the Objecteering 5 UML modeling tool [5].

Formal validation of specification as well as software testing usually require high skills, time and staff. In this paper, we discuss the new features added to AGATHA in order to use it in a transparent way and to exhaustively compute the behaviors of the specification.

We wish to promote an incremental way of elaborating a specification. As will be demonstrated, the toolset helps engineers in formally validating the developed systems at any step. We would like to insist on the transparency of using AGATHA to validate a UML specification. Thanks to the complete automation of AGATHA techniques, developers will be able to validate a specification while staying in the UML CASE tool used for modeling and then also generate tests for the implementation.

2. Transcription UML models to A-EIOLTS

We connect the AGATHA toolset to the environment of the AIT-WOODDES project that offers a method for designing UML specification, an automatic code generator and validation tools. In this context we generate tests for UML models designed with the ACCORD methodology [6]. The accepted UML models are designed with class diagrams. Each class should have one or more statechart diagram that represents its dynamic behavior. Collaboration diagrams are used to model interactions between instances of classes. The results provided by AGATHA will be turned into UML sequence diagrams.

2.1 Two step process transcription

The translation from UML to A-EIOLTS is a two-step process. First, the UML specification is checked against consistency rules to verify that the translation modules will be able to translate the specification to A-EIOLTS; this module also transforms the UML model into another UML model, of equivalent semantics, but using only a restricted set of UML's elements. A second module then translates this restricted UML into an A-EIOLTS file. In the following sections we only describe the second-step translator.

Another module can analyze the resulting file and bring the results back into the Objecteering CASE tool, for instance animating the statecharts to show the execution of the state machines for the objects involved in a given test case (see Fig.1).

The subset of UML that is used is designed to achieve the same level of simplicity in the description of the state machines than the A-EIOLTS input language of AGATHA. The second step converts the "simplified UML" into the A-EIOLTS file proper.

In this project, class diagrams are used to represent the classes involved, a collaboration diagram shows the messages exchanged by the different instances of these classes, and for each class a state machine and its state diagrams show the behavior of the objects. Sequence diagrams can be used as a feedback to represent the different possible tests provided by AGATHA.

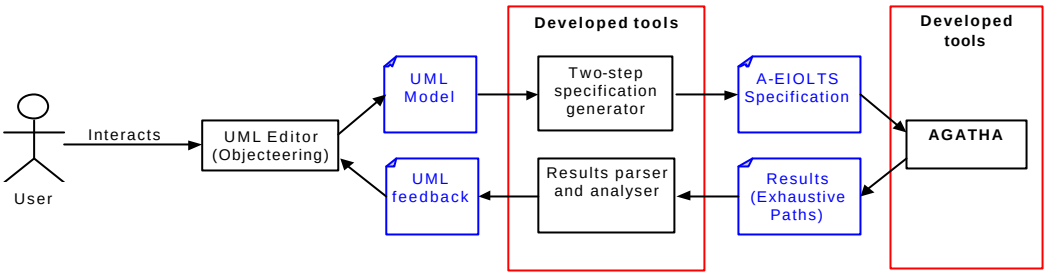


Fig.1 – Interfacing UML modeling and AGATHA

2.2 Active objects

UML defines a category of objects called **active objects**. Each active object has its own processing resource (typically, they have their own task, process or thread). As a result, active objects can run concurrently with others. They are opposed to passive objects, which have their own data but are carried out only when there are called by an active object that lends its thread to the passive object in order to execute the requested action.

Active objects, when associated with a UML state machine, have an event queue that allows them to store incoming events until the state machine is able to handle them. In this project, the translated models must contain **only active objects**.

2.3 AGATHA's input language

We describe here only the general principles of A-EIOLTS. This formalism is inspired of a simplified version of the ESTELLE language [7].

The hierarchical module structure is limited to a flat structure with only communicating controllers at the lowest level. Each module is composed with the declaration of I/O messages and variables, the list of nodes of the automata and of course the list of transitions between these nodes (see Fig.2 for an example of an A-EIOLTS transition).

```
TRANS
FROM state1
TO state2
WHEN input(x)
PROVIDED x > 0
OUPTUT ok
BEGIN
a := a + x ;
END;
```

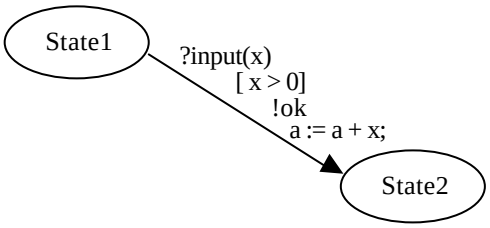


Fig.2 – Example of an A-EIOLTS transition

The following restrictions apply as well:

- Communications between modules are limited to synchronous rendezvous,
- Multiple rendezvous are not allowed: a rendezvous must entail only two automata (or modules, sender and recipient; neither multiple recipients nor broadcasting of messages are supported).

When the recipient for a message is a module, `OUTPUT` instructions lock their module until a rendezvous occurs, if any. On the other hand, a message sent to the environment or received from the environment is considered sent asynchronously and therefore non-blocking. Since rendezvous must include only two modules, at a given time, a module can send only one message to another module.

Since outputs are locking, it is no longer possible to follow the semantics of extended transition systems. In extended transition systems, you can send a message within the actions of a transition, those actions being no longer limited to assignments. To reproduce this semantics, it is necessary to create intermediary states. Thus, fusion of the controllers becomes statically computable, the rendezvous no longer depending on the actions.

Variable management is performed in the transition's body, using level 0 PASCAL instructions. The actions that can be specified on a transition are restricted to the following set:

- Variables, for instance `X`, `Y`, …
- Functions : `+|-|OR|AND|… (operators)`
  `0|1|…|TRUE| (constants)`
- Expressions `X:=E (assignments)`
  `C;C'.(sequencing)`
  `IF E THEN C;ELSE C'(conditional test)`

Nevertheless, it is important to note that this subset allows a user to express any complex instruction.

Guards ('PROVIDED') are of logic type, but notice that temporal guards have been added in order to validate SDL specifications [8].

Global variables must be avoided as much as possible, due to a particularly important risk of combinatorial explosion. Note that the use of global variables is groundless from a behavioral standpoint.

2.4 Defining a restricted UML state machine

We define a **restricted UML state machine** (or simplified UML), which is a restriction of the set of UML concepts related to state machines. Any state machine can be converted into this subset, without modifying the semantics.

To be easily translatable to A-EIOLTS, the restricted UML must be of similar complexity. Thus only simple states and simple transitions are supported. The event-handling mechanism for UML state machines is linked to UML objects, and cannot be changed. Therefore, like simple states and transitions, the event-handling mechanism is a fundamental element of UML semantics and is kept in the semantics of restricted UML

In the UML specification, a call event represents the reception of a request to synchronously invoke a specific operation. A-EIOLTS only supports one call event per transition. Actions to be executed on a transition can only consist of **one** action, of type CallEvent. It would have been possible to add operation calls towards the environment, but we keep things simple by allowing only one operation call per transition, thus suppressing the need to discriminate between operation call recipients. Note that an object calling one of its own operations is considered as a CallEvent.

According to the restrictions of A-EIOLTS, it is not possible for an output to be part of the A-EIOLTS transition's actions. One direct consequence is that a CallEvent cannot be the result of a conditional expression inside the actions of the UML transition.

Moreover, AGATHA always sends the `OUTPUT` message first, and then executes the action. This prevents a message from being sent after several actions took place (in particular, the parameters of the message will not depend on the actions in those transitions). In short, restricted UML only allows **either one CallEvent OR (exclusive) a series of assignments** (see Fig.3 for an illustration of accepted transition).

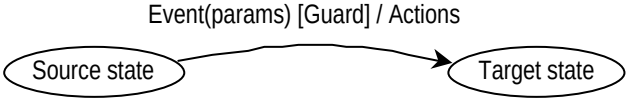


Fig.3 – A simple transition with its label

Finally, restricted UML is defined by the following rules:

- Only **simple states** are supported (no composite states),

- Only **simple transitions** are accepted (no pseudo-state except the initial pseudo-state),
- **Actions** are accepted only on **transitions** (no activity, no entry and no exit actions on states),
- **No Call Action** within a **conditional test** (IF-THEN-ELSE),
- Actions on a transition are either one single CallAction or (exclusively) **a series of assignments** and conditional tests separated by semicolons (";").

UML active objects or A-EIOLTS modules are executed concurrently in an asynchronous manner. But for UML active objects communication is asynchronous and for A-EIOLTS sending a message blocks the source module until the message is received by the target module (synchronous rendezvous). Therefore, the mechanisms involved in UML event processing are translated precisely into an A-EIOLTS description, in order to get the same communication semantics. In the next subsection we introduce the translation of this mechanism and then we introduce the concept of execution models, which is related to the way translation must be carried out.

2.5 Splitting the objects

According to the UML specification (OMG-UML V1.3, §2.12.4 – Semantics), a state machine (which can be used to model the behavior of an active object) is composed of three elements: one structural element and two processing elements.

UML gives this representation as an example only, noting that any other mechanisms achieving the same semantics would be conformant to the specification. But this example is very close to what A-EIOLTS enables, and so is our implementation .

The three elements are defined as follows:

- An *event queue* that holds incoming events instances until they are dispatched;
- An *event dispatcher mechanism* that selects and de-queues event instances from the event queue for processing;
- An *event processor* that processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question; because of this, the UML specification calls it the "state machine".

Therefore we naturally attach two A-EIOLTS modules for each UML active object:

- The first module is the **event processor**. Its A-EIOLTS specification is globally similar to the corresponding state machine, even if a close view will reveal minor changes (transitions split into several smaller transitions, additional states, added control messages, etc…). The event processor knows about the behavior of a given active object, its states and its transitions.
- The second module is the **event dispatcher**, which implements asynchronous communications on top of A-EIOLTS's synchronous rendezvous model. The event dispatcher must be ready at any time to receive events from any source, even if the event processor is not ready to handle them because it is already processing another message. In order to store the events it receives, the event dispatcher has to implement the **event queue** inside its module. The event dispatcher does not know the structure of the state machine; on the other hand it knows which events the event processor may receive, although it does not know *when* it may receive them.

The first execution model that has been implemented includes a First-In First-Out queue (see Fig. 4 for an overview of this decomposition). This decomposition corresponds to the structure proposed by the UML standard. The event dispatcher receives all the events. If the event processor is busy, the dispatcher stores the event for later processing; otherwise the event is transferred directly to the processor.

Therefore, the event dispatcher acts as an input interface for the active objects. Outputs, on the other hand, are sent directly by the event processor to the other active objects. In the case when the event processor must send an event to itself, it will in fact send it to its dispatcher, just as if it were another active object.

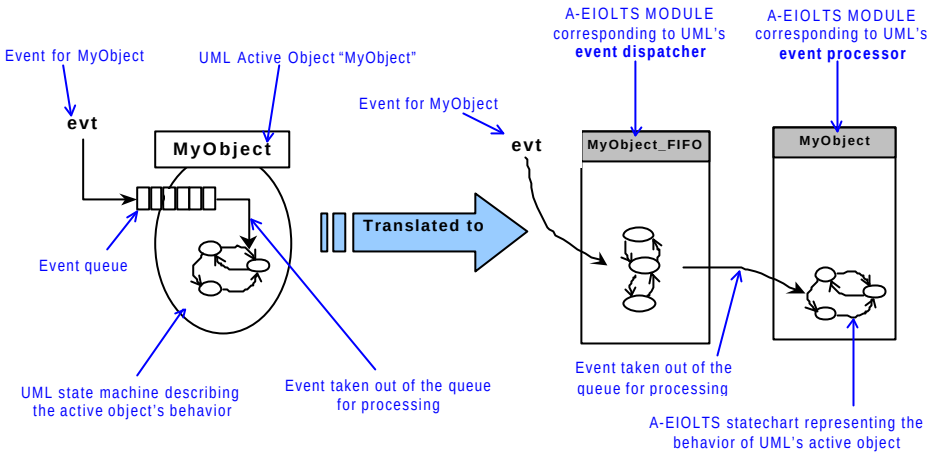


Fig. 4 – Decomposition of a UML active object into two A-EIOLTS modules

2.6 The Execution Models

UML restrictions impose the sketch lines of event handling, but many of the details are left to the implementor's discretion. Since our goal is to analyze the precise behavior of a system, we must impose the precise details of the execution model.

Details described in such execution models include, but will not be limited to, the handling of events. The event dispatcher will gain modularity if seen as a black box. We try to stick with this view as much as possible, although we initially use a FIFO list for our dispatcher.

"The processing of a single event by a state machine is known as a ***run-to-completion*** step. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all actions (but not necessarily activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion* set *is* the passage between two state configurations of the state machine."

(OMG-UML V1.3, §2.12.4.7)

The meaning of this is that an active object only processes one event at a time. It can, though, receive other events during that time and store them for later processing.

While the event processor is handling an event, it cannot process another one: the event dispatcher will queue any incoming events. All incoming events targeted at the processor will pass through the event dispatcher first, so the processor will never receive incoming events from something else than its dispatcher. Therefore, the dispatcher knows with certainty when the processor *enters* the RTC step, because it has just sent the corresponding event. Now, if we provide a way for the event processor to tell the dispatcher that it *leaves* the RTC step, the dispatcher will have reliable knowledge of when the processor is busy and when it is ready (idle and ready to receive an event).

From that model we can define a generic state machine for an event dispatcher. The dispatcher shown in Fig.5 uses a variable to store the type of message from transition to transition.

A specification containing unexpected and/or erroneous behaviors may lead to the flooding of an event dispatcher. Such flooding will be explored virtually *ad infinitum*, by a test generator toolset. For that reason, we add another safeguard by limiting the size of the FIFOs. When a dispatcher's FIFO is full, the dispatcher will deadlock. This way the execution path will be signaled as faulty

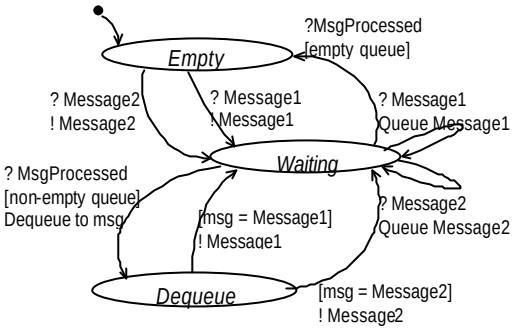


Fig.5 – Generic dispatcher with two possible events

2.7 Transitions and availability

Splitting transitions in the event processor will not really change the semantics of execution. In fact, it will even enhance the simulation. Consider, for instance, that Object 1 sends two events (a, b) to Object 3, and Object 2 also sends one event (c) to Object 3. The apparent randomness of task scheduling can change the exact order in which Object 3 will receive the events (c, a, b / a, b, c / a, c, b). The third case, in order to be simulated by a test generator, requires that a and b be sent on different transitions (and this is forced in A-EIOLTS, only one Rendezvous per transition). In fact, the apparently burdensome restrictions of A-EIOLTS concerning the sending and receiving of events have positive impact, since the forking of execution paths will often come from such reordering of events.

The event dispatcher has a duty towards *all* the event processors: it must **always** be ready to receive an event. But even the event dispatcher needs some time to store, restore or send an event; during that time it is not available. There also are other considerations about exactly what type of communications occurs between active objects. For this reason, it is very common that event queuing and dispatching operations be executed in a **critical section**. A critical section is a section where a thread has **exclusive and absolute priority over all threads in a set of threads**, a section of execution that will not be interrupted until it ends. In the case of queuing and dispatching of messages, the set of threads is the whole system, and such operations are considered globally atomic.

2.8 Generic event processor

As explained earlier, the event processor does not need to be able to receive messages at any time, the dispatcher takes care of that aspect. On the other hand, the processor knows what state the object is in, and what events it may receive. We shall build a generic event processor in several steps. As an example, consider the UML statechart diagram in Fig.6 :

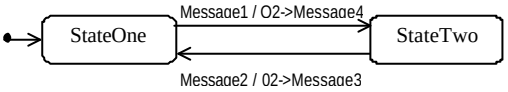


Fig.6 – Sample UML statechart diagram

Now let us translate this into a simple A-EIOLTS statechart diagram. The first problem is that the dispatcher does not know what the processor is ready to receive, which means that if an unexpected message arrives, the event dispatcher will still transmit it to the processor. Indeed, the dispatcher will try to send a message the processor will never receive, since it will be waiting forever for the dispatcher to send *another* message.

It might be interesting for the dispatcher to know that the active object did not change states. In fact, it is interesting for the handling of deferred events, which will be explained further below. To distinguish between messages that make the state machine change states (or, more precisely, that change states *or* perform an external self-transition), and messages that do not, the event processor will return either `MsgProcessed` on state change (or external self-transition), or `MsgAck` for messages that do not cause state change (internal transitions).

Now we can write our new event dispatcher (see Fig.7). This one will not deadlock when the dispatcher sends an unexpected message. Note that the event dispatcher should be changed accordingly to handle the new `MsgAck` callback, however, for the moment, we will not detail it: the only necessary

modification, at this point, is to duplicate all transitions that have `MsgProcessed` as trigger event, creating a twin transition with exactly the same clauses but triggered on `MsgAck`.

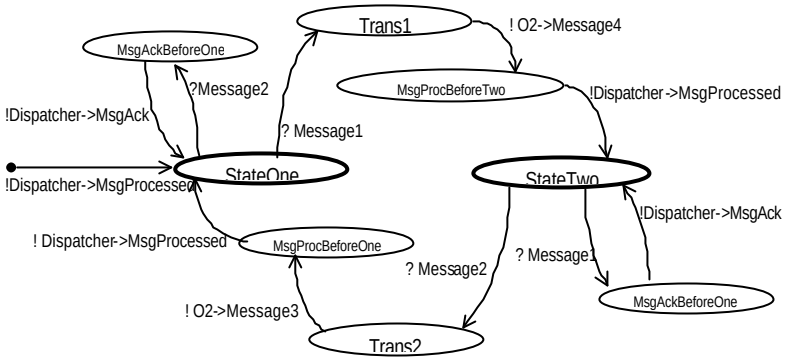


Fig.7 – Event processor capable of handling unexpected events

2.9 Deferred events and parameters

UML includes a notion that is not present in A-EIOLTS: deferred events. For a particular state, it is possible to specify that, although a particular event may not be handled in that state, the object must retain this event. When the state changes (or when an external self-transition is fired), the deferred event is examined again. If, in this new state, the event cannot be handled, the deferred event is consumed without side effect; if it is handled, the corresponding transition is fired. If the event is again deferred, it is stored again for later use, and so on.

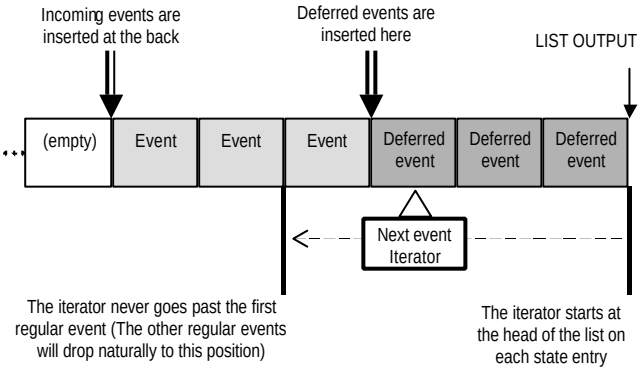


Fig.8 – Theoretical implementation of a FIFO with improved deferred event handling

When the processor consumes an event, deferred or regular, without leaving the state, it is pointless to try again the first events of the queue that were not eligible. Not only so, but if a new deferred event is added while the state machine is in a particular state, it will not be able to fire a transition at least until the next state change. For this reason, we can define an iterator that will "remember" the next event to be processed (see Fig.8).

Upon each state change, the iterator will be reset to the head of the FIFO, conforming to the fact that deferred events have priority. If the iterator reaches the first regular event, it will not go further since there will always be an event in that slot ready to be transmitted to the processor, unless all the non-deferred events have all been processed.

Until now, we have always considered simple events with no parameters, but it may be comfortable for a user to be able to send messages with parameters. Storing parameters in a FIFO is easy. Instead of pushing only the message ID, we push the message ID and all its parameters. When a message has to be popped, the first POP operation will retrieve the message ID. The dispatcher will therefore know how many parameters follow in the FIFO and will immediately pop them out.

2.10 About implementation

The generator has been developed using Objecteering's UML Profile Builder. The profile builder allows the user to extend the capabilities of Objecteering, by either using standard UML extension mechanisms (stereotypes, tagged values…) or adding behavior using the J language.

J is a programming language specific to Objecteering. The main feature of the J language is its ability to navigate the meta-model: the model of the current project is available in memory and navigable according to Objecteering's meta-model, which is very close to the standard UML meta-model.

3. The AGATHA kernel

After presenting the transcription from our UML models to A-EIOLTS, we describe in this section the main principles that AGATHA is based upon, and that keep the combinatorial explosion problem at bay. We shall see how AGATHA uses different academic techniques in order to compute the behaviors of the system.

3.1 AGATHA positioning

There exist several ways to validate systems specifications. A first one consists in theorem proving and model checking [9]. These kinds of techniques have proved successful for the validation of critical parts of systems. But two major drawbacks to these techniques remain: the combinatorial explosion due to variable domains, for the model checking; and a need for high-level skills from the developer –who must be aware of formal methods fundamentals– for theorem proving.

Automatic test generation is another way to tackle the problem of systems validation. Conformance testing is the most well-known part of this domain. Though AGATHA is able to generate tests for the implementation, discussion of this feature falls beyond the scope of this paper. Our first purpose is to validate the specification itself, and by the way generate tests in order to simulate them in the specification.

Most validation tools use enumerative techniques and are therefore limited by the combinatorial explosion problem when trying to exhaustively identify the execution tree of a system. Several validation tools focus verification on particular aspects: test purpose [10], temporal properties [11], etc…

The solution we wanted to build into AGATHA is an exhaustive symbolic path coverage. Notice that this criterion will help, in the future, using AGATHA for verification. If we want to demonstrate the truthfulness of a property on a specification, because of the exhaustivity obtained with AGATHA we just have to demonstrate it on the obtained paths.

The following subsections are an overview of the different academic techniques used in AGATHA in order to reach this exhaustive path coverage.

3.2 Main principle: symbolic execution

AGATHA uses "symbolic execution" as defined by [12], [13], [14]. The major drawback of numeric techniques is the combinatorial explosion due to variable domains. These domains can be huge, sometimes even infinite. Symbolic calculus allows the handling of such domains because computing all the behaviors is not equivalent to trying all the possible values for inputs. Instead of giving values for inputs, they keep their status of symbol all execution long.

So each behavior no longer depends on the result of a calculus being completely performed but on an expression representing constraints on the variables being denoted by the symbols of entries. Each transition fired from a point of the execution adds a new constraint on the variables. The entire constraint, at any point of the execution, is called "**path condition**".

First, a short comparison between a symbolic state and a numeric state: a numeric state is defined by the state in the automata and by the numerical values of the variables, as opposed to a symbolic state, which is defined by the state, the symbolic values of the variables and the path condition (see Figure 9 for a short example).

A symbolic state may represent an infinite set of numeric states. The execution tree that is the result of AGATHA calculus is a finite tree of symbolic states. Because AGATHA is exhaustive and strives to be

minimal, we want the execution tree to be as short as possible. Now if we want to detect as many redundant paths as possible we need to use reduction procedures.

Consider the transition in Figure 2:

TRANS	*For the initial state:*
FROM state1	Numeric State = (s1, 0) for a0 = 0
TO state2	Symbolic State = (s1, a0, true) that includes (s1, 0)
WHEN input(x)	
PROVIDED x > 0	*For the final state:*
OUPTUT ok	Numeric State = (s2, 1) for x = 1
BEGIN	Symbolic State = (s2, a0 + x, x >0) that includes (s2, 1)
a := a + x ;	
END;	

Fig.9 – Comparison between numeric and symbolic

3.3 Reduction procedures

The construction of the execution tree is subordinate to reduction procedures in order to eliminate as many redundant paths as possible with the following tactics:

- Cut "**empty"** path conditions when detected both from a Boolean criteria or polyhedral criteria. We use Presburger tools and theorem provers to achieve that.
- Avoid computation of a path deductible from another modulo a interleaving detection less sophisticated than in [15]: an internal transition without any temporal constraint with other transitions.
- Compute ***comparison procedures*** for each symbolic node and refer to an already existing symbolic.

The n-tuple of a symbolic node is the list of the actual control node for each of the n concurrent modules. These three reduction procedures are necessary to avoid the state explosion problem. We use several different heuristics to compute comparison procedures for each symbolic node:

- *ControlNode procedure*: two symbolic nodes are equivalent if the two n-tuple of control nodes are equal.
- *Inclusion procedure*: two symbolic nodes are equivalent if their n-tuple are equal and if the variables domains of one are included in the other.
- *Equality procedure:* two symbolic nodes are equivalent if their n-tuple are equal and if their variables domains are equal.

But it is sometimes also useful to introduce ***abstractions*** to reduce complexity. We currently work on automating several different abstractions. It is important to notice that in many specifications, there is no human intervention to abstract or to adapt the specification and obtain the results. With an abstract model of the specification, the AGATHA calculus always terminates and therefore the obtained execution graph is exhaustive.

3.4 Simplification procedures

The deeper a point of execution, the bigger the expression representing its path condition. Symbolic expressions of variables may also rapidly grow. That is why a simplification procedure must be applied "on the fly" in order to shorten expressions and detect useless paths [16].

As of today we use a simplifier based on rewriting techniques. The rewriting engine is Brute [17], Brute is a part of the CafeOBJ toolset. The rewriting rules file of AGATHA is actually composed of more than three hundred rules. These rules allow both to maintain symbolic expressions within a reasonable size range, and to obtain normal forms for the expressions, easing the comparison between expressions needed in algorithms such as comparison procedures.

We also use a polyhedric tool, Omega [18], in order to compute the inclusion and equality procedures. Using this tool we are able to compare variables domains of two symbolic nodes.

3.5 Composition

The symbolic execution process is performed on one module, but the global application (historically AGATHA was designed to validate concurrent embedded systems) is generally composed of many, so they have to be merged.

There are two possible ways to merge modules. The first solution is to use the composition introduced by Milner [19]. The global module is made out of the transitions of its components, except those that are synchronized by a rendezvous, which are replaced by an equivalent transition obtained by eliminating the exchanged parameter.

The other solution is to compute the symbolic execution on each module first and then merge the results to obtain the global application behavior. The major benefit of this latter approach is the parallelization of the calculus: execution trees for each module can be computed separately.

At the moment only the first solution is implemented in AGATHA. The second option will be integrated soon. But it is already possible to compute the execution tree on a subset of selected modules of the specification. All the unselected modules are considered as the environment, messages from these modules can occur in all the possible orderings with free parameters.

3.6 Constraints solvers

Once the execution tree is computed, the whole behavior of the system is exhibited. Livelocks and deadlocks are visible. We use the DaVinci [20] graphical interface to represent the execution tree. A constraints solver may then be used to get the appropriate values for symbolic variables satisfying path conditions and generate numerical test input sequences. AGATHA can use two different constraints solvers: the Presburger tool Omega or Con'Flex [21]. We elect to generate one numeric test for each symbolic test. Each symbolic test represents a equivalence class of numeric tests, the constraints solver compute only one solution for each path condition. In the case of a UML specification the format of this numeric test is a sequence chart diagram.

4. Examples

In this section, we present a "toy" example to illustrate the validation and especially the automatic test generation for restricted UML diagrams within the AGATHA toolset.

4.1 The Elevator

We define a simple version of an elevator specification. We define three classes: one for recording stages asked by the user, one for managing the engine of the elevator and one for managing the elevator and the interactions between the stage recorder and the engine manager. We also define two actors that represent external systems: the user and the elevator itself. So we design the class diagram as shown in Fig.10. Moreover and as we said before, we need a collaboration diagram for highlighting the different interactions between classes and external systems (see Fig.10 too).

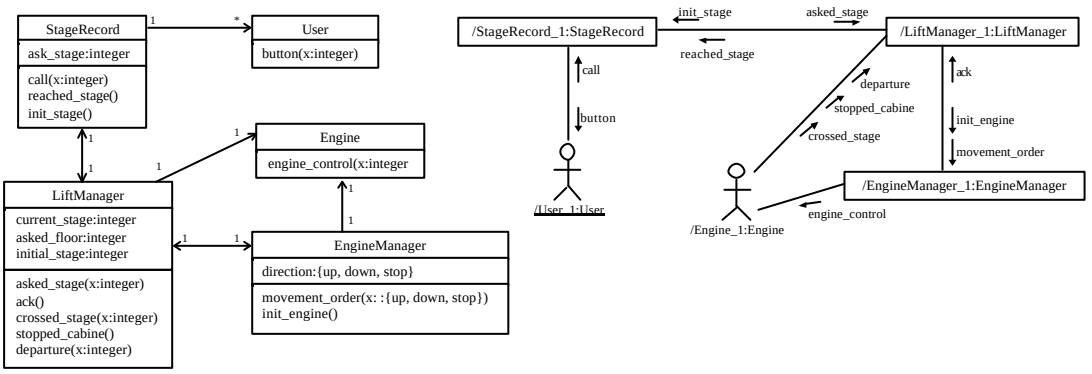


Fig.10 – Class diagram and collaboration diagram

For each class we build a state machine that defines the behavior (see Fig.11 and Fig.12).

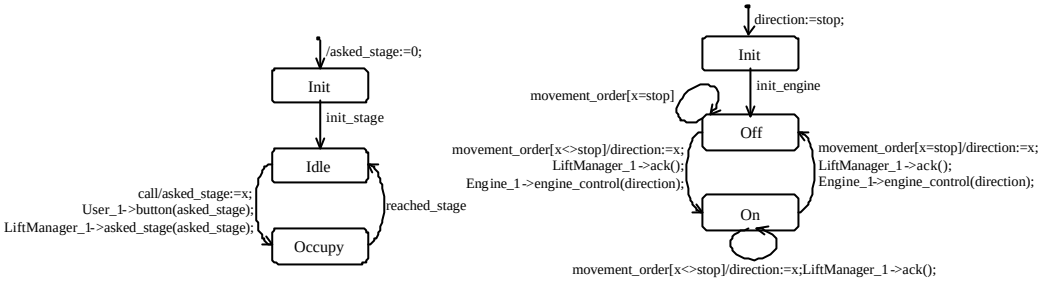


Fig.11 - State machines for the stage recorder (left) and the engine manager (right)

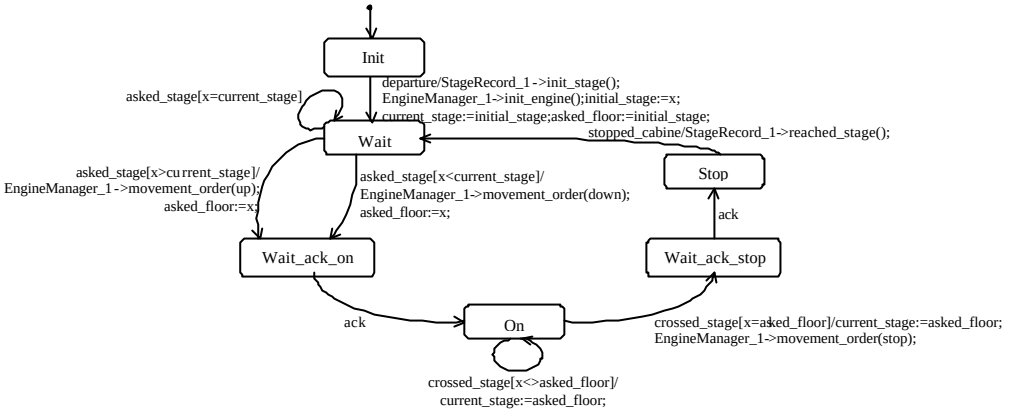


Fig.12 – State machine of the lift manager

4.2 Running the toolset

The AGATHA toolset works with three main steps: the translation of the UML specification into the A-EIOLTS formalism, the generation of the symbolic test cases and the translation of these symbolic test cases into UML sequence diagrams.

4.2.1 Translation

The translation of the UML specification into the A-EIOLTS begins with splitting the initial model. With this first-level translator, composed transitions are split into several transitions. As an example for the stage recorder, the transition between Idle and Occupy is split in 3 sub-transitions with two new states (see Fig.13).

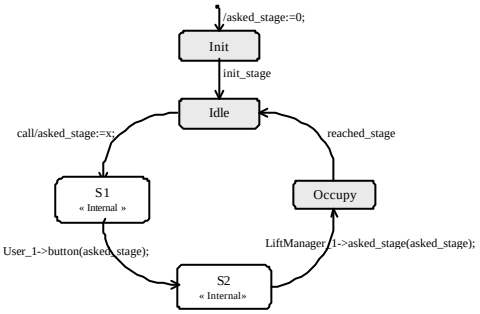


Fig.13 – Split statemachine for the stage recorder

The state machine flattened in a simple diagram can be easily translated in A-EIOLTS formalism. This makes certain transitions atomic and enables more precise analysis of the specification.

The second translator generates the model using an A-EIOLTS formalism. Each class is mirrored by two A-EIOLTS modules: one corresponding to UML's event processor (close to the state machine) and one corresponding to UML's event dispatcher.

4.2.2 Generation of symbolic test cases

The tool computes a symbolic execution tree from the A-EIOLTS specification and each path of this tree represents a symbolic test case.

In this example, let us look closer on the construction of the symbolic tree (see Fig.14). For each symbolic state of the tree we provide the value of variables as a 5-tuple : [StageRecord.asked_stage, LiftManager.current_stage, LiftManager.asked_floor, LiftManager.initial_stage, EngineManager.direction] and we provide the conjunction of all encountered guards (also called path condition).

The symbolic execution tree begins with the initial state of each state machine: *Init, Init, Init,* the 5-tuple is equal to *[0,$,$,$,stop]* where *$* represents a non-affected variable and the path condition (PC) is equal to TRUE. This $ value identifies variables that are used without being initialized before.

The first fireable transition is from *Init* to *Wait* of the lift manager state machine. This transition waits for an external message (*departure*) from the engine that initializes the elevator and the initial stage. Then events are sent to initialize the stage recorder (*StageRecord_1->init_stage()*) and the engine manager (*EngineManager_1->init_engine()*). The 5-tuple is equal to *[0,departure_1,departure_1,departure_1,stop]* where *departure_1* represents the value received by the message and the PC remains TRUE.

At each step, the tool computes all the fireable transitions and, for each case, the 5-tuple and the PC.

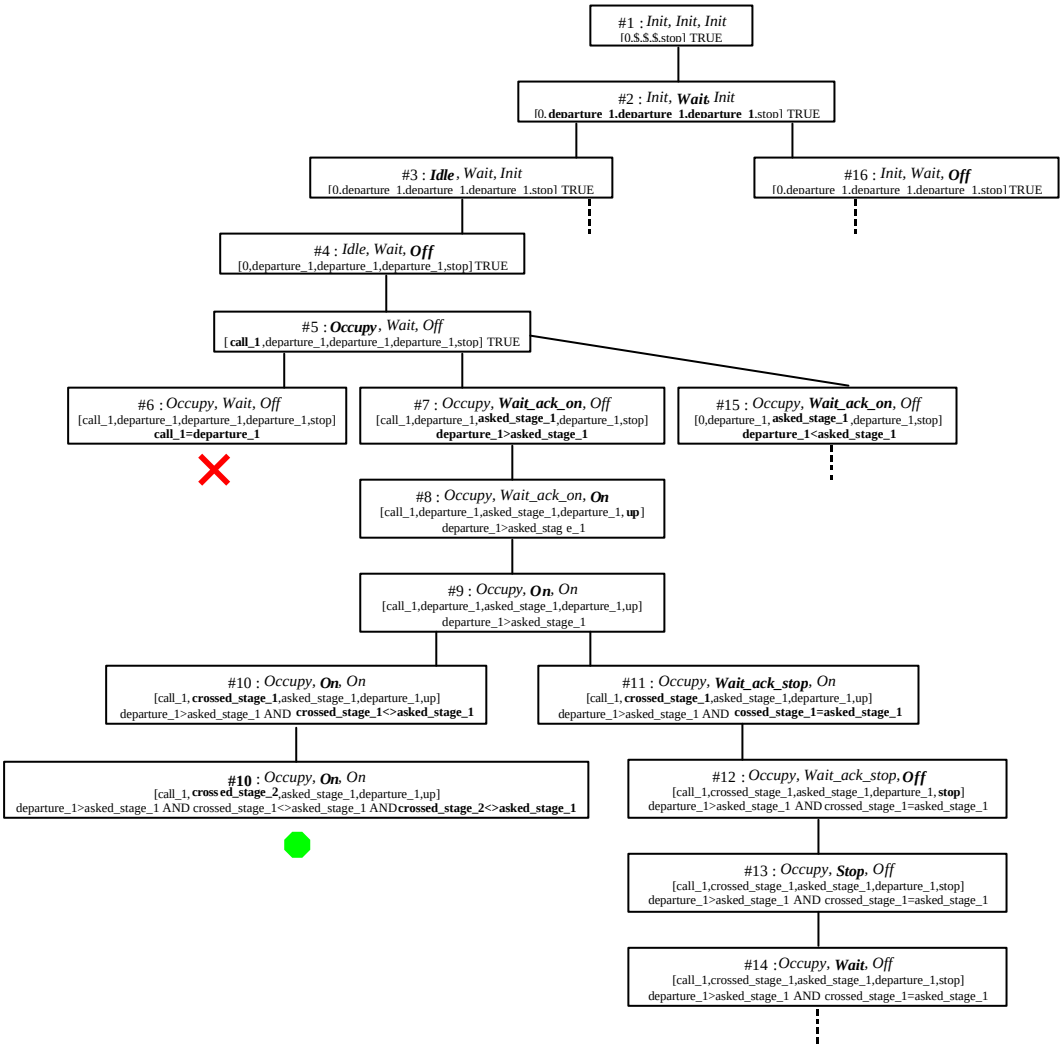


Fig.14 – Symbolic execution tree

For each computation, the tool compares the new symbolic state with the symbolic states already computed. If the control nodes are the same, domains of variables are compared. If there exists a numeric 5-tuple that verifies the constraints of the new symbolic state but not the constraints of the old symbolic state, then computing continues else it stops. For example, in symbolic state #9 *Occupy, On, On*, the 5-tuple is equal to *[call_1,departure_1,asked_stage_1,departure_1,up]* with the PC equal to

departure_1>asked_stage_1. If the tool selects the transition from *On* to *On* of the lift manager, then the new symbolic state corresponds to #10 *Occupy, On, On,* the 5-tuple is equal to *[call_1,crossed_stage_1,asked_stage_1,departure_1,up]* with the PC equal to *departure_1>asked_stage_1 AND crossed_stage_1<>asked_stage_1*. The control nodes are identical but the 5-tuple [2,1,2,0,up] verifies *[call_1,crossed_stage_1,asked_stage_1,departure_1,up]* but not *[call_1,departure_1,asked_stage_1,departure_1,up]* because current_stage=1 is different from initial_stage=0. The tool continues execution and fires the same transition. The symbolic state corresponds to #12 *Occupy, On, On,* the 5-tuple is equal to *[call_1,crossed_stage_2,asked_stage_1,departure_1,up]* and the PC is equal to *departure_1>asked_stage_1 AND crossed_stage_1<>asked_stage_1 AND crossed_stage_2<>asked_stage_1*. That time domains of variables are included and all solutions that verify the first 5-tuple verify the second. The execution stops and the symbolic state is mapped to state #10.

Let us focus on state #5. The state corresponds to *Occupy, Wait, Off,* the 5-tuple is equal to *[call_1,departure_1,departure_1,departure_1,stop]* with the PC equal to TRUE. The tool can fire the transition of the lift manager from *Wait* to *Wait,* the new state is *Occupy, On, On*, the 5-tuple remains the same the PC equal to *call_1=departure_1*. If we look at the state machines ,we can see that there is no more fireable transition. In fact, the stage recorder, in state Occupy, waits for the *reached_stage* message; the lift manager, in state Wait, waits for the *asked_stage* message; and the engine manager, in state Off, waits for the *movement_order* message. None of these messages can be sent and the system is blocked. The tool detects a deadlock.

Also note that the path condition of state #8 has been simplified. In fact the value of the PC is *departure_1>asked_stage_1 AND up<>stop*, but by definition up<>stop. Thanks to rewriting rules the path condition is simplified.

On the symbolic execution tree AGATHA can also detect some dead code like the loop on the state *On* of the engine manager. That transition is never used in each path of the symbolic tree and so this transition is unreachable.

4.2.3 Results

The tool provides the symbolic execution tree. Each path of the tree corresponds to a symbolic test case and each symbolic test case is translated into an UML sequence diagram.

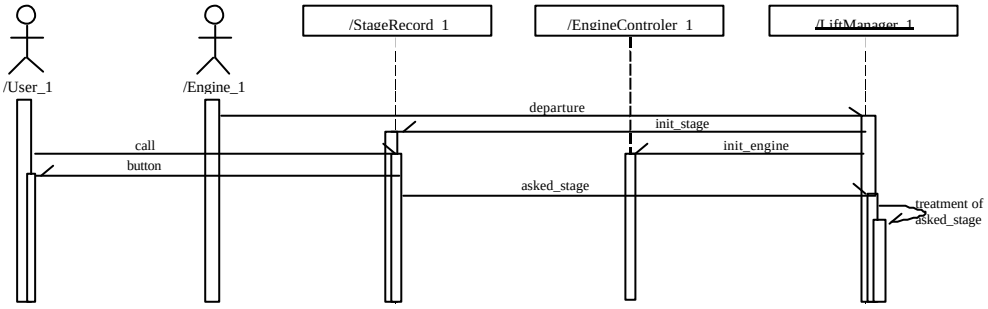


Fig.15 – Sequence diagram for the first path

On these sequence diagrams, the messages exchanged by the system appear. For our example, AGATHA computes a symbolic execution with twelve paths. For the first path, the first symbolic state corresponds to *Init, Init, Init* and the second corresponds to *Init, Wait, Init*. The lift manager received the *departure* message and then sent the *init_stage* message the stage recorder and the *init_engine* message to the engine recorder. The third symbolic state corresponds to *Idle, Wait, Off*. The stage recorder received the *init_stage* message. The fourth symbolic state corresponds *to Occupy, Wait, Off*. The stage recorder received the *call* message and sent the *button* message to the user. And so on until the path ends (see Fig.15).

In this example, we have just presented a symbolic test case. For each symbolic path the AGATHA toolset generates a symbolic sequence diagram. On each sequence diagram you can generate at least one instantiation of the symbolic test case and then obtain numeric variables for parameters of call events. For the first path of our example, the 5-tuple is equal to

[call_1,departure_1,departure_1,departure_1,stop] with the PC equal to *call_1=departure_1*. A numeric solution can be [1,1,1,1,stop].

We obtain these numeric sequence diagrams by using a constraints solver. Note that these tests can be useful for the future implementation.

4.3 Industrial example

Our team also participates in a European project, AIT-WOODDES. The main goal of this project is to deliver an environment for the design of embedded systems. In that context we work with the industrial PSA on an embedded navigation system for cars and we automatically generate a set of tests for this specification with our toolset.

5. Conclusions and perspectives

In this article we have described our toolset associated with the semantics of UML statecharts, allowing software developers to validate UML specifications. We have presented our tool, based on the AGATHA system, which is transparent for the user and definitely user-oriented. Indeed the user drives all of the validation process.

Furthermore the generated tests produce an exhaustive path coverage by using a melting-pot of formal techniques. The toolset also detects several types of deadlocks, livelocks and conception errors; it can create instantiated tests with the help of a constraints solver, not only on simple specifications but also on specifications of real industrial concurrent embedded systems.

Our tool is used as part of the AIT-WOODDES European project that aims at developing a full software workshop based on UML and targeting automotive embedded systems. The AGATHA system is also involved in projects with SDL specifications for aerospace applications with EADS. A version for statecharts [22] of STATEMATE is currently developed for PSA for embedded car system specifications [23]. The AGATHA system was also used with ESTELLE industrial specifications for EDF [24].

Our tool, in particular the UML translator, should be enhanced with all the power of the UML standard such as the notion of hierarchy in statechart diagrams. Usually to specify a system with UML a developer starts with the definition of some sequence diagrams. We can add a functionality that allows testing whether these sequence diagrams are compatible with the set of sequence diagrams computed by AGATHA.

Other applications are foreseen: enriching AGATHA with theorem proving –this should be made with backward symbolic execution–- in order to prove properties about the system. We could also imagine connecting an existing model checker to AGATHA. For very large or complex systems AGATHA will also embed new automatic simplification procedures, not working on generated expressions, but on the model itself, and based on abstraction principles. Finally, the possibly numerous generated numerical tests may exceed the capacity of an industrialist in terms of cost and time. With respect to criteria defined by the user, a selection of relevant tests will be performed, along with an estimate of their covering.

6. Acknowledgements

The authors would like to thank the FMICS 2002 reviewers, Sébastien Gérard and the whole ACCORD team, Pantxoa Amorena, François Terrier, Alain Faivre, Jean-Pierre Gallois and all of the AGATHA team for their help and their constructive comments and suggestions.

This work is supported by the European committee for the AIT-WOODDES IST project.

7. References

[1] Rumbaugh, I. Jacobson, G. Booch, The Unified Modelling Language Reference Manual, Reading, MA: Addison-Wesley, 1998.

[2] AIT-WOODDES Project N IST-1999-10069, http://wooddes.intranet.gr/.

[3] U. Buy, A. Orso, M. Pezzè: Automated Testing of Classes, ISSTA'00.

[4] J. Hartmann, C. Imoberdorf, M. Meisinger: UML-Based Integration Testing, ISSTA'00.

[5] Objecteering Tool version 5, Softeam Paris, 2001, http://www.softeam.fr.

[6] S. Gérard, N. S. Voros, C. Koulamas, Efficient system modeling of complex real-time industry; networks using the ACCORD/UML methodology, DIPES 2000

[7] H. ISO/TC97/SC21: Estelle - A Formal Description Technique Based on an Extended State Transition Model, ISO/TC97/SC21, IS 9074, 1997.

[8] D. Lugato, N. Rapin, J.-P. Gallois, Verification and tests generation for SDL industrial specifications with the AGATHA toolset, Proceedings of Workshop on Real-Time Tools, CONCUR'01.

[9] E. M. Clarke, O. Grumberg, D. A. Peled, Model Checking, The MIT press 1999.

[10] J. -C. Fernandez, C. Jard, T. Jeron, C. Viho, Using on the fly verification techniques for the generation of test suites, CAV'96.

[11] S. Yovine. Kronos: A verification tool for real time systems, Springer International Journal of Software Tools for Technology Transfer, Vol. 1, Nber 1/2, October 1997.

[12] L. A. Clarke. A system to generate test data and symbolically execute programs, IEEE Transactions on software Engineering, vol. SE-2, nº3, September 1976, pp 215-222.

[13] J.C. Huang. An approach to program testing, ACM computing surveys.7(3): 113-128, September 1975.

[14] J. C. King. Symbolic execution and program testing, Communication of the ACM,19(7). July 1976.

[15] P .Wolper, P. Godefroid. Partial-Order Methods for Temporal Verification, Université de Liège, Institut Montefiore, CONCUR 930 - Hildesheim, Belgium, August 1993.

[16] J.Chabin, J.-Y. Février, J.-P. Gallois, S. Ramangalahy, Génération de tests par exécution symbolique, Journées du GDR programmation, November 1995, Grenoble.

[17] M. Ishisone, T. Sawada, Brute: brute force rewriting engine, GAIST, January 2001, http://www.theta.theta.ro/cafeobj.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, D. Wonnacott, The Omega Library version 1.1.0, University of Maryland, November 1996, http://www.cs.umd.edu/projects/omega.

[19] R. Milner. Communication and concurrency, Prentice Hall International, 1989.

[20] M. Worner, M. Frohlich, DaVinci Tool version 2.1, Bremen University, July 98, http://www.informatik.uni-bremen.de/davinci.

[21] J-P. Rellier, F. Vardon, CON'FLEX version 1.2, Manuel de l'utilisateur, INRA, January 98, http://www-bia.inra.fr/ .

[22] D. Harel, Statecharts: a Visual Formalism for Complex Systems, Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[23] J.-Y. Pierron, J.-P. Gallois, E. Fievet, A. Lapitre, D. Lugato, Validation de systèmes industriels par le test symbolique sur spécification STATEMATE. ICSSEA'00.

[24] J.-P. Gallois, A. Lapitre, P. Lé, Analyse de spécifications industrielles et génération automatique de tests. ICSEA'99.

Heuristic-driven Techniques for Test Case Selection

J.C. Burguillo, M. Llamas, M.J. Fernández [1]

Departamento de Ingeniería Telemática
Universidad de Vigo
Vigo, Spain

T. Robles [2]

Departamento de Ingeniería Telemática
Universidad Politécnica de Madrid
Madrid, Spain

Abstract

We propose an approach to testing that combines formal methods with practical criteria, close to the testing engineer's experience. It can be seen as a framework to evaluate and select test suites using formal methods, assisted by informal heuristics. We also introduce the formalism of enriched transition systems to store information obtained during the testing phase, and to adapt classical test generation techniques to take advantage of the possibilities of the new formalism.

1 Introduction

In the context of Protocol Engineering, test generation algorithms are used to obtain a set of test cases from a given specification, intended to detect errors in non-conforming implementations. However, the number of test cases needed to guarantee an exhaustive coverage may be too large, even infinite. Therefore, execution of all potential test cases may be infeasible. As a consequence, in practical cases it is necessary to select a subset of all possible test cases prior the test execution phase. The reduction of the initially generated test case set is known in the literature as *test case selection*.

Test case selection should not be performed at random. An appropriate strategy should be applied to obtain a valuable test case collection, in the sense

[1] Email: [jrial,martin,manolo]@det.uvigo.es
[2] Email: robles@dit.upm.es

that it should detect as many non-conforming implementations as possible. For software testing, some criteria are available, like the division in equivalence partitions [12] or the test proposal selection in protocol testing [9].

On the other side, test case selection should not be based only on the system's formal specification. To select the most valuable test cases, additional information, external to the corresponding specification formalism, should also be used. Such information may consider most frequent errors committed by implementors, most harmful errors, most difficult to implement features, critical basic functionalities, etc.

In the field of Formal Description Techniques some proposals have been made to address the test case selection problem, key results may be found in [6,15,16,17]. T. Robles [13] introduced concepts for risk, cost and efficiency for a test case collection, which are revisited in this paper. This approach is based on the estimation, from the testing engineer's experience, of the risk involved when testing a system implementation. It formalises and simplifies the selection of test cases, and can be applied to most practical problems. This approach is similar to that presented in [15].

Thus, this paper proposes a method to evaluate and select test cases from practical criteria, close to the testing engineer's experience. Our aim is to provide implementable, and computationally feasible criteria. Additionally, we want the proposed methodology to be easily configurable for testing engineers, who can provide their experience through the introduction of heuristics to facilitate the testing of key aspects in a system, or specific parts of a system that are more prone to errors.

The next two sections discuss the theoretical background that serves as the foundation of our experience. Section 2 presents some general definitions and notation about the supporting representation framework and formal testing, and Section 3 presents our approach to test case selection. Finally, Section 4 offers a summary of the work described and some conclusions.

2 General Definitions and Notation

Along the next paragraphs we discuss basic theoretical concepts and notation related to testing and test case selection. First, we briefly introduce Labelled Transition Systems. Then, we provide some basic concepts from formal testing. After this, we introduce risk, coverage, cost and efficiency as the supporting heuristics to assist the testing engineer along test case selection.

2.1 Labelled Transition Systems

Labelled Transition Systems (LTS) will be the basic model to describe the behaviour of processes, such as specifications, implementations and tests.

Definition 1 *A labelled transition system is a 4-tuple* $< Stat, L, T, s_0 >$ *where Stat is a countable, non-empty set of states; L is a countable set of*

labels; $T \subseteq Stat \times (L \cup \{i\}) \times Stat$ *is the countable set of transitions and* i *denotes a special internal action, referred as* τ *in some models [11]; and* $s_0 \in Stat$ *is the initial state.*

An element $(s, \mu, s') \in T$ *is represented as* $s - \mu \rightarrow s'$*. We use the following notations (sets) derived (constructed) from the transition relation:*

$s = \epsilon \Rightarrow s'$: $s = s'$ *or* $s - i - \ldots \rightarrow s'$

$s = a \Rightarrow s'$: $\exists s_1, s_2 \in Stat$ *such that* $s = \epsilon \Rightarrow s_1 - a \rightarrow s_2 = \epsilon \Rightarrow s'$

$s = \sigma \Rightarrow s'$: $\exists \{s_1, ..., s_{n-1}\} \subseteq Stat$*, and a trace* $\sigma = a_1 \ldots a_n$ *such that* $s = a_1 \Rightarrow s_1 = \cdots \Rightarrow s_{n-1} = a_n \Rightarrow s'$.

$s = \sigma \Rightarrow$: $\exists s' \in Stat$ *such that* $s = \sigma \Rightarrow s'$

$s \neq \sigma \Rightarrow$: $\nexists s' \in Stat$ *such that* $s = \sigma \Rightarrow s'$

$\mathbf{Tr}(P)$: $\{\sigma \in L^* \mid P = \sigma \Rightarrow\}$

$\mathbf{Init}(P)$: $\{a \in L \mid P = a \Rightarrow\}$

P **after** σ : $\{s' \mid P = \sigma \Rightarrow s'\}$

$\mathbf{Ref}(P, \sigma)$: $\{A \subseteq L \mid \exists s' \in (P \textbf{ after } \sigma)$ *and* $\forall a \in A, s' \neq a \Rightarrow\}$

$\mathbf{Path}(P)$: $\{\varphi \in T^* \mid P - \varphi \rightarrow s', s' \in Stat\}$

The symbol L^* (respectively T^*) denotes the set of strings (sequences, traces) constructed using elements from L (respectively T). A trace $\sigma \in L^*$ is a finite sequence of observable actions over L, where ϵ denotes the empty sequence. The special label $i \notin L$ represents an unobservable, internal action, used to model non-determinism. Thus $= \epsilon \Rightarrow$ represents a null transition or a sequence of transitions including only internal actions (i.e. traces do not have internal actions). We use $t \lll \varphi$ to denote that transition t appears in the path φ.

We represent an LTS by a tree or a graph, where nodes represent states and edges represent transitions. Given an LTS $P =< Stat, L, T, s_0 >$, we write $P = \sigma \Rightarrow$ to represent transitions from the initial state of P and must be considered as a syntax sugar. When a given state does not accept further actions (i.e. deadlock state), we label it as **stop**.

$\mathbf{Tr}(P)$ is the set of traces accepted by process P, $\mathbf{Init}(P)$ the set of labels from L accepted by P, and $\mathbf{Ref}(P, \sigma)$ is the set of refusals of P after trace σ. Finally, $\mathbf{Path}(P)$ is the set of transition sequences accepted by P. We denote the class of all labelled transition systems over L by $LTS(L)$. LTS model the semantics of languages used to describe distributed and concurrent systems like LOTOS [8], CSP [1] or CCS [11], among others.

2.2 Formal Testing Concepts

Concerning testing, it is important to define a relation to model the conformance of a implementation with its specification. There are several relations in the literature that may be selected [14]. As we want to compare our framework with other approaches and reuse the existing theory, we selected the conformance relation **conf** described in [2,14]. It has the advantage that only the behaviour contained in the specification must be tested, reducing the test space. The relation **conf** is defined as follows:

Definition 2 (Conformance: conf) *Let* $I, S \in LTS(L)$*, we say that* I **conf** S *if and only if for every trace* $\sigma \in \mathbf{Tr}(S)$ *and for every subset* $A \subseteq L$ *the following proposition holds: If* $A \in \mathbf{Ref}(I, \sigma)$ *then* $A \in \mathbf{Ref}(S, \sigma)$
In case $\sigma \notin \mathbf{Tr}(I)$ *we assume* $\mathbf{Ref}(I, \sigma)$ *is empty.*

To decide about the success of a test case we use *verdicts*. Reference [10] proposes three possible verdicts: **Pass** (**pass**, when the observed behaviour satisfies the test), **Fail** (**fail**, when the observed behaviour is an invalid specification behaviour) and **Inconclusive** (**inc**, the observed behaviour is valid so far, but it has not been possible to complete the test). These concepts are formalised below [14]:

Definition 3 (Test case) *A test case tc is a 5-tuple* $< Stat, L, T, v, s_0 >$*, such that* $< Stat, L, T, s_0 >$ *is a deterministic transition system with finite behaviour, and* $v : Stat \rightarrow \{\mathbf{fail}, \mathbf{pass}, \mathbf{inc}\}$ *is a function to assign verdicts.*

Definition 4 (Test suite) *A test suite or test collection ts is a set of test cases:* $ts \in PowerSet(LTS_t(L))$

The execution of a test case is modelled by the parallel synchronous execution of the test case with the implementation under test (IUT). Such execution continues until there are no more interactions, i.e. a deadlock is reached. Such deadlock may appear because the test case *tc* reaches a final state, or when the combination of *tc* and the IUT reaches a state where the actions offered by *tc* are not accepted.

An implementation passes the execution of a test case if and only if the verdict of the test case is **pass** when reaching a deadlock. As the implementation may have nondeterministic behaviour, different executions of the same test case with the same IUT may reach different final states, and as a consequence different verdicts. An implementation passes a test case *tc* if and only if all executions of *tc* produce a **pass** verdict. This means that we should execute every test case several times to obtain a final verdict, ideally an infinite number of times.

Test generation algorithms provide test suites from specifications. Ideally, an implementation must pass a test suite if and only if it conforms. Unfortunately, in practice, such test suite would have infinitely many test cases. As a consequence, in the real world we have to restrict ourselves to (finite-size) test suites that can only detect non-conformance, but cannot detect conformance.

Table 1
Error Weighting

Target	Parameter	Range
Event	$R_I(e) = E_I(e) \times I_I(e)$	$(0, \infty)$
Implementation	$R_I(S)$	$(0, \infty)$
Measurement, Event	$MR_I(e, ts)$	$[0, \infty)$
Measurement, Implementation	$MR_I(S, ts)$	$[0, \infty)$

Legend. I: implementation under test; e: event in I; ts: test suite; S: specification corresponding to I.

Such test suites are called *sound*.

2.3 Risk, Coverage, Cost and Efficiency

Through the next few paragraphs we introduce the concepts of *error weight* or *risk*, *coverage*, *cost* and *efficiency*, which will support the comparison and selection of test cases to be passed to an implementation.

To analyse the coverage obtained after testing an implementation we have to take into account several factors. On one side, test cases are derived from a formal object, i. e. the formal specification. As a consequence, after testing an implementation we get a specific coverage level for the behaviours in the specification. On the other side, coverage depends on the implementation itself because, given a formal specification, the selected implementation technology (i.e. programming language or programming tools) will be more or less prone to errors.

Table 1 proposes some heuristics to *a priori* evaluate the influence of errors in a given implementation, which will be used to select an adequate test suite. $R_I(e)$ assigns a weight to a (possible) error, i.e. estimates the risk involved in committing errors when implementing event e. It is calculated from two values: an estimation of the chances of event e being erroneously implemented ($E_I(e)$), and an estimation of the impact of the corresponding error in the rest of the system ($I_I(e)$). $R_I(S)$ estimates the chances for the implementation not to conform to the corresponding specification, and measures the risk of erroneously implementing S.

$MR_I(e, ts)$ represents the amount of risk for event e that can be detected through a testing process using test suite ts, and $MR_I(S, ts)$ represents the amount of risk for implementation I that can be detected using test suite ts. Risk measurement for a single test case is a particular case where suite ts is composed by a single test case. Note that, from the definitions above, $MR_I(e, ts) \leq R_I(e)$ and $MR_I(S, ts) \leq R_I(S)$.

The underlying mathematical model we need is considerably simplified through the assumption of independence among errors. However, in prac-

Table 2
Coverage Parameters

Target	Parameter	Range
Event	$K_I(e,ts) = \frac{MR_I(e,ts)}{R_I(e)}$	$[0,1]$
Implementation	$K_I(S,ts) = \frac{MR_I(S,ts)}{R_I(S)}$	$[0,1]$

Table 3
Cost Parameters

Target	Parameter	Range
Event	$C_I(e) = P_I(e) + X_I(e)$	$(0,\infty)$
Implementation	$C_I(S,ts)$	$(0,\infty)$

tice, errors are not independent from each other, as erroneous sentences in a program may affect the evolution of other parts of the program. As a solution, correlation among errors is reflected in our model as error weight values, that is, we model such interdependence through parameter $I_I(e)$. Then, testing engineers will estimate the correlation among errors, using available error statistics and their own expertise, to define $I_I(e)$ accordingly.

This can be seen as a compromise between a convenient mathematical foundation and the need to consider error correlation in real cases. Note that, independently of being supported by the underlying mathematical model or through explicit parameters, getting the correlations between failures right is crucial to get the most of the approach discussed in this paper.

From the parameters above, we define *coverage* as the quotient between a measurement of the detection power of a test suite and a measurement of the risk (c.f. table 2). $K_I(e,ts)$ represents the coverage for event e using test suite ts, whereas $K_I(S,ts)$ represents the coverage for implementation I, corresponding to specification S, using test suite ts.

When executing a test suite ts on an IUT we are checking whether some of the error possibilities estimated have been materialised into actual errors. If errors appear, they should be corrected. Conversely, if errors are not found, our confidence increases. Given two test suites ts_1 and ts_2, using the parameters above we can compare their coverage, and therefore their ability to detect errors in an IUT. However, there is another factor when comparing test suites that should be taken into account: the resources needed. To estimate this aspect, we introduce a new parameter: the *cost* (c.f. table 3). $C_I(e)$ estimates the cost of testing event e as the sum of the cost due to its implementation in a test case ($P_I(e)$) and the cost of executing that event on the implementation ($X_I(e)$). $C_I(S,ts)$ represents the cost of testing an implementation I using test suite ts generated from specification S.

Using cost values we can better discriminate among several test suites. Therefore, the next step will be to relate the parameters defined above to

obtain another reference to facilitate the selection of test cases. For this, we define the *efficiency* of a test suite ts obtained from S ($F_I(S, ts)$) as the quotient between the coverage of that suite and the cost associated to its use to test I.

$$F_I(S, ts) = \frac{K_I(S, ts)}{C_I(S, ts)}$$

The values of this new parameter are in the range $[0, \infty)$. Its values increase when coverage increases and with cost reduction.

We need a procedure to calculate values for the heuristics above taking into account our representation formalism, namely Labelled Transition Systems. We try to assess conformance for a system implementation from its formal specification. Thus, we will take as a reference the risk involved when implementing all events in the specification. In this way, we can formulate the risk for a IUT as the sum of the risk values for its events.

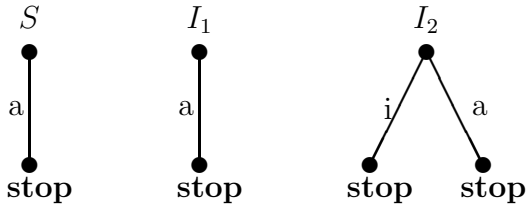


Fig. 1. S, I_1 and I_2

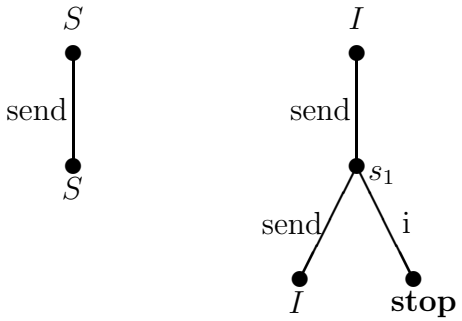


Fig. 2. S and I are recursive processes

On the other side, due to nondeterminism, practical test cases should be executed several times to gain confidence on the testing process. For example, consider the specification S in figure 1 and its implementations I_1 and I_2. While the implementation I_1 is equal to S and will always accept event a as stated by S, implementation I_2 sometimes executes an internal action and then refuses event a. Obviously, this latter implementation does not conform with S.

If we are testing a physical implementation, which may behave as I_1 or I_2, we will need to execute several times a from the initial state in order to discover if it conforms with S. Each time event a is accepted we increase our confidence on the implementation. Conversely, if we obtain a single refusal we

can guarantee that the IUT does not conform. In other words, measurement risk values vary along the testing process.

Additionally, the presence of recursive behaviours makes testing dependent on the level of recursion where the test is passed. We name recursive behaviours those ones that are self-instantiated. Consequently, the recursion level will be the number of times a behaviour has been instantiated. For instance, specification S in Figure 2 contains a recursive behaviour and never stops. Again, to check a physical implementation of S that behaves as I in Figure 2, we might need to execute many times event *send* to detect that sometimes such event is refused. As a consequence, the risk measurement involved when testing an event is spread along the successive levels of recursion (i.e. successive event instantiations).

Taking into account both aspects, we can decompose the risk of every event in an LTS (i.e. the weight assigned to errors in events) as:

$$R_I(e) = \sum_{r=1}^{\infty} \sum_{n=1}^{\infty} R_I^{r,n}(e) \leq \infty$$

where $R_I^{r,n}(e)$ represents the risk of event e when being tested for the n-th time at recursion level r using a given test case. Then, the risk detection power of a test suite ts becomes:

$$MR_I(S, ts) = \sum_{tc \in ts} \sum_{e \in E(tc)} \sum_{r=1}^{Rc_e} \sum_{n=0}^{N_e(r)} R_I^{r,n}(e)$$

where Rc_e and $N_e(r)$ are respectively the deepest recursion level where event e has been tested and the number of times we tested such event for every recursion level. If test cases $tc \in ts$ have a tree structure we can obtain several possible values for every successful run of the test case. So, we may measure the risk, *a priori*, using available statistics.

2.4 A Priori and a Posteriori Values

As the IUT is an entity whose behaviour is unknown, there may be differences between what we desire to test and what we really test in practice. These differences may be due to:

- **Nondeterminism**: due to nondeterministic behaviour in the implementation, it is possible that, in a first try, we cannot test those behaviours we are interested in. Because of this, it may be needed to execute test cases several times until we reach an appropriate result. New executions modify coverage values.
- **Failures**: if we detect a non-conforming implementation, it may not be possible to achieve the expected coverage because some test cases may not be executable due to errors in the implementation.

As a consequence we can identify [7] two classes of cost and coverage values:

- **A priori values**, which are obtained when we estimate the risk measurement and the cost to execute a test case *tc* assuming all possible implementation responses, as defined by the corresponding specification.
- **A posteriori values**, which are obtained after executing the test case *tc*.

3 Test Case Selection

Now, we will discuss our approach to test case selection, which is based on a classical approach, as discussed below. But first we introduce Enriched Transition Systems as a way to keep track of the structural information needed to know those parts of the specification already tested.

3.1 Enriched Transition Systems

When we try to execute several test cases over an implementation, it would be desirable to have access to the values of risk, cost and coverage obtained along the process. For this, as discussed above, we need information about recursion levels and testing iterations. Besides, if these values were available, we could select new test cases depending on the results obtained from the ones that have been already executed.

To maintain the information gathered after the execution of test cases we define a new type of transition systems [5]:

Definition 5 (Enriched Transition System) *An enriched transition system (ETS) is a 5-tuple denoted by* $S =< Stat, L, T, N(t,r), s_0 >$*, such that* $< Stat, L, T, s_0 >$ *is a labelled transition system and* $N(t,r)$ *is the number of times transition* $t \in T$ *is executed at recursion level* $r \in [1, \infty)$*.*

The set of enriched transitions systems over the label set L is denoted by $ETS(L)$. Available notation and definitions for $LTS(L)$ are extended to $ETS(L)$ defining them over the underlying transition system. Unlike classical LTS, ETS are dynamic, i.e. for every transition $t \in T$, function $N(t,r)$ changes its values along the test process.

When we execute a test case on an implementation I generated from a specification S, events in the enriched specification $S_E \in ETS(L)$ are updated with the number of executions in every recursion level. In this way, we maintain information concerning which behaviours or specification parts have not been sufficiently tested. Note that from the specifications described as ETS we can easily obtain risk and coverage values.

We assume that every transition has its own risk value. We also assume the existence of an heuristic function for measuring risks $f_{MR}(e,r,n) \rightarrow [0, R_I(e)]$ provided by the test engineer. This function will provide the risk measured for individual executions in a given level of recursion. This function must be convergent, and the sum over r and n of all risk measurements for a single event e must be less than or equal to the risk of that event.

Example 1 *A suitable risk measurement function can be defined as*

$$MR_I^{r,n}(e) = \frac{R_I(e)}{2^{r+n}} \quad for\ r, n \geq 1$$

Up to now, we have been considering transition systems without any additional information about which parts may be recursively called, which parts correspond to the main process, etc. In other words, when we traverse a plain LTS we do not know which states are recursively accessed from other states. With ETS, we consider every transition as a potential process (i.e. as a potential destination for a recursive call). Every time we reach a previously visited state, we assume that we have increased by one the recursive level for the next transition. In this way, we just need to check how many times we have visited a state to obtain the level of recursion.

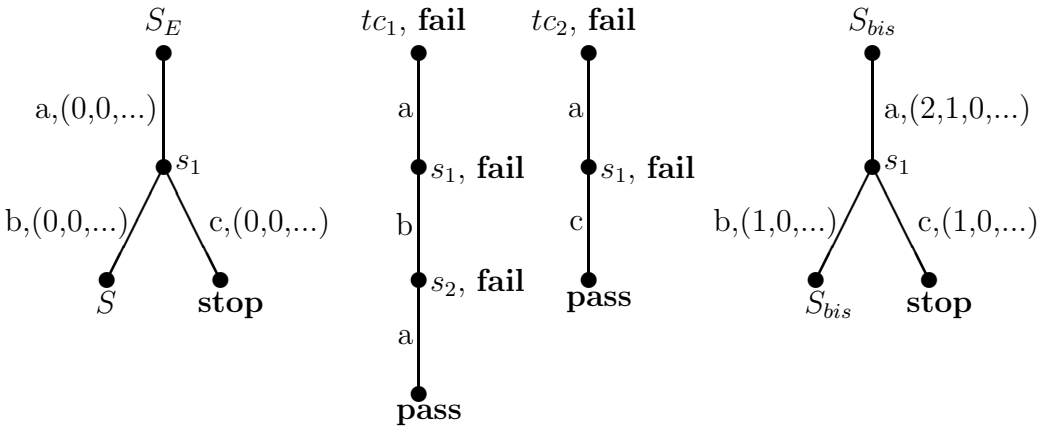


Fig. 3. S, tc_1, tc_2 and S_{bis}

Example 2 *Suppose that we have the recursive specification $S_E \in ETS(L)$ appearing in Figure 3. Function $N(t, r)$ appears next to the corresponding label for every transition. We have represented the function $N(t, r)$ as a sequence where the first element is the number of times we executed the transition in the first recursion level, the second element corresponds to the second level of recursion and so on. Initially, all values in the sequence are zero because we did not execute any test yet. Suppose also that we have a physical object I that implements correctly the behaviour described in the specification, i.e. $I = S_E$, and that we want to execute test cases tc_1 and tc_2 described in Figure 3.*

S_{bis} represents a snapshot of $S_E \in ETS(L)$ after the execution of both test cases. Event a has been tested twice in the first level of recursion, one for each test case. Besides, this event has also been tested in the second level of recursion, which corresponds to the last transition of tc_1. The rest of the events have been executed only once in the initial recursion level.

3.2 Algorithms for Risk-driven Test Case Selection

For test generation and selection, we firstly adopted a classical testing algorithm and modified it to take into account risk and coverage values. The classical approach selected was Tretmans' [14], which constructs tree-like deterministic test cases recursively selecting at random a subset of all possible specification transitions from a given state.

Table 4
Generating test cases for S

Given $S \in ETS(L)$, we construct a test case $tc := \sum\{a; tc_a \mid a \in A_{MR}\}$ recursively as follows:

(i) Construct the set $C_S := \{\mathbf{Init}(S') \mid S = \epsilon \Rightarrow S'\}$

(ii) Among all possible sets $A \subseteq \mathbf{Init}(S)$, select the set A_{MR} having a maximum value of $\frac{\sum_{e \in A} MR_I^{r,n}(e)}{Card(A)}$ and satisfying one of the following:
  (a) $\forall C \in C_S : A_{MR} \bigcap C \neq \emptyset$ and $v(tc) = \mathbf{fail}$, or
  (b) $\emptyset \in C_S$ and $A_{MR} = \mathbf{Init}(S)$ and $v(tc) = \mathbf{pass}$, or
  (c) $A_{MR} = \emptyset$ and $v(tc) = \mathbf{pass}$

(iii) Construct recursively tc_a as a test case for $\sum\{i; S' \mid S = a \Rightarrow S'\}$

(*) When representing a test case, $\sum$ represents branching and $a; s$ is short notation for transitions (i.e. $-a \rightarrow s$).

In our case (c.f. Table 4). We modified Tretmans algorithm to select (considering the conditions expressed in [14]) the set $A_{MR} \subseteq \mathbf{Init}(s)$ that maximizes the mean risk measurement.

Concerning the test generation process and the ETS formalism, before we generate any test case, we make a copy of $S_E \in ETS(L)$ and name it S_E^{bkp}. During the generation process we will work with S_E^{bkp} instead of S_E. Then, each time a new set A_{MR} is selected, the values of $N(t, r)$ in copy S_E^{bkp} are updated accordingly as they are executed. For example, if due to recursion the same transition is selected for a second time in the being generated test case, the corresponding value for $N(t, r)$ will reflect that now we are in the second level of recursion. These values are updated in S_E^{bkp} and are considered *a priori* values (c.f section 2.4). In other words, *a priori* values are updated along the generation of a test case over the copy, and they guide the construction of the test case in a dynamic fashion.

Once a test case has been completely generated, we recover the original ETS specification, formerly S_E, and execute the test case. After the execution of the test case, values of $N(t, r)$ in S_E are updated according to the execution sequence obtained *a posteriori*.

This cycle (i.e. test generation using *a priori* values, test execution to obtain *a posteriori* values, which are used as the initial values for the next iteration) is repeated until test cases with the desired coverage or cost are

obtained. This way, we construct dynamically test cases to cover those parts less adequately tested so far. This approach has been illustrated recently with a case study [4] and described extensively in [5].

Nevertheless, the algorithm in table 4 has two drawbacks:

(i) **Unnecessary cost increments**: the algorithm generates test cases with a tree structure introducing additional branches to cover non deterministic behaviours. When executing such test cases, they might examine certain parts of the implementation already tested, while others might not be covered enough, originating extra cost and decreasing effectiveness.

(ii) **Partial selection versus global selection**: the selection of A_{MR}, along the test case generation, has not considered any prediction level. This means that there could be cases where the chosen transitions have not been previously tested, but which drive to behaviours with a reduced impact over the global risk.

Table 5
Generating test cases using prediction.

Given $S \in ETS(L)$, i_p, l_{max} and $s_x = s_0$. A test case tc of S is:

$$tc := \{a; tc_a \mid \mathbf{PathTr}(\varphi_{opt}) = a.\sigma'\}$$

with $\varphi_{opt} \in \Gamma : \Gamma = \{\varphi \in \mathbf{Path}(s_x) : |\varphi| \leq i_p\}$ that satisfy:

1. $MR_I(S, \varphi_{opt}) \geq MR_I(S, \varphi), \forall \varphi \in \Gamma$.
2. $|tc| \leq l_{max}$
3. Using $\mathbf{PathTr}(tc) = \sigma.a$ we assign verdicts with:
   a) if $L \in \mathbf{Ref}(S, \sigma)$ then $v(tc) = \mathbf{pass}$;
   b) if $\{a\} \in \mathbf{Ref}(S, \sigma)$ then $v(tc) = \mathbf{inc}$;
   c) if $\{a\} \notin \mathbf{Ref}(S, \sigma)$ then $v(tc) = \mathbf{fail}$;

being $MR_I(S, \varphi) = MR_{ini}(S, \varphi) + \frac{MR_{end}(S,\varphi)}{1+\alpha.N_{inc}}$ and divided in $\varphi = \varphi_{ini}.\varphi_{inc}$ where φ_{ini} is the initial subpath φ without **inc** verdicts and φ_{end} is the ending subpath from the first **inc** verdict. $\alpha \in [0, 1]$ is a parameter we may select and N_{inc} is the number of verdicts **inc** that have appeared. We calculate:

$$MR_{ini}(S, \varphi) = \sum_{t \lll \varphi_{ini}} R_I^{r,n}(t)$$
$$MR_{end}(S, \varphi) = \sum_{t \lll \varphi_{end}} R_I^{r,n}(t)$$

tc_a is the test case generated from the state s_y such that $s_x - a \rightarrow s_y$.

Therefore, we want to complement the possibility of generating test cases

with a tree structure, c.g., the ones appearing in table 4, with the generation of test cases oriented to check certain behaviours poorly tested so far. The algorithm presented in table 5 can be used in the later test phases when some specification parts still have a low level of coverage. In such table, function **PathTr**(φ) returns the trace $\sigma \in \mathbf{Tr}(P)$, obtained following path φ. Again, during test case generation we must use a copy (S_E^{bkp}) to modify its a priori $N(t,r)$ values. The main properties of this new algorithm are:

(i) We introduce a *prediction parameter* (i_p) and a *maximum length* (l_{max}).

(ii) From state $s' \in Stat(S)$ we evaluate the risk of all possible transition paths $\varphi \in \mathbf{Path}(s')$ such that $|\varphi| \leq i_p$, i.e., paths with less length than the prediction parameter.

(iii) We choose the path φ_{opt} that, a priori, measures more risk. Concerning risk measurement, we take care of the presence of verdicts **inc** using the parameter $\alpha \in [0,1]$. This parameter allows to penalize test cases that may end without a conclusive verdict, but generating a cost. If $\alpha = 0$ then the presence of inconclusive verdicts is not considered. If $\alpha = 1$, we reduce the *a priori risk measurement*, computing the risk contained in the nondeterministic sequence and dividing its value by$(1 + N_{inc})$. A typical initial value may be $\alpha = 0.5$.

(iv) Once φ_{opt} has been chosen, we take its first transition t and update the value of $N(t,r)$ in S to model its execution, advance to the next state and repeat step 2 until the test case tc reaches the desired length.

Changing the prediction parameter i_p we may tune the precision when generating the test case. With $i_p = 1$ we have the same information than in the algorithm presented in table 4. With $i_p = \infty$ we will choose the (*a priori*) best test case. The price we have to pay when increasing the value of i_p is the computational cost needed to evaluate all possible paths from a given state and the inherent risk measurement computations. Our experience shows that i_p values around 3 to 5 are quite feasible and specification realistic.

Example 3 *Figure 4 shows the specification S. The risk values estimated for its events are:* $R_a = 2$, $R_b = 1$, $R_c = 5$, $R_d = 3$ *y* $R_e = 1$. *Considering there is no recursion, we select the next function to measure the risk:*

$$f_{MR}(e,n) = \frac{R_I(e)}{2^n}$$

Such function satisfies:

$$R_I(e) = \sum_{n=1}^{\infty} R_I^n(e) = \sum_{n=1}^{\infty} f_{MR}(e,n) = \frac{R_I(e)}{2^n}$$

Therefore, in every execution we measure part of the risk for an event, and the global risk we may measure is equal to the risk of failure for the event.

Using the algorithm that appears in table 4 we may select for the set A one of the sets $\{a\}$, $\{d\}$ *or* $\{a,d\}$. *Their respective values for risk measurement are:*

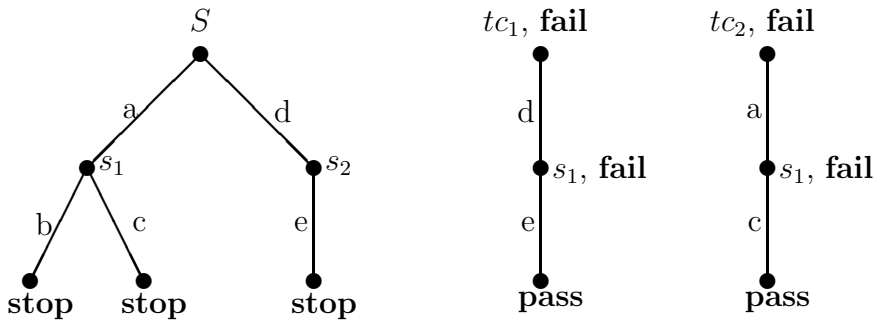


Fig. 4. S, tc_1 and tc_2

- $f_{MR}(a,1) = 2/2 = 1$
- $f_{MR}(d,1) = 3/2 = 1.5$
- $\frac{f_{MR}(a,1)+f_{MR}(d,1)}{Card(\{a,d\})} = \frac{1+1.5}{2} = 1.25$

Hence, using this algorithm, we would choose $A = \{d\}$. *Following the steps described in table 4, we obtain the test case* tc_1, *which appears in figure 4.*

On the other hand, we will use the predictive algorithm of table 5 with a prediction parameter $i_p = 2$. *In the initial state of* S_E^{bkp} *we must calculate all transition sequences of length* $i_p = 2$ *and determinate their risk measurement. There are three cases:*

(i) $a; b$: *with the risk measurement* $f_{MR}(a,1) + f_{MR}(b,1) = 2/2 + 1/2 = 1.5$

(ii) $a; c$: *with the risk measurement* $f_{MR}(a,1) + f_{MR}(c,1) = 2/2 + 5/2 = 3.5$

(iii) $d; e$: *with the risk measurement* $f_{MR}(d,1) + f_{MR}(e,1) = 3/2 + 1/2 = 2$

As the bigger measurement of risk is present in the second option, we take its first transition, modify a priori the values of $N(t,r)$ *in* S_E^{bkp} *for that transition, advance to the next state and repeat the procedure.*

After the first transition there are only two possibilities b *or* c, *both of length 1. We proceed to determinate their risk measurement, which are:* $f_{MR}(b,1) = 1/2 = 0.5$ *and* $f_{MR}(c,1) = 5/2 = 2.5$. *Therefore, we choose the transition with* b *obtaining the test case* tc_2 *in figure 4.*

The a priori global risk measurement for tc_1 *is* $MR_I(S, tc_1) = 2$ *and for* tc_2 *is* $MR_I(S, tc_2) = 3.5$. *The second test case is clearly better than the first concerning risk measurement. Thus, if the election of transitions is done with a certain level of prediction we can take advice of the information that an enriched transition system offers.*

4 Conclusions

We have presented in this paper an approach to testing supported by formal methods, which also includes non-formal heuristics to introduce the experience of the testing engineer to evaluate the costs of the testing process.

Our experience showed us that this approach, based on error weighting and cost values, provides a way to assign values to different test cases, which

permits to classify them according to different criteria, taking into account the desired coverage and supported cost. Test generation can be directed by these heuristics to obtain context-adapted test suites.

This proposal has been experimented recently with a practical case study: the testing of a protocol for mobile auctions in a distributed, wireless environment [4]. LOTOS was selected as the supporting formal language. Nevertheless, the ideas discussed here are not specific to LOTOS, but applicable to a wide range of formal techniques, with comparable expressive power.

References

[1] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. Journal of the ACM 31, 1984

[2] Brinksma, E.: A Theory for the Derivation of Tests. Protocol Specification, Testing and Verification VIII, 63-74. 1988.

[3] Brinksma, E., Tretmans J., Verhaard, L.: A Framework for Test Selection. Protocol Specification, Testing and Verification, XI. Elsevier Science Publishers B.V. 233-248, 1991.

[4] Burguillo, J.C., Fernández, M.J., González, F.J., Llamas, M. Heuristic-driven Test Case Selection from Formal Specifications: A Case Study. 11th Conference on Formal Methods Europe (FME2002). Copenhagen, Denmark, July 22-24, 2002. (paper accepted, to be published by LNCS). http://floc02.diku.dk/FME/

[5] Burguillo-Rial, J.C.: Contribución a la Fase de Prueba de Sistemas Concurrentes y Distribuidos mediante Técnicas de Descripción Formal. Ph. D. Dissertation (in Spanish), Universidad de Vigo, Spain, 2001.

[6] Heerink, L., Tretmans, J.: Formal Methods in Conformance Testing: a Probabilistic Refinement. In B. Baumgarten, H.J. Burkhardt, and A. Giessler, editors, Int. Workshop on Testing of Communicating Systems IX, Chapman & Hall, 1996, 261-276.

[7] Huecas, G.: Contribución a la Formalización de la Fase de Ejecución de Pruebas. Ph. D. Dissertation (in Spanish), Universidad Politécnica de Madrid, Spain, 1995.

[8] Information Processing Systems - Open Systems Interconnections: LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. IS 8807, ISO, 1989.

[9] Information Processing Systems - Open Systems Interconnections: Conformance Testing Methodology and Framework. IS 9646, ISO, 1991.

[10] ITU-T: Recommendation Recommendation Z.500. Framework on Formal Methods in Conformance Testing. ISO ITU-T, Mayo 1997.

[11] Milner, R.: Communication and Concurrency. Prentice-Hall International, London, 1989

[12] Myers, G.L.: The Art of Software Testing. John Wiley & Sons Inc., 1979.

[13] Robles, T.: Contribución al Tratamiento Formal de la Fase de Pruebas del Ciclo Software en Ingeniería de Protocolos. Ph. D. Dissertation (in Spanish), Universidad Politécnica de Madrid, Spain, 1991.

[14] Tretmans, J.: Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. Computer Networks and ISDN Systems, 29: 49-79, 1996.

[15] Velthuys, R.J., Schneider, J.M., Zoerntlein, G.: A Test Derivation Method Based on Exploiting Structure Information. Protocol Specification, Testing and Verification XII, 1992.

[16] Zju, J., Vuong, S.T.: Generalized Metric Based Test Selection and Coverage Measure for Communication protocols. Formal Description Techniques and Protocol Specification, Testing and Verification. FORTE X/PSTV XVII. IFIP 1997.

[17] Zju, J., Vuong, S.T., Chanson, S.T.: Evaluation of Test Coverage for Embedded System Testing. 11th International Workshop on Testing of Communicating Systems. 1998.

Scalable System-level CTI Testing through Lightweight Coarse-grained Coordination

Tiziana Margaria [1,2]

METAFrame Technologies GmbH
Dortmund, Germany

Bernhard Steffen [2]

University of Dortmund
Dortmund, Germany

Abstract

We propose a solution to the problem of system-level testing of functionally complex communication systems based on lightweight coordination. The enabling aspect is here the adoption of a coarse-grained approach to test design, which is central to the scalability of the overall testing environment. This induces an understandable modelling paradigm of system-wide test cases which is adequate for the needs and requirements of industrial test engineers. The approach is coarse-grained in the sense that it renounces a detailed model of the system functionality (which would be unfeasible in the considered industrial setting). The coordination is lightweight in the sense that it allows a programming-free definition of system-level behaviours (in this case complex test cases) based on the coarse models of the functionalities. These features enable test engineers to graphically design complex test cases, which, in addition, can even be automatically checked for their intended purposes via model checking.

1 Introduction

System-level testing of communication systems is an intrinsically business-critical issue for all the stakeholders: technology providers, service providers, and customer companies that rely on those systems as basis of their business. It is also a very complex problem, because it involves a variety of technologically heterogeneous subsystems. Although adequate test tools for the unit test

[1] eMail:TMargaria@METAFrame.de
[2] eMail:{Tiziana.Margaria, Bernhard.Steffen}@cs.uni-dortmund.de

of each subsystem are available, an integrated approach is still missing: the system-level tests must be designed and executed almost entirely manually.

We present here a coordination-based integrated test environment which realizes at Siemens the high-level coordination of complex system-level-tests for a scenario of commercial, state-of-the-art Computer-Telephony Integrated systems. Central aspect is here the adoption of a coarse-grained approach to test design, which is central to the scalability of the overall testing environment. This enables an understandable modelling of system-wide test cases, adequate for industrial test engineers, that is based on a *lightweight coordination* model. These features enable test engineers to graphically design complex test cases, which, in addition, can even be automatically checked for their intended purposes via model checking.

In the following, we introduce the concrete application scenario (Sect. 2), describe our coordination-based scalability approach (Sect. 3 to 5), which concerns test case design, test execution, and validation via coarse-grained model checking. Finally we address related work (Sect. 6) and conclude (Sect. 7) with remarks on the benefits and the current perspectives.

2 CTI System-Level Testing in Practice

The application domain concerns *Computer Telephony Integrated* (CTI) systems, i.e. composite (sensitive to platform aspects), embedded (due to hardware/software codesign practices), reactive systems that offer high availability services to a number of clients, and which therefore run on distributed architectures (e.g. client/server architectures). The supported capabilities cover at the moment the collaboration between LAN-enabled midrange telephone switches and a variety of third-party, typically client-server, applications running on PCs. Integration of WAN capabilities and mobile phones is foreseen for the next generation of switches, thus it must be conceptually captured already by today's environment. In any installed scenario, complex subsystems affect each other in a variety of different ways, so mastering today's testing scenarios for telephony systems demands an integrated, open and flexible approach to support the management of the overall test process, from the specification and design of tests to their execution and to the analysis of test results.

As a typical example of an integrated CTI platform, Fig. 1 shows a midrange telephone switch and its environment. The switch is connected to the ISDN telephone network and communicates directly via a LAN or indirectly via an application server with CTI applications that run on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they react to stimuli sent by the switch (e.g. notify incoming calls). System level test investigates the interactions between such subsystems. Typically, each participating subsystem requires an individual test

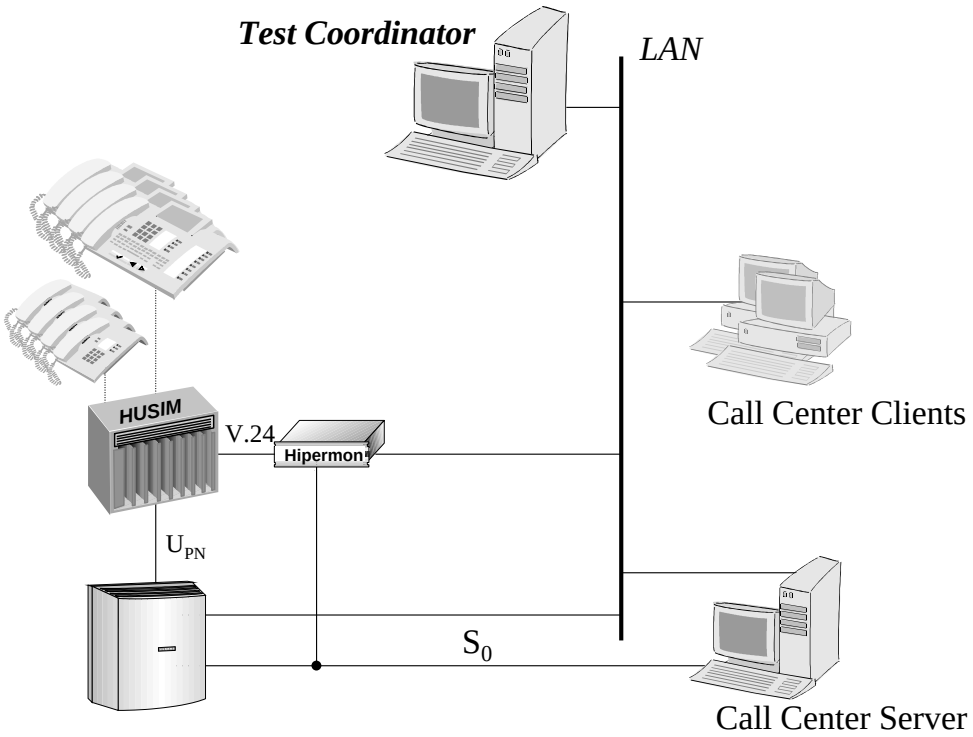


Fig. 1. Example of an Integrated CTI Platform

tool. In the scenario of Fig. 1 three different test tools are needed: *Husim*, an emulator for *UPN Devices* (i.e. telephones), *Hipermon* [6], a telephony and LAN interface tracer, and *Rational Robot* [14], a GUI capture/replay test tool for applications located on the application PCs.

Accordingly, in order to test systems composed of several independent subsystems that intercommunicate, one must be able to coordinate a heterogeneous set of test tools in a context of heterogeneous platforms. This task exceeds the capabilities of today's commercial test management tools, which typically cover only the needs of specific (homogeneous) subsystems and of their immediate periphery.

Traditional formal-methods based test automation approaches fail to enter practice in this scenario because they require a fine granular formal model of the involved systems as a basis. In reality, none of the depicted subsystems has any formal model, but all have a running reference implementation, which is itself a moving target, yet constitutes de facto the basis of all functional and regression testing activities. We thus need an approach capable of developing a formal coordination layer on top of existing black or graybox implementations which rapidly evolve.

Due to the gray/blackbox availability of the systems under test the coordination is necessarily *coarse grained*. Due to the rapid evolution of the systems (with cycles of one week to three months) the coordination must be extremely *lightweight*: there is no hope of having the resources for "reprogramming" new test cases once a subsystem varies. Adaptions and changes have to be easy

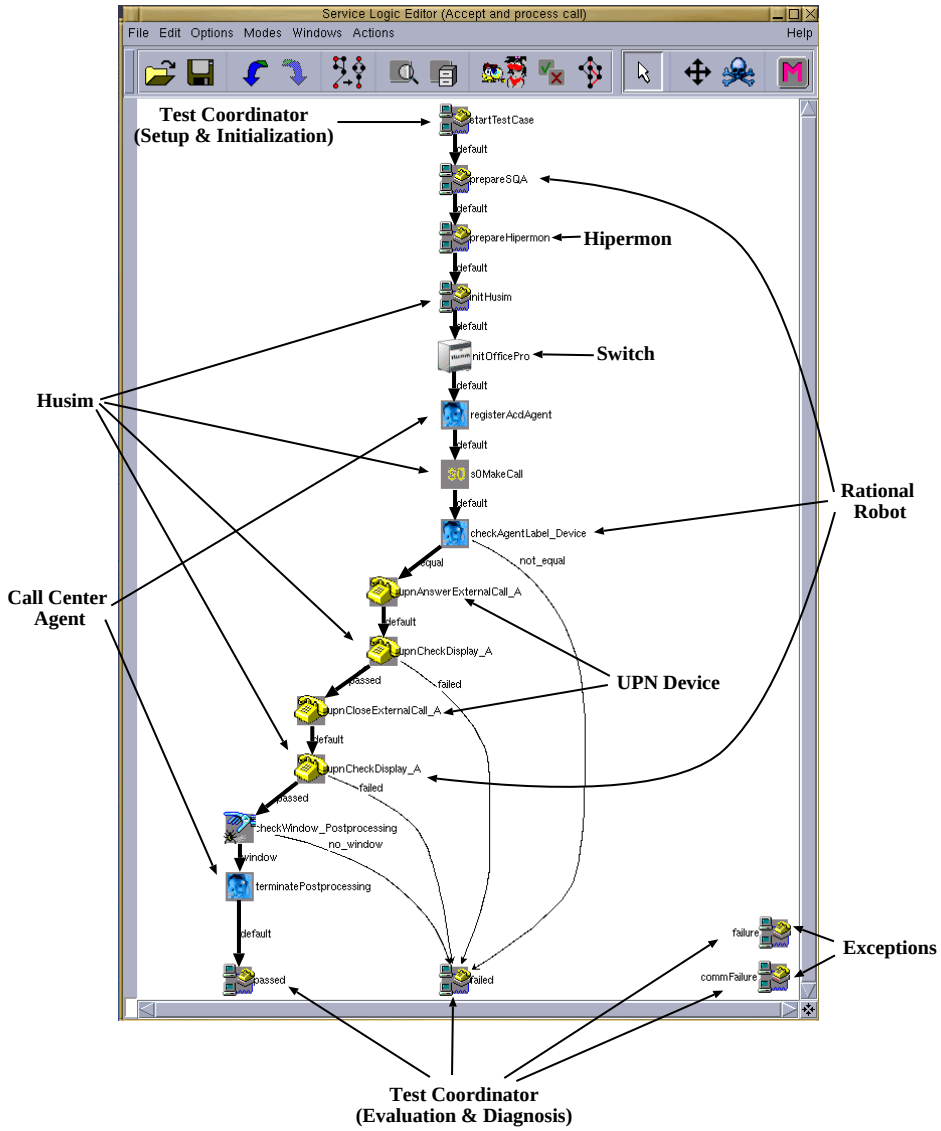


Fig. 2. Test Case in the ITE Environment

and programming-free. Taken together, this defines the 'meta-level' on which

- test engineers are used to think,
- test cases and test suites can be easily composed and maintained,
- test scenarios can be configured and initialized,
- critical test case consistency requirements (including version compatibility and frame conditions for executability) are easily *formulated*, and
- error diagnosis must occur.

2.1 Test Coordination as Superposition

The ITE (Integrated Test Environment) deployed at Siemens contains a dedicated *Test Coordinator* (TC), see Fig. 1 tool which constitutes the system

level test management, organization, and coordination layer of the ITE. The TC is an application-specific specialization for the testing domain of an existing general purpose environment for the management of complex workflows, (METAFrame Technologies' *Agent Building Center* ABC [18]). The ABC offers built-in features for the programming-free coordination and the management of libraries of functional components. This platform also forms the basis of the new release of ETI (Electronic Tool Integration platform) [17,2].

The test coordinator is responsible for the definition and enforcement of complex behaviours which are *superposed* on the system under test. The challenge is precisely how to handle this superposition in an independent, understandable, and manageable way:

- it should be *expressive* enough to capture the coordination tasks, like steering test tools, measuring responses, and taking decisions that direct the control flow within a system-level test case
- it should be *non-intrusive*, i.e. we cannot afford having to change the code of the subsystems, and this both for economical reasons and lack of feasibility: most applications are complete black boxes
- it should be *intuitive and controllable* without requiring programming skills. This implies that we need a lightweight, possibly graphical approach to coordination definition, and that easy and timely validation of coordination models should be available.

In our solution, we have adopted a coarse-grain approach to modelling system behaviour, that accounts for the required simplicity and allows direct animation of the models as well as validation via model checking.

3 Coarse-Grained System Models

Instantiating the TC to cover a tool or a CTI application consists of designing a set of application-specific test blocks that cover the relevant behaviour of the application and are executed during the test runs. The test blocks embody coarse granular functionalities of a subsystem, whose implementation is not further formally described. They constitute the *computational core* of the system, are atomic in the coordination model and are not subject to modifications.

These test blocks are used by test designers to graphically construct test cases by drag-and-drop on the TC canvas. The resulting test graphs are directly executable on a system in the field, and, at the same time, they constitute the formal models for verification by means of model checking.

Fig. 2 shows a typical test graph, which illustrates the complexity of the scenarios. Each test block is marked with the name of the subsystem it controls. Some test blocks control directly subsystems-under-test (e.g. when initializing the switch) while others control the corresponding test tools. It is easy to see that even this relatively simple test case needs to access almost all

participants of the test scenario of Fig. 1 in a varied way, and that even for small configurations (only one PC application) the current practice of manual coordination must require specialized personnel, is tedious, time consuming, by far not exhaustive, and error prone.

The central aim of the ITE Test Coordinator is to relieve test engineers from the manual activities of

- programming test blocks, by largely automating the test block generation, and
- executing the system-level test, by automating the coordinative execution (initialization, execution, analysis, and reporting).

For the design of appropriate system-level test cases it is necessary to know what features the system provides, how to operate the system (and the corresponding test tools) in order to stimulate a feature, and how to determine if features work. This information is gathered by the test experts, and after identification of the system's controllable and observable interfaces it is transformed into a set of stimuli (inputs) and verification actions (inspection of outputs, investigation of components' states). Our coarse-grained approach allows test engineers to capture these user-level test activities directly in terms of coordination-level executables test blocks.

3.1 The ITE Component Model

ITE has a very simple **component model**:

(i) a *name* characterizing the block,

(ii) a *class* characterizing the tool, subsystem, or – for test case-specific blocks – the specific management purpose (e.g. report generation) it relates to,

(iii) a set of *formal parameters* that enable a more general usage of the block (e.g. phone ID),

(iv) a set of *branches* which direct the flow of the test execution in dependence of the results of the test block execution (e.g. equal or unequal for the *CheckAgentLabel* block), and

(v) *execution code* written in the coordination language, typically to wrap the actual code that realizes the functionality.

It is easy to see that the *name, class, formal parameters*, and *branches* of this component model provide a very abstract characterization of the components, which will be used later to check the consistency of coordination graphs. The computational portion is encapsulated in the execution code, which is independent of the coordination level, thus it is written (or, as in this application, generated) once and then reused across the different scenarios (e.g. to test several CTI applications).

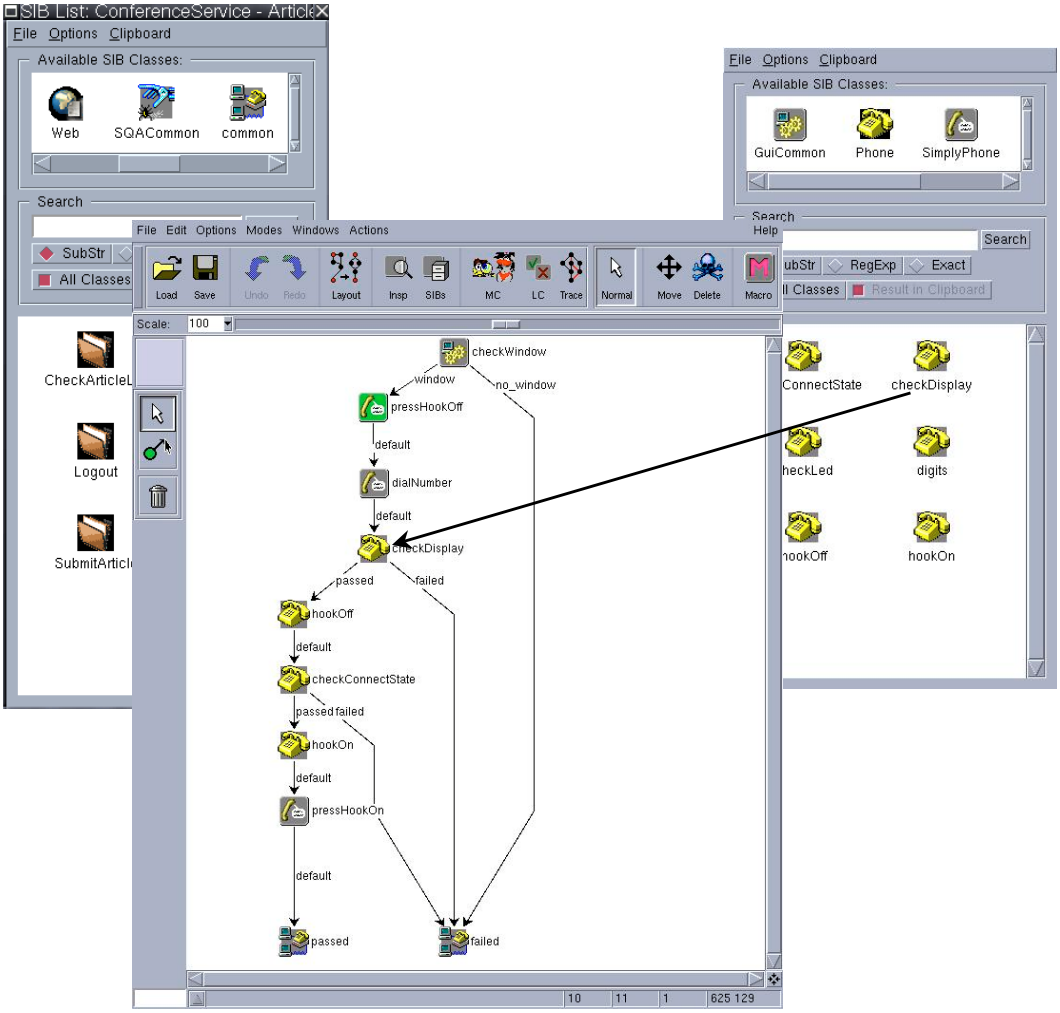


Fig. 3. Fragment of the Taxonomies as SIB Palettes)

3.2 Formal Test Case Models

Test cases are composed of elementary modules, called SIBs (service independent building blocks). The complexity of these SIBs ranges from elementary statements to relatively large procedures steering the routing or application machinery. They are classified in our test design environment in terms of a taxonomy, which reflects the essentials of their profile. A taxonomy is a directed acyclic graph, where sinks represent SIBs, which are atomic entities in the taxonomy, and where intermediate nodes represent groups, that is sets of modules satisfying some basic property (expressed as predicates). Fig. 3 shows a fragment of our taxonomy as it is presented by the ITE test case editor. It shows two snapshots of SIB palettes: on the left we recognize e.g. groups for internet actions (web) and for steering the test tool Rational Robot (SQA-Common), on the right we see groups for telephony activities (GUICommon and Phone).

Test cases are internally modelled as Kripke structures whose nodes represent elementary SIBs and whose edges represent branching conditions:

Definition 3.1
A *test case model* is defined as a triple $(\mathcal{S}, Act, Trans)$ where

- $\mathcal{S}$ is the set of available SIBs
- Act is the set of possible branching condition
- $Trans = \{(s, a, s')\}$ is a set of transitions where $s, s' \in \mathcal{S}$ and $a \in Act$.

Through this non-standard abstraction in our model we obtain a **separation of concerns** between the control-oriented coordination layer, where the test engineer is not troubled with implementation details while designing or evaluating test cases, and the underlying data-oriented communication mechanisms enforced between the participating subsystems, which are hidden in the test block implementation. Our tools support the automatic generation of test blocks according to several communication mechanisms (CORBA [10], RMI [19], and other more application-specific ones), as explained in [11].

3.3 Test Block Libraries

A library of test blocks arose this way at Siemens in a very short time, covering test blocks that represent and implement, e.g. (cf. again Figs. 2 and 3):

Common actions: Initialization of test tools, system components, test cases and general reporting functions,

Switch-specific actions: Initialization of switches with different extensions,

Call-related actions: Initiation and pick up of calls via a PBX-network or a local switch,

CTI application-related actions: Miscellaneous actions to operate a CTI application via its graphical user interface, e.g., log-on/log-off of an agent, establish a conference party, initiate a call via a GUI, or check labels of GUI-elements.

3.4 Generating Test Blocks

This simple and well structured component model enables the automatic generation of coordinable components. In this application domain, only a few components are generated out of directly programmed code (in some script language for some proprietary tools or APIs, e.g. for the communication with the Hipermon). Most components are directly obtained from behaviour recordings during experiments (e.g. the body of the communication "answers" from the Rational Robot). The general structure of most of the test blocks is in fact similar: a parameterized test script written in some typically proprietary language must be transferred to/from the subsystem or its test tool. We provide tools for the automatic generation of tool-specific adapter code that

makes legal test blocks out of such test scripts. Thus the definition of a new test block is rather simple: the tester in the field records a GUI test script which performs some actions, the test engineer defines the abstract component as described above, and the script code is automatically wrapped into a test block that can be directly made available to the test engineers, who graphically construct new test cases.

4 Organization of Coordination in the ITE

All we need to define coordination in the ITE is already provided by the test case model together with the executable code of each test block. The graph structure that we use for the description of test cases also defines a superposition of coordination sequences over the executable code, and it is thus independent of the chosen communication paradigms and of the underlying programming language.

4.1 The Framework

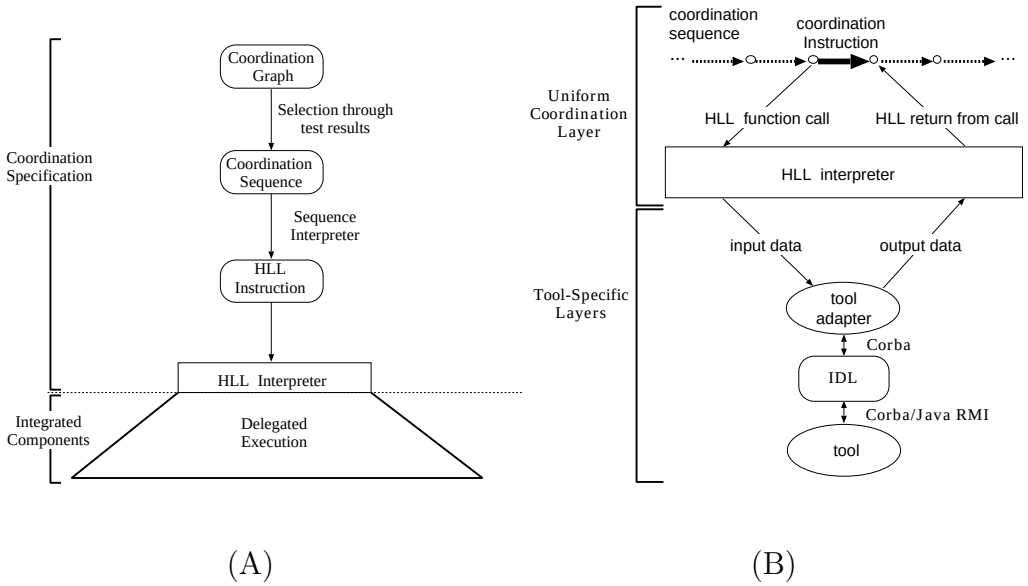
The coordination environment of the ITE bases on the paradigm of application development in the underlying Agent Building Center tool (ABC), which is *coordination-oriented*. In the ABC, application development consists in fact of the behaviour-oriented combination of building blocks on a *coarse* granular level. Building blocks are identified on a functional basis, understandable to application experts, and usually encompass a number of 'classical' programming units (be they procedures, classes, modules, or functions). They are organized in application-specific collections. In contrast to (other) component-based approaches, e.g., for object-oriented program development, the ABC focusses on the dynamic behaviour: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behaviour in terms of control.

Throughout the behaviour-oriented development process, the ABC offers access to mechanisms for the verification of libraries of constraints by means of model checking (Sect. 5). The model checker individually checks hundreds of typically very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in the case of a constraint violation (see Fig. 5), since the feedback is provided on the coordination graph, i.e. at the application level rather than on the code.

These characteristics are the key towards distributing labour according to the various levels of expertise.

Programming Experts: They are responsible for the software infrastructure, the runtime-environment for the compiled services, as well as for programming the single building blocks.

Domain Modelling Experts: They classify the building blocks, typi-



(A) (B)

Fig. 4. The Coordination Environment in the ITE Scenario

cally according to technical criteria like their version or specific hardware or software requirements, their origin (where they were developed) and, here, most importantly, according to their intent for a given application area. The resulting classification scheme is the basis for the constraint definition in terms of modal formulas.

Application Experts: They develop concrete applications just by defining their coordination structure. This happens without programming: they graphically combine building blocks into coarse-granular flow graphs. These coordination graphs can be immediately executed by means of an interpreter, in order to validate the intended behaviour (rapid prototyping). Model checking guarantees the consistency of the constructed graph with respect to the constraint library.

4.2 The Testing Scenario

In the ITE the test cases play the role that applications play in the ABC and we are able to use all the benefit offered by the development environment: in particular, the design environment provides the capability of designing hierarchical test cases, and the interpreter provides an efficient mechanism for test case execution.

4.3 Test Case Execution

The general principle of the test case execution in the ITE is shown in Fig. 4(A). From the coordination point of view, a test case model is interpreted as a coordination graph, where the actually executed coordination sequence (a path in the graph) is determined at runtime by results of the execution of the actual test block. In the concrete test case of Fig. 2, the branching is determined

at the coordination level through the results of the check points (i.e. *checkAgentLabel*, *upnCheckDisplay*, and *checkWindow*). The coordination sequence is executed by means of a sequence interpreter (*tracer* tool): for each test block it delegates the execution to the corresponding execution code. This reflects our policy of separation between coordination and computation: it embodies the superposition of the coordination on the components'code and it enables a *uniform view* on the tool functionalities, abstracting from any specific technical details like concrete data formats or invocation modalities.

Intertool communication is realized via parameter passing and tool functionality invocation by function calls which, via their arguments, pass abstract data to the adapters encapsulating the underlying functionalities as sketched in Fig. 4(B). The functionalities can be accessed via the *Corba* or *Java RMI* mechanism. In the concrete setting of system level tests the input data for the test tools are test scripts, that can be passed to the subsystems by the corresponding test tools (delegation stack principle).

The practical impact of our coordination based test environment exceeded our expectations: ITE has been already successfully used in industrial system-level testing of advanced CTI applications based on Siemens' HICOM family of switches [11,5]. In such scenarios, we have been able to fully automate the test case execution, and to document an efficiency improvement of factors over 30 during the test execution phase [12].

5 Model Checking as an Aid to Test Case Design

In [13] we presented some pragmatic verification-oriented aspects of our solution: we showed how the component-based test design was introduced on top of a library of elementary (but intuitively understandable) test case fragments (the test blocks), and we showed that the correctness and consistency of the test design is fully automatically enforced in ITE via *model checking*. The impact of this approach on the efficiency of test case design and documentation is dramatic in industrial application scenarios.

The ITE contains an iterative model checker based on the techniques of [16]: it is optimized for dealing with large numbers of constraints, in order to allow verification in real time. Concretely, the algorithm verifies whether a given model satisfies properties expressed in a user friendly, natural language-like macro language. In the CTI setting:

- the *properties* express correctness or consistency constraints the target CTI service or the test case itself are required to respect.
- the *models* are directly the coordination graphs, where building block names correspond to atomic propositions, and branching conditions correspond to action names. Figs 2 and 5 show typical test graphs for illustration.

Classes of constraints are formed according to the application domain, to

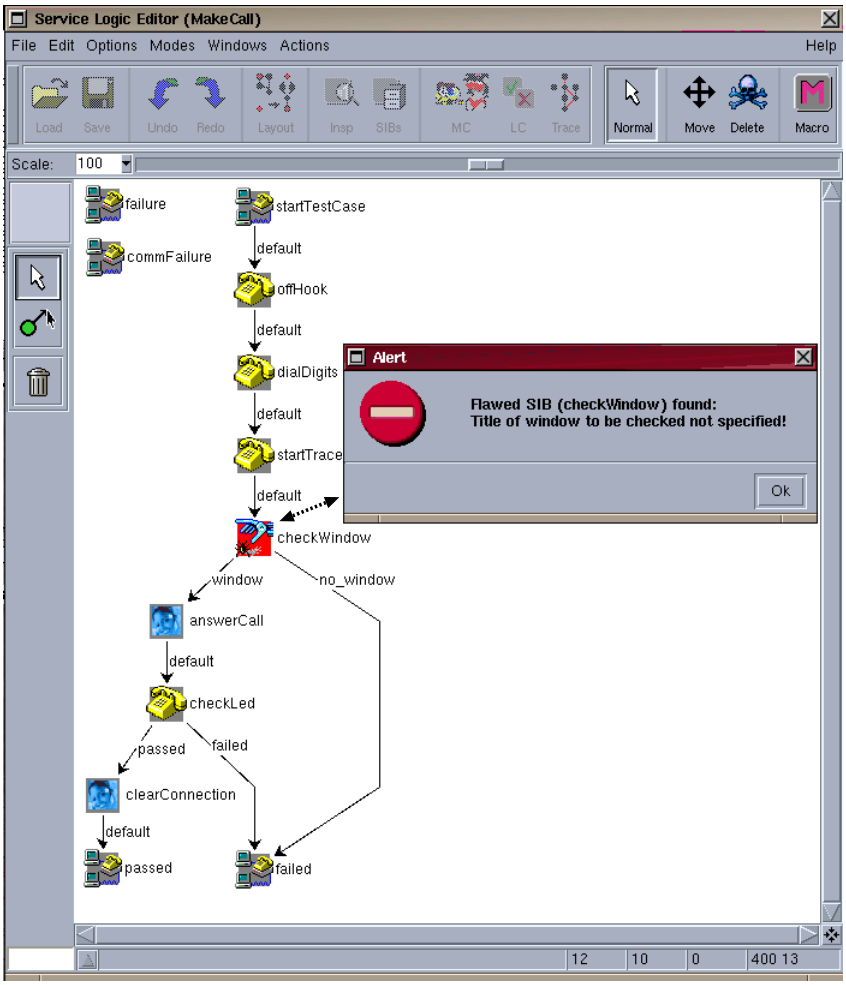


Fig. 5. Test Case Checking in the ITE Environment

the subsystems, and to the purposes they serve. This way it depends on the global test purpose, which constraints are bound to a test case. [3]

5.1 The Logic

Local Constraints.

The overall on-line verification during the design of a new test case captures both local and global constraints. Local constraints specify requirements on single SIBs, as well as their admissible later parameterization.

Whereas the specification of single SIBs is done simply by means of a predicate logic over the predicates expressed in the taxonomy, parametrization conditions are formulated in terms of a library of corresponding predicates. The verification of local constraints is invoked during the verification of the

[3] It was not possible to obtain clearance for publication of confidential material pertaining to the actual implementation of portions of the system, including complex test cases and specific constraints.

global constraints.

Global Constraints: The Temporal Aspect.

Global constraints allow users to specify causality, eventuality and other vital relationships between SIBs, which are necessary in order to guarantee test case well-formedness, executability and other frame conditions.

A test case property is global if it does not only involve the immediate neighbourhood of a SIB in the test case model [4], but also relations between SIBs which may be arbitrarily distant and separated by arbitrarily heterogeneous submodels. The treatment of global properties is required in order to capture the essence of the expertise of designers about do's and don'ts of test case design, e.g. which SIBs are incompatible, or which can or cannot occur before/after some other SIBs. Such properties are rarely straightforward, sometimes they are documented as exceptions in thick user manuals, but more often they are not documented at all, and have been discovered at a hard price as bugs of previously developed test cases. This kind of domain-specific knowledge accumulated by experts over the years is particularly worthwhile to include in the design environment for automatic reuse.

In the ITE, such properties are gathered in a Constraint Library, which can be easily updated and which is automatically accessed by the model checker during the verification.

Global constraints are expressed internally in the *modal mu-calculus* [9]. The following negation-free syntax defines mu-calculus formulas in positive normal form. They are as expressive as the full modal mu-calculus but allow a simpler technical development.

$$\Phi ::= A \mid X \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid [a]\Phi \mid \langle a\rangle\Phi \mid \nu X.\,\Phi \mid \mu X.\,\Phi$$

In the above, $a \in \mathit{Act}$, and $X \in \mathit{Var}$, where A is given by the SIB taxonomy, Act by the library of branching conditions, and Var is a set of variables. The fixpoint operators νX and μX bind the occurrences of X in the formula behind the dot in the usual sense. Properties are specified by *closed* formulas, that is formulas that do not contain any free variable.

Formulas are interpreted with respect to a fixed labeled transition system $\langle \mathcal{S}, \mathit{Act}, \rightarrow\rangle$, and an environment $e : \mathit{Var} \rightarrow 2^{\mathcal{S}}$. Formally, the semantics of the mu-calculus is given by:

$$\begin{aligned}
[\![X]\!]e &= e(X)\\
[\![\Phi_1 \vee \Phi_2]\!]e &= [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e\\
[\![\Phi_1 \wedge \Phi_2]\!]e &= [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e\\
[\![[a]\Phi]\!]e &= \{\, s \mid \forall s'.\ s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e \,\}
\end{aligned}$$

[4] I.e., the set of all the predecessors/successors of a SIB along all paths in the model.

$$[\![\langle a\rangle\Phi]\!]e = \{\, s \mid \exists s'.\ s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e \,\}$$
$$[\![\nu X.\Phi]\!]e = \bigcup\{S' \subseteq \mathcal{S} \mid S' \subseteq [\![\Phi]\!]e[X \mapsto S']\}$$
$$[\![\mu X.\Phi]\!]e = \bigcap\{S' \subseteq \mathcal{S} \mid S' \supseteq [\![\Phi]\!]e[X \mapsto S']\}$$

Intuitively, the semantic function maps a formula to the set of states for which the formula is "true". Accordingly, a state s satisfies $A \in \mathcal{A}$ if s is in the valuation of A, while s satisfies X if s is an element of the set bound to X in e. The propositional constructs are interpreted in the usual fashion: s satisfies $\Phi_1 \vee \Phi_2$ if it satisfies one of the Φ_i and $\Phi_1 \wedge \Phi_2$ if it satisfies both of them. The constructs $\langle a\rangle$ and $[a]$ are *modal operators*; s satisfies $\langle a\rangle\Phi$ if it has an a-derivative satisfying Φ, while s satisfies $[a]\Phi$ if each of its a-derivatives satisfies Φ. Note that the semantics of $\nu X.\ \Phi$ (and dually of $\mu X.\Phi$) is based on Tarski's fixpoint theorem [Tars55]: its meaning is defined as the greatest (dually, least) fixpoint of a continuous function over the powerset of the set of states.

For the project we provide a simple 'sugared' version of LTL, which is translated into the modal mu-calculus for model checking. Here it is important to provide a natural language-like feeling for the temporal operators. As indicated by the example below, the standard logical connectors turned out to be unproblematic. We omit the formal definition of **next**, **generally**, **eventually**, and **until** here, as they are standard. In addition, we have implemented a pattern-driven formula editor which further simplifies the extension of the constraint library.

5.2 Expressing Test Case Properties

The library of constraints is also structured according to the main purposes addressed by the constraints.

Legal Test Cases: Constraints in this class define the characteristics of a correct test case, independently of any particular system under test and test purpose. Specifically, testing implies an evaluation of the runs wrt. expected observations done through *verdicts*, represented through the predicates passed and failed. For example, to enable an automated evaluation of results, verdict points should be disposed in a *nonambiguous* and *noncontradictory way* along each path, which is expressed (in a more user-friendly syntax) as follows:

$$(\mathsf{passed} \ \vee \mathsf{failed}) \Rightarrow \ \mathbf{next}(\mathbf{generally}\ \neg(\mathsf{passed} \ \vee \mathsf{failed}\ ))$$

POTS Test Cases: these constraints define the characteristics of correct functioning of Plain Old Telephone Services (POTS), which build the basis of any CTI application behaviour. Specific constraints of this class concern the different signalling and communication channels of a modern phone with an end user: signalling via tones, messaging via display, optic signalling via LEDs, vibration alarm. They must e.g. all convey correct and consistent information.

System Under Test -Specific Test Cases: these constraints define the

correct initialization and functioning of the single units of the system under test (e.g. single CTI applications, or the switch), of the corresponding test tools, and of their interplay. Fig. 5 shows the detection of a mismatch for this class of correctness criteria. Here, a misconfiguration in the test case definition is discovered: the identifier (the "title") of the window expected to appear on the Call Center Agents PC after a call has been started in the switch should have been specified before it can be accessed by Rational Robot. In fact, the Robot (the GUI test tool for the PC Call Center Application) needs this information in order to check that this window appropriately appears on the screen. As we see in this example, also the diagnostic information in case of property violation is provided in a user-friendly way: the violating path is the one that leads to the highlighted (red, instead of gold) Rational Robot test block, and a verbal formulation of the failed property allows the test designer to spot the problem without need to master temporal logics.

6 Related Work

Our work differs both from the usual approach to test definition and generation and from the usual attitude in the coordination community.

Most research on *test automation for telecommunication systems* concentrates on the generation of test cases and test suites on the basis of a formal model of the system: academic tools, like TORX [21], TGV [3,7], Autolink [15], and commercial ones like Telelogic Tau [20] presuppose the existence of fine-granular system models in terms of either automata or SDL descriptions, and aim at supporting the generation of corresponding test cases and test suites. This approach was previously attempted in the scenario we are considering here, but failed to enter practice because it did not fit the current test design practice, in particular because there did not exist any fine granular formal model of the involved systems.

The requirements discussed in this paper exceed the capabilities of todays commercial testing tools. To our knowledge there exist neither commercial nor academic tools providing comprehensive support for the whole system-level test process. We combine commercial test tools (in this case Rational Robot [14], Hipermon [6]) that deal with the subsystems of the considered scenario in order to capture the global test process.

Concerning *coordination approaches*, the closest to ours is in our opinion that of [1], which proposes the use of coordination contracts to promote the separation of the coordination aspects that regulate the way objects interact in a system, from the way objects behave internally. Like for us, their main concern is supporting evolutionary aspects of the whole system. In their work, contracts fulfill a role similar to architectural connectors: they make these coordination features available as first-class citizens, so that it is possible to treat them distinctly from the functionality of the components. Contracts

are based on superposition mechanisms [8] for supporting forms of dynamic reconfiguration of systems. These mechanisms enable contracts to be added or replaced without the need to change the objects to which they apply. [4] describes CDE, an environment for developing coordination contracts in Java. The CDE approach is still programming-oriented: unlike our coordination graphs, contracts must be programmed, they do not (yet) support macros or hierarchy, and no automatic verification for contracts is available. In our application domain, scalability of the approach is a major demand! It must be applicable to a regression testing scenario of hundreds of complex applications with very high regression frequencies. Accordingly, it is important that contracts (for us, test cases/coordination graphs are the global contracts) be

- definable in a programming-free fashion,
- themselves largely reusable, since we coordinate large behaviours,
- hierarchical, and
- subject to the validation of "reusability" of contracts via model checking in different contexts.

Indeed, hierarchy and formal verification of test cases are appreciated and heavily used features of the ITE environment.

7 Conclusion

Coordination graphs provide a useful abstraction mechanism to be used in conceptual modelling because they direct developers to the identification and promotion of interactions as first-class citizens, a pre-condition for taming the complexity of system construction and evolution. Their incarnation for system-level test case application has proven the feasibility of the approach and, more importantly, its adequacy for adoption in an industrial environment. Coarse grained modelling was natural for the test engineers, who are used to a functionality-oriented macro-model of the systems, of the test tools, and of the applications they test. It was also gracefully fitting with their pre-existing practice. Lightweight coordination solved the problem of keeping track of continuously evolving subsystems. The graphical configuration of test cases was perceived as intuitive, easy to manage, and for the first time providing an illustrative means to depict system-wide behaviours. The additional benefit of verification of the test cases wrt. abstract requirements or conventions was also perceived as useful and economically productive: it greatly enhanced reusal of test cases in similar contexts, and it allowed fast debugging of new test cases to take care of changed contexts or settings.

Our partners are confident that the scalability of the approach to the next generation of switches (which will involve widely networked and mobile applications) is within reach.

Acknowledgements

This is joint work with G. Brune, H.-D. Ide, W. Goerigk, B. Hammelmann, and A. Erochok (SIEMENS ICN Witten), A. Hagerer, O. Niese and M. Nagelmann (METAFrame Technologies GmbH Dortmund).

References

[1] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes, M. Wermelinger: *Coordination Technologies For Component-Based Systems*, to appear at Integrated Design and Process Technology, IDPT-2002, Pasadena (CA), June, 2002, Society for Design and Process Science.

[2] V. Braun, T. Margaria, B. Steffen: *The Electronic Tool Integration Platform* appears in the Special Theme Issue on "Internet Based Technology Transfer Services" of the Journal Asia Pacific Tech Monitor.

[3] J.-C. Fernandez, C. Jard, T. Jéron, C. Viho: *An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology*, Science of Computer Programming, 29, 1997.

[4] J. Gouveia, G. Koutsoukos, L. Andrade, J. Fiadeiro: *Tool Support for Coordination Based Evolution*, Proc. TOOLS 38, W.Pree (ed.), IEEE Computer Society Press 2001, pp. 184-196.

[5] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, H.-D. Ide: *An Efficient Regression Testing of CTI Systems: Testing a complex Call-Center Solution*, Accepted for publication in Annual Review of Communic., Vol. 55, Int. Engineering Consortium, Chicago, 2001.

[6] Herakom GmbH, Germany, `http://www.herakom.de`.

[7] C. Jard, T. Jeron: *TGV: Theory, Principles and Algorithms*,Proc. Int.Symposium on Integrated Design and Process Technology 2002, Pasadena, June 2002, (to appear).

[8] S. Katz: *A Superimposition Control Construct for Distributed Systems*, ACM TOPLAS 15(2), 1993, pp. 337-356.

[9] D. Kozen: *Results on the Propositional μ-Calculus*, Theoretical Computer Science, Vol. 27, 1983, pp. 333-354.

[10] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Revision 2.3, Object Management Group, 1999.

[11] O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, H.-D. Ide: *An Automated Testing Environment for CTI Systems Using Concepts for Specification and Verification of Workflows*, In Annual Review of Communic., Vol. 54, Int. Engineering Consortium, Chicago, 2000.

[12] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, W. Goerigk, H.-D. Ide: *An Automated Regression Testing of CTI Systems*, In Proc. IEEE European Test Workshop 2001, pp. 51-57, Stockholm (S), 2001.

[13] O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, H.-D. Ide: *Library-based Design and Consistency Checking of System-level Industrial Test Cases*, FASE 2001, Int. Conf. on Fundamental Aspects of Software Engineering, Genova, LNCS 2029, Springer Verlag, 2001, pp. 233-248.

[14] Rational, Inc.: *The Rational Suite description*, `http://www.rational.com/products`.

[15] M. Schmitt, B. Koch, J. Grabowski, D. Hogrefe: *Autolink - A Tool for Automatic and Semi-automatic Test Generation from SDL-Specifications*, Technical Report A-98-05, Medical Univ. of Lübeck, Germany, 1998.

[16] B. Steffen, A. Claßen, M. Klein, J. Knoop, T. Margaria: *The Fixpoint Analysis Machine*, (invited paper) CONCUR'95, Pittsburgh (USA), August 1995, LNCS 962, Springer Verlag.

[17] B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, Int. J. STTT (1997)1, Springer Verlag, 1997, pp. 9-30.

[18] B. Steffen, T. Margaria: *METAFrame in Practice: Intelligent Network Service Design*, In *Correct System Design – Issues, Methods and Perspectives*, LNCS 1710, Springer Verlag, 1999, pp. 390-415.

[19] Sun: *Java Remote Method Invocation.* `http://java.sun.com/products/jdk/rmi`.

[Tars55] A. Tarski: *A Lattice-Theoretical Fixpoint Theorem and its Applications*, Pacific Journal of Mathematics, V. 5, 1955.

[20] Telelogic: *Telelogic Tau*, `http://www.telelogic.com`.

[21] J. Tretmans, A. Belinfante: *Automatic testing with formal methods*, In EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review - EuroStar Conferences, Galway, Ireland, November 8-12, 1999.

Andrew D. Gordon (Microsoft Research, Cambridge, UK)

Authenticity Types for Cryptographic Protocols

based on joint work with Alan Jeffrey, DePaul University (Chicago, IL, USA)

Cryptographic protocols are essential for the security of many critical networking applications, such as authenticating various financial transactions. Moreover, many new consumer-to-business and business-to-business protocols are being proposed and need cryptographic protection. It is famously hard to specify and verify such protocols, even if we assume that the underlying cryptographic algorithms cannot be cryptanalysed. My invited talk describes a new approach to specifying and verifying authenticity properties of security protocols, based on a typed process algebra. The theory has been developed in a series of papers with Alan Jeffrey. Our approach requires little human effort per protocol, puts no bound on the size of the opponent attacking the protocol, and requires no state space enumeration. Moreover, the types for protocol data provide Some intuitive explanation of how the protocol works. My talk explains the basic ideas by example, states our main results, and outlines our plans for applications of this technology.

References

A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In 15th IEEE Computer Security Foundations Workshop (CSFW 2002), Cape Breton, June 24-26, 2002. IEEE Computer Society.

A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In 14th IEEE Computer Security Foundations Workshop (CSFW 2001), pages 145-159, Cape Breton, June 11-13, 2001. IEEE Computer Society. A journal version is to appear in the Journal of Computer Security.

A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication

A methodological process for the design of a large system: two industrial case-studies.

Nestor Lopez, Marianne Simonot, Véronique Viguié Donzeau-Gouge

Cedric Research Laboratory, Conservatoire National des Arts et Métiers,
292 Rue Saint-Martin, 75141 Paris Cedex, France.
email: (donzeau, lopezne, simonot)@cnam.fr

Abstract

This paper presents two examples taken from industrial case-studies that have been specified using an event system approach. Component specifications, taking the form of pre-post formula, have been derived. Constraints which ensure the correctness of the whole process are given.

1 Introduction

Our purpose is to present a methodological process which can be used for an *entire* system development. We define a system according to the properties it must satisfy and express them in a formal way. This process leads us to build two sorts of mathematical models: event models and operational models. They define the formal specification of the entire system. These models are proved to be consistent and the correctness of the whole process is ensured, i.e. the correctness of the implementation of each component and the correctness of the implementation of the entire system.
The applications we have in mind heavily interact with their environment. They may use existing hardware and software components.
Addressed issues: A first issue is the big quantity of information required by this kind of specification, which makes necessary to introduce a certain hierarchy into the expressed ideas. This is done in two ways: by defining the more abstract aspects and gradually focusing upon details; and by grouping properties into specific categories (properties which describe the interactions between components and with the environment, and properties which are local to one component, like functional properties). We use refinement techniques [Back 88], [Morg 90], [Abr 96-1], [But 96] to support the abstraction i.e. to introduce details and to express some design decisions.
A second issue is how to specify interactions of a system with its environment. The trend is to build a closed model [Abr 96-2] which includes the

whole environment. However, this model can be extremely complex. In our approach, we develop a closed model which specifies only the interactions of the system with the environment. Furthermore, refining a closed system might lead to a program failing to implement the specification we have in mind [Lop,Sim,Don 00].

Choices: The formal specification language must allow the expression of nondeterministic specifications and has to be supported by efficient proof assistants, So we neither combine different logic nor introduce new ones. We largely rely on existing proven methods backed by tools. We adopt the set theory as well as the refinement techniques as they are encapsulated in the actual tools [Atelier B], [B Toolkit].

Process: We start from an informal specification written in a natural language which describes all the properties the system must satisfy. Domain experts participate in the elaboration and validation of this document. When system details are introduced, some implementation choices can be made: for example re-using existing devices with well-defined interfaces.

Then we build, step by step, an event model which only describes the interactions of the system with its environment and between the different components. It expresses the hypothesis that we make on the environment. We only keep from the initial properties those that express interactions. This process is controlled by event refinement techniques as defined in [Lam,Sha 90], [But 96], [Abr 96-2], [Back,Kur 88].

After this, the event model is transformed into a shared module which is used in the specification of components. This module provides the external and internal communications mechanisms of the system. It is called *interface module*. Roughly speaking, by module we mean a set of program specifications which can be refined to sequential program implementations.

Finally, the specification of each component is written separately. It incorporates the properties of the informal specification related to only to this component, and which therefore, have not yet been taken into account. Component specifications are used as the starting point for the usual refinement process for sequential programs [Back 88], [Abr 96-1].

The correctness of the whole process is presented in [Lop 02]. In this article, the conditions that ensure this correctness will only lightly be touched upon.

In the rest of this paper, we treat two examples extracted from industrial case-studies which are specified and partially proven using [Atelier B]. They are presented in the syntax of this tool and some comments are added to help the reader. However, the B method does not include two important notions we use: shared modules and auxiliary variables. These notions are introduced and formally defined in [Lop 02]. They tend to extend the utilization of the B method to concurrent systems and distributed systems. The first example illustrates the way we use to specify a large system. The second example illustrates the way we use to derive component specifications. The event spec-

ification of this example is detailed in [Lop,Sim,Don 00].

2 An event based specification: The passenger exchange function of the Météor Metro Line

Meteor is a new and totally automated metro line built in Paris. The line is formed by trains and arrival/departure platforms. Trains and platforms have doors. The passenger exchange function [1] must ensure the passengers' safety, which is built upon four independent principles that can be expressed as follows:
P0 An exchange cannot generate an unsafe situation.
P1 To guarantee the passengers' safety, it is necessary to ensure the immobility of the train during the exchange.
P2 Passengers are safe if the trains and the platforms are closed universes.
P3 If a dangerous situation occurs, the passengers should have the possibility of leaving the train at any time to get a safe environment.
Several kinds of passenger exchanges can be identified: the usual ones, and emergency evacuations which happen when a dangerous situation is detected. Such an evacuation has to be launched as soon as a danger occurs [2].

2.1 Recalls

Events. An event specification takes the form $\exists a.\ G(x,a) \wedge A(x,a,x')$. The guard constraints the occurrence of the event and the action specifies its effect. The variable a allows to express the external non-determinism, i.e., external values provided by the environment. The guard $G(x,a)$ defines the states in which the event can be observed, the action $A(x,a,x')$ relates system states before and after the observation. The concrete syntax of an event (in [Abr 96-2]) is any a where $G(x,a)$ then $S_{x,a}$ where $S_{x,a}$ is the generalized substitution which is a translation of $A(x,a,x')$. For instance $x' = x \cup a$ is translated into $x := x \cup a$.
Event Refinement. Let $E_1(x,x')$ be of the form $\exists a.\ G_1(x,a) \wedge A_1(x,a,x')$ and $E_2(w,w')$ be of the form $\exists a.\ G_2(w,a) \wedge A_2(w,a,w')$, the specifications of two events. They are defined on two different state spaces. Let $J(x,w)$ be a total relation defined between these two spaces, the formula $E_1 \sqsubseteq_J E_2 =_{def}$ $\forall x,w,w'.\ \exists a.\ (J(x,w) \wedge G_2(w,a) \wedge A_2(w,a,w')) \Rightarrow \exists a,x'.\ G_1(x,a) \wedge A_1(x,a,x') \wedge J(x',w')$ expresses the fact that all state changes observed with E_2 are also observed with E_1.
Event System. An event system $Ss = (x, IS, C, E_i)$ is formed by a shared

[1] This case-study has been proposed by the RATP and Matra Transport International within the framework of an agreement between the CNAM computer research center (lab. CEDRIC), RATP, MTI(SIEMENS), and STERIA under the contract DJ.0246/98.
[2] This presentation is a simplified version of this problem -a more detailed description is provided in [Lop 02], [Lop 99].

variable x that represents the system state, an invariant IS that expresses static properties of the system, the specification C that initializes the system and the set of events E_i for $i \in 1..n$ that defines the dynamics. Such systems are transition systems, whose initial states are defined by the initialization, and whose transitions are defined by the events.
System Consistency. An event system is said to be consistent if the invariant is established by the initialization and preserved by each event.
System Refinement. Let S_1 and S_2 be two event systems, let $J(x, w)$ be a total relation between their variables, S_1 is refined by S_2 ($S_1 \sqsubseteq_J S_2$) if each event of S_2 is a refinement of the corresponding event of S_1, and if the initialization of S_2 is a refinement of the initialization of S_1. The corresponding proof obligations are generated by the actual tools.
Adding new events. According to [Abr 96-2], [Lop 02], it is possible during a refinement:

- to add new events. For this, we have just to assume that the new event is present in the abstract system with $true \wedge skip$ as its specification. $skip$ means that the new event does not modify any variable of the abstract system. This means that when a new variable is introduced, the refinement must include all the events which modify this variable. This constrains the way events are introduced.
- to split an event. This decomposition allows us to observe an event with greater precision. In order to do this, it suffices to assume that in the abstract system, the abstract event E is duplicated as many times as needed and that each event in the decomposition must be a refinement of E.

2.2 First model

This model introduces the main notions of the system: trains tt, platforms qq, and their possible states (variables): unsafe (td, qd) or safe ($tt - td$, $qq - qd$). At this level, only the property P0 can be expressed. We can observe five events:

- `UnsafeTrain`, `SafeTrain` which modify the variable td.

- `UnsafePlatf`, `SafePlatf` which modify the variable qd.

- `Transfer` which has no effect upon the environment and can be activated at any time.

Note that `Transfer` can occur at any time and not only in a safe situation. Note also that the property P0 is expressed by the fact that the action of `Transfer` does not modify the system state.

SYSTEM $Meteor1.1$
SETS tt; qq /* Trains, platforms of the line */
VARIABLES td, qd /* Unsafe trains and platforms */
INVARIANT $td \subseteq tt \quad \wedge \quad qd \subseteq qq$
INITIALISATION $td, qd := \emptyset, \emptyset$ /* The line is safe */
EVENTS
Transfer $\widehat{=}$
SELECT $TRUE$ /* An exchange is always observable. */
THEN $SKIP$ /* The safety is not modified (P0). */
END;

UnsafeTrain $\widehat{=}$ /* A safe train becomes unsafe */
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin td$
THEN $td := td \cup \{t1\}$
END;

SafeTrain $\widehat{=}$ /* An unsafe train becomes safe */
ANY $t1$ WHERE $t1 \in td$
THEN $td := td - \{t1\}$
END;

UnsafePlatf $\widehat{=}$ /* $q1$ becomes unsafe */
ANY $q1$ WHERE $q1 \in qq \wedge q1 \notin qd$
THEN $qd := qd \cup \{q1\}$
END;

SafePlatf $\widehat{=}$ /* $q1$ becomes safe */
ANY $q1$ WHERE $q1 \in qd$
THEN $qd := qd - \{q1\}$
END

The proofs of consistency of this model are generated and discharged by the Atelier B. We can prove additional properties using this model. For example, deadlock freeness is implied by the fact that under the invariant, the disjunction of the guards is always true. We can also prove properties of event traces the system should allow: for instance, `Transfer` can be observed at any time.

2.3 Second model

This model introduces the notion of *locked train* needed to express property P1. A locked train is totally stopped and to start moving, some actions need to be executed before.

The variable ti models the set of locked trains. Hence, two new events `LockedTrain` and `UnlockedTrain` are introduced.

At this level, we want to observe separately two kinds of exchanges : normal exchanges and other cases. To differentiate these two cases, we distinguish those trains which are doing a passenger exchange in normal conditions: variable ttv. This partition is done by decomposing the event `Transfer` of the previous model into two events `EndPassExch` (normal exchange) and `Transfer1` (other exchanges). An event, `StartPassExch`, must be added to modify ttv. It allows to observe the beginning of a normal passenger exchange.

A normal exchange is done when a train is in ttr. Hence, ttv only includes trains which satisfy this property. To express this, we introduce a partial function t_q from trains to platforms, the variable qtv denotes the set of trains which are doing a passenger exchange in the normal conditions.

`StartPassExch` is activated if the train is stopped in front of a platform, and hence, t_q appears in its guard. Other events that modify this variable are introduced: `DepartureFromPlatf` and `ArrivalToPlatf`.

Property P1 is an invariant of the system. The first four events of the system $Meteor1.1$ are included here in addition to the events that we have just introduced. The first four events are still not synchronized. The new events are synchronized as follows:

`LockedTrain` → `Transfer1` → `UnlockedTrain`

`ArrivalToPlatf` → `LockedTrain` → `StartPassExch` → `EndPassExch` → `UnlockedTrain` → `DepartureFromPlatf`.

`LockedTrain` → `UnlockedTrain`

The guarantee of this synchronization is obtained by proof obligations involving only guards.

SYSTEM $Meteor1.2$

VARIABLES

td, qd, /* Unsafe trains and platforms */

t_q, /* Trains in front of platforms */

ttr, /* Trains doing an exchange */

ttv, **qtv**, /* Trains and platforms doing a passenger exchange */

ti /* Locked trains */

INVARIANT

$td \subseteq tt \ \wedge \ qd \subseteq qq \ \wedge$

$\mathbf{t\_q} \in \mathbf{tt} \rightarrowtail\!\!\!\!\!\!\rightarrow \mathbf{qq} \ \wedge$

$\mathbf{ttv} \in \mathbf{dom(t\_q)} \ \wedge$

$\mathbf{ttv} \subseteq \mathbf{ttr} \ \subseteq \mathbf{ti} \subseteq \mathbf{tt} \ \wedge$ /* includes **(P1)** */

$\mathbf{qtv} = \mathbf{t\_q[ttv]} \ \wedge$

INITIALISATION $t\_q, ttr, ttv, td, ti, qtv, qd := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

EVENTS

UnsafeTrain $\hat{=}$ ...

SafeTrain $\hat{=}$ ...

StartPassExch $\hat{=}$ /* $t1$ starts a Transfer */

ANY $t1$ WHERE

$t1 \in ti \wedge t1 \notin ttr \wedge$

$t1 \in dom(t\_q) \wedge t\_q(t1) \notin qtv$

THEN $ttv, qtv, ttr := ttv \cup \{t1\},$

$qtv \cup t\_q[t1], ttr \cup \{t1\}$

END;

UnsafePlatf $\hat{=}$ ...

SafePlatf $\hat{=}$ ...

EndPassExch $\hat{=}$ /* $t1$ finishes a Transfer */

ANY $t1$ WHERE

$t1 \in ttv$

/* $t1$ is in Transfer */

THEN $ttv, qtv, ttr := ttv - \{t1\},$

$qtv - t\_q[t1], ttr - \{t1\}$

END;

Transfer1 $\hat{=}$

ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttv$ /* $t1$ is not in Transfer */

THEN $SKIP$ /* end of the Transfer */

END;

LockedTrain $\hat{=}$
/* An unlocked train becomes locked */
ANY $t1$ WHERE $t1 \in tt \wedge t1 \notin ti$
THEN $ti := ti \cup \{t1\}$
END;

UnlockedTrain $\hat{=}$
/* A locked train becomes unlocked */
ANY $t1$ WHERE $t1 \in ti \wedge t1 \notin ttr$
THEN $ti := ti - \{t1\}$
END;

ArrivalToPlatf $\hat{=}$
/* A train arrives to a (free) platform */
ANY $t1, q1$ WHERE $t1 \in tt \wedge$
$q1 \in qq \wedge t1 \notin dom(t\_q) \wedge$
$q1 \notin ran(t\_q) \wedge t1 \notin ti$
THEN $t\_q := t\_q \cup \{t1 \mapsto q1\}$
END;

DepartureFromPlatf $\hat{=}$
/* A train is leaving a platform */
ANY $t1$ WHERE
$t1 \in dom(t\_q) \wedge$
$t1 \notin ti$
THEN $t\_q := \{t1\} \lhd\!\!\!- \; t\_q$
END;

2.4 Third model

In this model, we observe the normal openings of the doors. It introduces the safety property P2 which can be rewritten as follows: (P2.1) a train which is not closed is in a passenger exchange state, (P2.2) a platform which is not closed is a platform on which a passenger exchange is currently taking place or which is in a safe state. We introduce open trains *to* and open platforms *qo* as well as four events `OpenedTrain`, `ClosedTrain`, `OpenedPlatf`, `ClosedPlatf`, which modify these variables. These events detail the transition `StartPassExch` $\rightarrow$ `EndPassExch`. Their guards define the following synchronization:

$$\texttt{StartPassExch} \rightarrow \left|\begin{array}{l}\texttt{OpenedTrain}\\ \\ \texttt{OpenedPlatf}\end{array}\right| \rightarrow \left|\begin{array}{l}\texttt{ClosedTrain}\\ \\ \texttt{ClosedPlatf}\end{array}\right| \rightarrow \texttt{EndPassExch}$$

The doors of a platform are opened only if the platform is safe. Here, we introduce the variable *qs* and, therefore, two events `SafeZone` and `UnsafeZone` which allow to modify this variable. These events detail the opening process of a platform. `SafeZone` must occur before `OpenPlatf` and `UnsafeZone` before `ClosedPlatf`.

SYSTEM *Meteor*1.3
VARIABLES
td, *qd*, *t_q*, *ttr*, *ttv*, *qtv*, *ti*,
to, **qo**, /* Opened trains and platforms */
qs, /* safe platform zones */

INVARIANT

$td \subseteq tt \wedge qd \subseteq qq \wedge$
$ttv \subseteq ttr \subseteq ti \subseteq tt \wedge$ /* includes **(P1)** */
$\mathbf{to} \subseteq \mathbf{ttr} \subseteq \mathbf{tt} \wedge$ /* includes **(P2.1)** */
$\mathbf{qs} \subseteq \mathbf{qq} \wedge$
$t\_q \in tt \rightarrowtail\!\!\!\!\twoheadrightarrow qq \wedge$
$ttv \in dom(t\_q) \wedge qtv = t\_q[ttv] \wedge$
$\mathbf{qo} \subseteq \mathbf{qq} \wedge$
$\mathbf{qo} \subseteq \mathbf{qtv} \cup \mathbf{qs}$ /* **(P 2.2) */**

INITIALISATION $t\_q, to, ttr, ttv, td,\ ti, qo, qtv, qd, qs := \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset$

EVENTS

UnsafeTrain $\widehat{=}$ ...
SafeTrain $\widehat{=}$ ...
StartPassExch $\widehat{=}$ ...
Transfer1 $\widehat{=}$ ...
LockedTrain $\widehat{=}$ ...
ArrivalToPlatf $\widehat{=}$ ...

OpenedTrain $\widehat{=}$ /* $t1$ becomes opened */
ANY $t1$ WHERE $t1 \in ttr \ \wedge t1 \notin to$
THEN $to := to \cup \{t1\}$
END;

OpenedPlatf $\widehat{=}$ /* $q1$ becomes opened */
ANY $q1$ WHERE $q1 \in qtv \cup qs \wedge q1 \notin qo$
THEN $qo := qo \cup \{q1\}$
END;

SafeZone $\widehat{=}$ /* $q1$ zone becomes safe */
ANY $q1$ WHERE $q1 \in qq \ \wedge q1 \notin qs$
THEN $qs := qs \cup \{q1\}$
END;

UnsafePlatf $\widehat{=}$ ...
SafePlatf $\widehat{=}$ ...
EndPassExch $\widehat{=}$
ANY $t1$ WHERE $t1 \in ttv \wedge$ /* $t1$ is in a transfer: */
$t1 \notin to \ \wedge t\_q(t1) \notin qo$
THEN $ttv, qtv, ttr := ttv - \{t1\},$
$qtv - t\_q[t1], ttr - \{t1\}$ /* End of the transfer */
END;

UnlockedTrain $\widehat{=}$ ...
DepartureFromPlatf $\widehat{=}$ ...

ClosedTrain $\widehat{=}$ /* $t1$ becomes closed */
ANY $t1$ WHERE $t1 \in to$
THEN $to := to - \{t1\}$
END;

ClosedPlatf $\widehat{=}$ /* $q1$ becomes closed */
ANY $q1$ WHERE $q1 \in qo$
THEN $qo := qo - \{q1\}$
END;

UnsafeZone $\widehat{=}$ /* $q1$ zone becomes unsafe */
ANY $q1$ WHERE $q1 \in qs$
THEN $qs := qs - \{q1\}$
END;

2.5 Fourth model

In this model, we introduce the property P3 and we observe an emergency evacuation. In this case, passengers have to leave the train, so the train state must allow the opening of the doors, and an access to a safe circulation zone must also be provided. All of the trains which are in an emergency evacuation are modeled by the variable teu. We have two events which allow us to observe the beginning and the end of an emergency evacuation: `StartEva` and `EndEva`.

New notions: the evacuation zones ze, and among them, the subset of evacuation zones which are safe zs. We must be able to locate a train with respect to the evacuation zones; this is the role of the function t_z. The events which modify t_z (`ArrivingZoneEva` and `LeavingZoneEva`) and zs, (`SafeZoneEva` and `UnsafeZoneEva`) are introduced. The localization of platforms into the evacuation zones is defined by the relation q_z.

As previously, the event `Transfer1` is decomposed into two events : `EndEva`

and `Transfer2`. `EndEva` models the end of an emergency evacuation, the event `StartEva` puts the train in a state of evacuation (*teu* is modified).

And just like the previous systems, guards ensure the expected synchronization.

SYSTEM *Meteor*1.4

SETS **ze** /* Evacuation zones */

CONSTANTS **q_z** /* Localization of platforms into zones */

PROPERTIES **q_z** $\in$ **qq** $\leftrightarrow$ **ze**

VARIABLES

td, *qd*, *t_q*, *ttr*, *ttv*, *qtv*	*ti*,*t*0, *q*0, *qs*,
t_z /* Localization of trains */	**teu**, **zs** /* Urgent evacuations and evacuation zones */

INVARIANT

$td \subseteq tt \wedge qd \subseteq qq \wedge t\_q \in tt \rightarrowtail qq \wedge$	$ttv \subseteq ttr \subseteq ti \subseteq tt \wedge$ /* includes **(P 1)** */
$ttv \in dom(t\_q) \wedge qtv = t\_q[ttv] \wedge$	$to \subseteq ttr \subseteq tt \wedge$ /* includes **(P 2.1)** */
$qo \subseteq qq \wedge$	$qs \subseteq qq \wedge$
$qo \subseteq qtv \cup qs$ /* **(P2.2) */**	**t_z** $\in$ **tt** $\leftrightarrow$ **ze** $\wedge$ **dom**(**t_z**) = **tt** $\wedge$
t_q^{-1}; **t_z** $\subseteq$ **q_z**	**teu** $\subseteq$ **ttr** $\wedge$
ttv $\cap$ **teu** $= \emptyset \wedge$	**teu** $\subseteq$ **td** $\wedge$
zs $\subseteq$ **ze** $\wedge$	**teu** $\subseteq$ **t_z**$^{-1}$[**zs**] /* **(P 3.1) */**
ran(**q_z**) = **ze** /* **(P 3.2) */**	

INITIALISATION $t\_q, t\_z, to, ttr, ttv, teu, td, ti, qo, qtv, qd, qs, zs$:

$t\_q = \emptyset \wedge t\_z \in tt \leftrightarrow ze \wedge dom(t\_z) = tt \wedge to = \emptyset \wedge ttr = \emptyset \wedge ttv = \emptyset \wedge teu = \emptyset \wedge td = \emptyset \wedge$
$ti = \emptyset \wedge qo = \emptyset \wedge qtv = \emptyset \wedge qd = \emptyset \wedge qs = \emptyset \wedge zs = \emptyset$

EVENTS

UnsafeTrain $\widehat{=}\ldots$	UnsafePlatf $\widehat{=}\ldots$	SafeTrain $\widehat{=}\ldots$	SafePlatf $\widehat{=}\ldots$
StartPassExch $\widehat{=}\ldots$	EndPassExch $\widehat{=}\ldots$	LockedTrain $\widehat{=}\ldots$	UnlockedTrain $\widehat{=}\ldots$
ArrivalToPlatf $\widehat{=}\ldots$	DepartureFromPlatf $\widehat{=}\ldots$	OpenedTrain $\widehat{=}\ldots$	ClosedTrain $\widehat{=}\ldots$
OpenedPlatf $\widehat{=}\ldots$	ClosedPlatf $\widehat{=}\ldots$	SafeZone $\widehat{=}\ldots$	UnsafeZone $\widehat{=}\ldots$

StartEva $\widehat{=}$ /* *t*1 starts an evacuation */
ANY *t*1 WHERE $t1 \in ti \wedge t1 \in td \wedge$
$t1 \notin ttr \wedge t\_z[t1] \subseteq zs$
THEN $teu, ttr := teu \cup \{t1\}, ttr \cup \{t1\}$
END;

EndEva $\widehat{=}$ /* an evacuation is finished */
ANY *t*1 WHERE $t1 \in teu \wedge$
$t1 \notin to \wedge t1 \notin td$
THEN $teu, ttr := teu - \{t1\}, ttr - \{t1\}$
END;

Transfer2 $\widehat{=}$
ANY *t*1 WHERE $t1 \notin ttv \cup teu \wedge t1 \in ti$ /* *t*1 is not in transfer */
THEN $SKIP$ /* The transfer is finished */ END;

ArrivingZoneEva $\widehat{=}$ /* a train arrives in a zone */
ANY $t1, z1$ WHERE $t1 \in tt \wedge$
$t1 \notin ti \wedge z1 \notin t\_z[t1] \wedge z1 \in ze \wedge z1 \notin zs$
THEN $t\_z := t\_z \cup \{t1 \mapsto z1\}$ END;

LeavingZoneEva $\widehat{=}$ /* *t*1 leaves *z*1 */
ANY $t1, z1$ WHERE $t1 \in tt \wedge$
$t1 \notin ti \wedge z1 \in t\_z[t1] \wedge z1 \notin zs$
THEN $t\_z := t\_z - \{t1 \mapsto z1\}$ END;

SafeZoneEva $\widehat{=}$ /* *z*1 becomes safe */
ANY *z*1 WHERE $z1 \in ze \wedge z1 \notin zs$
THEN $zs := zs \cup \{z1\}$ END;

UnsafeZoneEva $\widehat{=}$ /* *z*1 becomes unsafe */
ANY *z*1 WHERE $z1 \in zs \wedge z1 \notin t\_z[td]$
THEN $zs := zs - \{z1\}$ END;

The other kind of passenger exchanges are introduced using a similar process.

2.6 Conclusion

What we obtain at this point is a formal description of the required behavior of the system. It takes the form of an event model.

This model describes an automaton representing the external behavior. Some safety properties specify the static of the system, the others specify the dynamic. Here, P1, P2 and P3 are static, P0 is dynamic. Static properties are embedded in the invariant. The final specification has a lot of events which have been gradually introduced, thanks to the event refinement mechanism. This top down analysis leads us to introduce system details, though the order of insertion between properties which have a similar abstraction level is more or less arbitrary. For instance, the introduction of emergency evacuations could be done before or at the same time as normal exchanges. Furthermore, the refinement process allows us to split the proofs of consistency of the final event system. Event Synchronization is distributed through the guards.

This application has been developed under AtelierB. Almost all the generated proof obligations have been automatically discharged.

However, we do not yet have the internal architecture of the system. The isolation of each component has not yet been done and, moreover, for each component, we do not have an implementation. In order to illustrate the whole process, we will present a second application which has been taken to the point of implementation.

3 Flight Warning System (FWS): The event model

3.1 Introduction

This case-study - the flight warning system (FWS) used in the airbus A340 aircraft- was proposed by the Aerospatiale Company. FWS's role is to monitor aeroplane subsystems. When an abnormal situation appears, FWS must decide on when, and how, to emit warning signals. One of the difficulties of this study comes from an imposed constraint on the final architecture: the system must be formed by two cyclic concurrent processes. The first one, `P1`, deals with examining all the alarms. If an alarm is detected "present", this process confirms it after a pre-defined period of time. If an alarm is detected 'absent', it removes it from the set of confirmed alarms. The second process, `P2`, has also to examine all the alarms. It is charged with emitting signals associated to the alarms which have been confirmed by the first process.

Since the two cycles are concurrent it is not possible to specify the application as being the following sequence: `P1 ; P2`. We have to allow for the fact that an alarm is examined by `P2` before `P1`, therefore, in this case, no signal will be emitted. Let us imagine one alarm a being activated at a time `t`. Six

cases are possible:

(i) `t ...P1 ...P2`: the alarm a 'happens' before its examination by `P1` and `P2`. The alarm is treated by `P1` during its running cycle and is treated by `P2`. A signal is emitted [3].

(ii) `t ...P2 ...P1`: the alarm a happens before its examination by `P1` and `P2`. The alarm is treated by `P2`. As a has not yet been examined by `P1`, no signal is emitted during the running cycle of `P2`. A signal for a will be emitted during the next cycle of `P2` except if `P1`, during its own next cycle, examines a before `P2` and detects that a is absent.

(iii) `P1 ...P2 ...t` or `P2 ...P1 ...t`: a will be treated by the two processes during their next cycles.

(iv) `P1 ...t ...P2`: same as situation 3.

(v) `P2 ...t ...P1`: during its running cycle, `P1` examines and confirms a. `P2` will treat a during its next cycle (unless `P1` detects absence of a).

3.2 The Event model

Here, we present a simplified version of the case study which does not cover details concerning the flying phases and the signal composition. A complete description is given in [Lop 96-1], [Lop 96-2].

The role of the event specification is to model the interactions between components and the environment. To do so, we need the set of alarms WW and the set of signals Ss, and the following variables:
wp: set of emitted alarms,
se: set of emitted signals,
wc: set of confirmed alarms,
$We1$, $Wp1$: alarms examined and detected *present* by `P1`
$We2$, $Wc2$ alarms examined and detected *confirmed* by `P2`.
$Num1$: counter. Number of executed cycles of `P1`.
$Num2$: counter. Number of executed cycles of `P2`.

With this model, we observe the following:
the beginning of presence (`NewWarning`) and the end of presence (`EndWarning`) of a warning situation (interaction with the environment),
the beginning of the emission (`EmittedSignal`) and the end (`EndSignal`) of a signal (interaction between `P2` and the environment),
the confirmation (`ConfirmWarning1`) and its end (`AbsentWarning1`) of an alarm (interaction between `P1` and `P2`),
the examination (`ExamineWarning1`) of an alarm by `P1` (interaction between `P1` and the environment),
the examination (`ExamineWarning2`) of an alarm by `P2` (interaction between

[3] In the real system the signal can be emitted but there is no guarantee of this as the aeroplane can be in a state where many alarms are activated, so there is a competition [Lop 96-1].

P2 and the environment),
the beginning of a new cycle of P1 (BeginCycle1),
the beginning of a new cycle of P2 (BeginCycle2).

This event model is built as explained in the previous section.

SYSTEM FWS2
SETS Ww, Ss
VARIABLES $wp, se, wc, Num1, Wp1, We1, Wc2, We2, Num2$
INVARIANT $wp, wc, Wp1, We1, Wc2, We2 \subseteq Ww \ \wedge se \subseteq Ss \ \wedge Num1, Num2 \in NAT$
INITIALIZATION wp, se, wc, $Num1$, $Wp1$, $We1$ $Wc2$, $We2$,$Num2$: $(wp = \emptyset \ \wedge se = \emptyset \ \wedge wc = \emptyset \wedge Num1 = 0$
$\wedge Wp1 = \emptyset \ \wedge We1 = \emptyset \ \wedge Wc2 = \emptyset \ \wedge We2 = Ww \ \wedge Num2 = 0)$
EVENTS

NewWarning $\widehat{=}$ /* A new alarm is présent */
ANY wx WHERE $wx \in Ww \ \wedge wx \notin wp$
THEN $wp := wp \cup \{wx\}$ END;

EmittedSignal $\widehat{=}$ /* A new signal is emitted */
ANY sx WHERE $sx \in Ss \ \wedge sx \notin se$
THEN $se := se \cup \{sx\}$ END;

ExamineWarning1 $\widehat{=}$ /* P1 is examining an alarm */
ANY wx WHERE $wx \in Ww \ \wedge wx \notin We1$
/* wx has not yet been examined */
THEN
IF $wx \in wp$ THEN /* The alarm is present */
$Wp1, We1 := Wp1 \cup \{wx\}, We1 \cup \{wx\}$
ELSE $We1 := We1 \cup \{wx\}$ END
END;

BeginCycle1 $\widehat{=}$ /* New cycle for P1 */
SELECT $We1 = Ww$
THEN $We1, Wp1, Num1 := \emptyset, \emptyset, Num1 + 1$
/* All the alarms will be examined */
END;

EndWarning $\widehat{=}$ /* End of an alarm */
ANY wx WHERE $wx \in wp$
THEN $wp := wp - \{wx\}$ END;

EndSignal $\widehat{=}$ /* Extinction of a Signal */
ANY sx WHERE $sx \in se$
THEN $se := se - \{sx\}$ END;

ConfirmWarning1 $\widehat{=}$ /* P1 confirms an alarm */
ANY wx WHERE $wx \in Ww \ \wedge wx \notin wc$
THEN $wc := wc \cup \{wx\}$ END;

AbsentWarning1 $\widehat{=}$ /* P1 indicates that wx is absent */
ANY wx WHERE $wx \in Ww$
THEN $wc := wc - \{wx\}$ END;

BeginCycle2 $\widehat{=}$ /* New cycle for P2 */
SELECT $We2 = Ww$
THEN $We2, Wc2, Num2 := \emptyset, \emptyset, Num2 + 1$
/* All the alarms will be examined */
END;

ExamineWarning2 $\widehat{=}$ /* P2 is examining an alarm */
ANY wx WHERE $wx \in Ww \ \wedge wx \notin We2$ /* wx has not yet been examined : $wx \notin We2$ */
THEN IF $wx \in wc$
THEN $Wc2, We2 := Wc2 \cup \{wx\}, We2 \cup \{wx\}$ /* The alarm is confirmed */
ELSE $We2 := We2 \cup \{wx\}$ END /* The alarm is not confirmed */
END;

Note that interactions between P1 and P2 are achieved via the variable wc which is modified by P1 and is accessed by P2.
Note also that the events BeginCycle1 and BeginCycle2 remove at once all the elements of respectively we_1, wp_1 and we_2, wc_2: they model the beginning of each cycle. As indicated in the guards ($we_1 = Ww$ and $we_2 = Ww$), these events can be activated only when all alarms have been examined by each process.

4 From event models to modules: Introduction

The event system FWS2 models the interactions of the system with the environment and between the two processes P1 and P2.
We will now illustrate, along with this example, the end of the process: going from this event system to the specification of each component and to the specification of the interface module.

In order to do so, we have first to transform the event system into a module which will be shared by the components. Its role is to provide each component with the operations it needs in order to interact with the environment: this is the interface module.
The specification of each component is not directly derived from the event model. It must be elaborated separately and it must contain the properties characterizing the component on its own. Don't forget that these properties have not been taken into account in the event model. Here component specifications can use some variables of the interface module. By doing so, we can relate local properties of the component to the expected behaviour of the whole system. This specification can later be refined in order to obtain its implementation.

We will now illustrate this part of the process step by step.
Modules and operations. A module specification $MS = (x, IM, A, O_i)$ is formed by a shared variable x that represents the module state, an invariant IM that expresses static properties of the module, the specification A that initializes the module and the set of operations O_i for $i \in 1..n$ that defines the dynamics. Each operation O_i takes the form of a *pre-post specification* $(P_i(x), Q_i(x, x'))$. In the syntax of the tool, it takes the form `Pre` $P(x)$ `THEN` `S` where `S` is a generalized substitution. As there is a translation of generalized substitution into before-after predicates [Abr 96-1], it corresponds to the usual pre-post formulation for program specification. Such modules are transition systems, whose initial states are defined by the initialization, and whose transitions are defined by the operations. From this semantic point of view, modules and event systems are equivalent.
Consistency of a module. A module is said to be consistent if the invariant is established by the initialization and preserved by each operation.
Refinements. A module M is refined by $M1$ if each operation of M is refined by the corresponding operation of $M1$ and if the initialization of M is refined by the initialization of $M1$. An operation, this is, a program specification (P, Q), is refined by another one (R, S) if for all program t, $\{R\}\ t\ \{S\}$ implies $\{P\}\ t\ \{Q\}$. These proof obligations are generated by the tools. There is a semantic difference between an operation and an event: refining a program specification allows to weak the pre-condition whereas refining an event allows to strength the guard.
Module implementations. Let M be a module (x, IM, A, O_i) . An implementation of M is a set of programs $t_0, t_1, ..., t_n$ such that:
$\{true\}\ t_0\{A \wedge IM\}$, and $\{IM \wedge P_i\}\ t_i\ \{IM \wedge Q_i\}$.
Those conditions plus the consistency of M, implies the following property:
For all $\mathcal{M}$ we have $\{true\}\ t_0; \mathcal{M}(t_1, \ldots, t_n)\{IM\}$ where $\mathcal{M}$ is a combination with if then else, while, sequence (;) and operations O_i. In other words, the invariant is satisfied by all module states.
Module importation. We can import a module to implement another one [Abr 96-1].

Let M be the same module as before and M_1 be (x, IM_1, A_1, V_i) (for $i \in 1..k$). Each V_i is of the form (R_i, S_i). To implement M_1 through an importation of M, we have:

(i) to provide for each V_i an expression $\mathcal{M}_i(O_1, ..., O_n)$ built with combinations of calls of O_i,

(ii) to prove that $V_i \sqsubseteq \mathcal{M}_i(O_1, ..., O_n)$.

If the imported module M has an implementation i.e. a set of programs $t_0, t_1, ..., t_n$ such that: $\{true\}\ t_0\{A \wedge IM\}$, and $\{IM \wedge P_i\}\ t_i\ \{IM \wedge Q_i\}$, the conditions above ensure that, for all $\mathcal{M}$:
$\{true\}\ t_0; \mathcal{M}(\mathcal{M}_1(t_1, \ldots, t_n), \ldots, \mathcal{M}_k(t_1, \ldots, t_n))\{IM_1\}$.

5 Towards an interface module: first step

5.1 Transformation of an event into an operation

An event system specification allows us to observe a complete system - a system which does not interact with the observer. So, a system specification models a closed universe. A program taking inputs and producing outputs is not a closed universe. The way to connect these two approaches is to consider the event system as a tool to *observe* the program behaviour. This is the case if each transition allowed by an operation is also a possible transition of the event which corresponds to this operation [Lop,Sim,Don 00], [Lop 02].
Then, to transform a system into a module, it suffices to transform each event:
$E \ =$ any a where $G(x, a)$ then $S_{x,a}$ into the operation
$[res \longleftarrow]op(a) \ =$ pre $G(x, a)$ then $S_{x,a}$.

We can freely add output parameters. It is obvious that in this case, the event system and its corresponding module define the same transition system.

More complex translations can be done between events and operations. In these cases, some proof obligations have to be generated and discharged. For more details, see [Lop 02], chapter 5.7.

5.2 Application to FWS

The module interface is obtained from FWS2 following the translation defined above.

MACHINE FWS_ENV
VARIABLES $wp, se, wc, Num1, Wp1, We1, Wc2, We2, Num2$
INVARIANT $wp, wc, Wp1, We1, Wc2, We2 \subseteq Ww \wedge se \subseteq Ss \wedge Num1, Num2 \in NAT$
INITIALIZATION $wp, se, wc, Num1, Wp1, We1\ Wc2, We2, Num2$:
$(wp = \emptyset \wedge se = \emptyset \wedge wc = \emptyset \wedge Num1 = 0 \wedge Wp1 = \emptyset \wedge We1 = \emptyset \wedge Wc2 = \emptyset \wedge We2 = Ww \wedge Num2 = 0)$
OPERATIONS

NewWarning$(wx) \triangleq$
PRE $wx \in Ww \wedge wx \notin wp$
THEN $wp := wp \cup \{wx\}$ END;

EmittedSignal$(sx) \triangleq$
PRE $sx \in Ss \wedge sx \notin se$
THEN $se := se \cup \{sx\}$ END;

EndWarning$(wx) \triangleq$
PRE $wx \in wp$
THEN $wp := wp - \{wx\}$ END;

ConfirmWarning1$(wx) \triangleq$
PRE $wx \in Ww \wedge wx \notin wc$
THEN $wc := wc \cup \{wx\}$ END;

EndSignal(sx) $\triangleq$
  PRE $sx \in Se$
  THEN $se := se - \{sx\}$ END;
$bres \longleftarrow$ ExamineWarning2(wx) $\triangleq$
  PRE $wx \in Ww \ \wedge wx \notin We2$
  THEN
    IF $wx \in wc$
    THEN $Wc2, We2, bres :=$
    $Wc2 \cup \{wx\}, We2 \cup \{wx\}, TRUE$
    ELSE $We2, bres :=$
      $We2 \cup \{wx\}, FALSE$ END
  END;
BeginCycle2 $\triangleq$
  PRE
    $We2 = Ww$
  THEN $We2, Wc2, Num2 :=$
  $\emptyset, \emptyset, Num2 + 1$ END

AbsentWarning1(wx) $\triangleq$
  PRE $wx \in Ww$
  THEN $wc := wc - \{wx\}$ END;
$bres \longleftarrow$ ExamineWarning1(wx) $\triangleq$
  PRE $wx \in Ww \ \wedge wx \notin We1$
  THEN
    IF $wx \in wp$
    THEN $Wp1, We1, bres :=$
    $Wp1 \cup \{wx\}, We1 \cup \{wx\}, TRUE$
    ELSE $We1, bres :=$
      $We1 \cup \{wx\}, FALSE$ END
  END;
BeginCycle1 $\triangleq$
  PRE
    $We1 = Ww$
  THEN $We1, Wp1, Num1 :=$
$\emptyset, \emptyset, Num1 + 1$ END;

6 Toward an interface module: second step

6.1 Logical and abstract variables

All module variables are used to express properties. Some of them model objects which will be effectively implemented by the program -they are named *abstract variables*, and have an operational content. Others are only logical -they are called *logical or auxiliary variables* [Aba,Lam 88]. Abstract variables are the only ones transformed during the module refinement process. They are implemented as well as data-refined. Logical variables will not appear in the final implementation.

Let M be a module which contains abstract variables (a) and logical variables (l). Let M_1 be the module obtained from M when all the expressions where logical variables appear, are eliminated. We want to take, as an implementation of M, any implementation of $M1$. This is possible if each operation O_M of M is of the following form:
$O_M = \ (p(a) \ \wedge \ pl(a,l) \ , \ q(a,a') \wedge ql(a,l,l'))$
and satisfies :
$\forall a, a', l. \ (p(a) \wedge pl(a,l) \wedge q(a,a') \ \rightarrow \ \exists l'. \ ql(a,l,l'))$.
Consider now the corresponding operation in M_1. It takes the form: $op_{M_1} = (p(a) \ , \ q(a,a'))$.

So we have:

(i) as soon as O_M can be activated, O_{M_1} can also be activated (the pre-condition has been constrained).

(ii) abstract variables a, a' do not depend on logical variables (but l may depend on a).

(iii) under the pre-condition of O_M, all the states which can be reached by O_{M_1} are also attainable by O_M. This means that, during the refinement process, whatever implementation choice is made which reduces the in-

ternal non-determinism of O_{M_1}, it will realize the post-condition of O_M.

These constraints ensure that any implementation of M_1 is also an implementation of M [Lop 02].

From this, we can roughly consider M as an overloading of M_1.

Logical variables allow to express logical properties, which involve notions not directly present in M. These variables also allow to constrain the use of operations of M_1.

6.2 Application to FWS

In the interface module `FWS_ENV`, abstract variables are those used to communicate with the environment. Other variables are logical:

VARIABLES wp, se, wc
LOGICAL VARIABLES $Num1$, $Wp1$, $We1$, $Wc2$, $We2$, $Num2$

The base module corresponding to `FWS_ENV` i.e. `FWS_ENV` without logical variables is:

MACHINE FWS_BASE
VARIABLES wp, se, wc
INITIALIZATION wp, se, wc: $wp = \emptyset \ \wedge se = \emptyset \ \wedge wc = \emptyset$
OPERATIONS
NewWarning(wx) $\widehat{=}$
  PRE $wx \in Ww \ \wedge wx \notin wp$
  THEN $wp := wp \cup \{wx\}$ END;
EmittedSignal(sx) $\widehat{=}$
  PRE $sx \in Ss \ \wedge sx \notin se$
  THEN $se := se \cup \{sx\}$ END;
EndSignal(sx) $\widehat{=}$
  PRE $sx \in Se$
  THEN $se := se - \{sx\}$ END;
$bres \longleftarrow$ ExamineWarning2(wx) $\widehat{=}$
  PRE $wx \in Ww$
  THEN
    IF $wx \in wc$
    THEN $bres := TRUE$
    ELSE $bres := FALSE$ END
  END;
EndWarning(wx) $\widehat{=}$
  PRE $wx \in wp$
  THEN $wp := wp - \{wx\}$ END;
ConfirmWarning1(wx) $\widehat{=}$
  PRE $wx \in Ww \ \wedge wx \notin wc$
  THEN $wc := wc \cup \{wx\}$ END;
AbsentWarning1(wx) $\widehat{=}$
  PRE $wx \in Ww$
  THEN $wc := wc - \{wx\}$ END;
$bres \longleftarrow$ ExamineWarning1(wx) $\widehat{=}$
  PRE $wx \in Ww$
  THEN
    IF $wx \in wp$
    THEN $bres := TRUE$
    ELSE $bres := FALSE$ END
  END;

This interface module is the effective interface module. `NewWarning` and `EndWarning` are implemented by the devices which detect warning situations (sensors and other electronic devices); `EmittedSignal` and `EndSignal` are implemented by output devices; `ExamineWarning1` is implemented by a mechanism provided by the environment to obtain inputs and so on...

Note that `BeginCycle1` and `BeginCycle2` have disappeared. They are, in fact, logical operations which formally express the necessity of examining all the alarms. They have a logical control role.

7 Component specification

7.1 Sharing

At this point, we have an interface module which provides the communication operations of the system. We have now to write the component specifications. In this application, we want:
1- to have the interface module E=(x, IE, A, V_i).
2- to model each component in a separate and independent module: $M_1 = (x, y, IM_1, A_1, W_i)$ and $M_2 = (x, z, IM_2, A_2, S_i)$. By independent, we mean that the consistency is proved independently for each module. It also means that each module has to be refined independently.
3- to force each module to be implemented using an import of the *same* interface module E. E becomes a shared module.
4- to preserve the correctness. By correctness, we mean that every interleaving of calls of programs realizing M_1 and M_2 operations (which modify interface variables through operation calls of E) establishes *both* the invariants of M_1 and M_2.
Formally: for every program t_{V_j} which realizes V_j and for every combination $\mathcal{N}_1$ of t_{V_j} and $\mathcal{N}_2$ of t_{V_j} we must have:
$\{true\}\ IE; A_1; A_2; (\mathcal{N}_1||\mathcal{N}_2)\ \{IM_1 \wedge IM_2\}$.
As E needs to be shared by M_1 and M_2, the correctness cannot be expressed locally on each component module.

This is clearly not true in general: as M_1 and M_2 can share variables of E, an operation of $M2$ can break the invariant of $M1$ and vice-versa.

To avoid this problem, one solution is the following:
• For each component M_j we must identify the set of operations O_j of E which are exclusively used by M_j. O_j is the set of operations that M_j will be allowed to use.
• Let R_j be the set of variables modified by O_j, $R_j \subseteq x$. R_j is the set of variables that can be used by M_j.

7.2 Application to FWS

We are able now to model the two processes `P1` and `P2`. We focus on `P2`, the model of `P1` can be made in the same way. First we have to extract the `FWS_ENV` operations which will only be used by the process `P2`. They are `EmittedSignal`, `EndSignal`, `ExamineWarning2`, `BeginCycle2`. So the variables of `P2` coming from `FWS_ENV` are se,$Wc2$,$We2$,$Num2$. It will also include its proper variables in order to express its own local properties. The proper variables are: wa, wni, $wnis$, sa, snc, they have been introduced to formalize the following local properties:
1. Confirmed warnings not inhibited by the current fly phase are activated warnings. $Wc2 \cap wni \subseteq wa$.
2. No cancelled signals, associated with activated warnings, are activated sig-

nals.
3. Activated warnings are confirmed warnings.
4. Activated signals are associated to activated warnings.
5. Emitted signals are activated signals.
6. The crew is always able to determine the warnings associated to emitted signals.

Another property is, that during each cycle, all the alarms must be examined. This is expressed by $We2 = Ww$. We use here logical variables. As the process `P2` is cyclic, the module includes only one operation `Cycle2` with a *true* pre-condition and which models a step of the cycle. All the properties of the module are expressed into the invariant - the post-condition of `Cycle2` just establishes the invariant and counts the number of steps.

MACHINE P2 Processus 2.
SETS Ww, Ss
CONSTANTS s_w, c_w
PROPERTIES $s\_w \in Ss \rightarrow Ww \ \wedge c\_w \in Cc \leftrightarrow Ww$
VARIABLES se, $Wc2$, $We2$, $Num2$, wa, wni, $wnis$, sa, snc
INVARIANT
$se \subseteq Ss \ \wedge Wc2 \subseteq Ww \ \wedge We2 \subseteq Ww \wedge \quad Num2 \in NAT \ \wedge wa \subseteq Wc2 \wedge$
$wni \subseteq Ww \ \wedge wnis \subseteq wa \ \wedge sa \subseteq Ss \wedge \quad se \subseteq sa \ \wedge snc \subseteq Ss \wedge$

$Wc2 \cap wni \subseteq wa \wedge$	**(Property 1.)**
$s\_w^{-1}[wa] \cap snc \subseteq sa \wedge$	**(Property 2.)**
$wa \subseteq Wc2 \wedge$	**(Property 3.)**
$s\_w[sa] \subseteq wa \wedge$	**(Property 4.)**
$se \subseteq sa \wedge$	**(Property 5.)**
$wnis \subseteq ran(c\_w) \wedge$	**(Property 6.)**
$We2 = Ww$	**communication property**

INITIALIZATION
se, $Wc2$, $We2$, $Cc2$, $Ce2$, $Pc2$, $Pe2$, $Num2$
wa, wni, $wnis$, sa, snc: $INVARIANT$
OPERATIONS
Cycle2 $\triangleq$
se, $Wc2$, $We2$, $Num2$, wa, wni, $wnis$, sa, snc:
$(INVARIANT \ \wedge Num2 = Num2\_0 + 1)$
END

Note that the variable $Num2$, which is incremented by each cycle, forces each implementation of `Cycle2` to behave correctly: if this increment does not occur, a refinement of `Cycle2` by `skip` would still be correct.
When $Num2$ is incremented, the following happens:
$Num2$ is a variable "coming" from `FWS_ENV`, hence each implementation of `Cycle2` will call `BeginCycle2` as it is the only operation of `FWS_ENV` which modifies $Num2$. Each implementation must reestablish the invariant, as it is the post-condition of `Cycle2`. And, in particular, it must establish $We2 = Ww$. This can only be done by as many calls to `ExamineWarning2`, as the number of alarms in Ww.

8 Conclusion

A first result of this work is to propose a method which takes into account the entire development of a system. This method has been used to model two industrial case studies. We have used the existing tools (in our case Atelier B) to formally prove a large part of the proof obligations generated by the method and which is actually supported by these tools. Almost all the generated proof obligations have been automatically discharged (around 85 %). The new notions : logical variables, shared modules and event-to-operation transformations, generate new proof obligations. They are not supported by the actual tools ; these proof obligations have been proved "by hand". The fact that it has been used on industrial cases, shows that the proposed method is suitable to treat large-scale systems. Secondly, what is interesting is that these new notions solve the problems met in previous works, when we tried to model applications of this kind with only the notion of module [Lop 96-1]. Another point of interest is that these extensions do not represent a profound modification to the existing theory. Hence, their incorporation into existing tools can be achieved.

Further work has to be done to improve (to refine) the conditions we have formulated in order to ensure the correctness of the whole process.

`Acknowledgements`: The authors want to express their gratitude to the persons which have made this work possible: J.R. Abrial, A. Burlureau, P.Desforges, the Matra team and Aerospatiale.

References

[Atelier B] ClearSy, http://www.atelierb.societe.com/index.html

[B Toolkit] B-Core http://www.b-core.com/btoolkit.html

[Aba,Lam 88] M. Abadi and L. Lamport, *The existence of refinement mappings*, Technical Report. Digital Systems Research Center, Palo Alto, California, 1988.

[Abr 96-1] Abrial J.R., *The B Book. Assigning programs to Meanings*, Cambridge University Press, 1996.

[Abr 96-2] Abrial J.R., *Extending B without changing it, for Developing Distributed Systems*, in First B conference, H.Habrias editor, 169-190, Nantes, 1996.

[Back 88] Back R.J., *A calculus of refinements for program derivations*, Acta Informatica 25, 593-624, 1988.

[Back,Kur 88] Back R.J.R., Kurki-Suonio R., *Distributed cooperation with action systems*, in ACM Transactions on Programming Languages and Systems, vol. 10, No. 4, pages 513-554, ACM, 1983.

[But 96] Butler M.J., *Stepwise refinement of communicating systems*, Science of Computer Programming 27, 139-173, 1996.

[Lam,Sha 90] Lam S.S., Shankar U., *A Relational Notation for State Transition Systems*, in IEEE Transactions on Software Engineering, Vol 16, No 7, 755-775, 1990.

[Lop 96-1] Lopez N., *Construction de la specification formelle d'un systeme complexe*, in First B conference, H.Habrias editor, 63-119, Nantes, 1996.

[Lop 96-2] Lopez N., *Construction de la specification formelle d'un systeme complexe*, Memoire d'ingenieur CNAM, 1996.

[Lop 99] Lopez N., *An 'event based B' industrial experience*, in the proceedings of the B user Group Meeting, edited by Ken Robinson, Applying B in an industrial context, World Congress on Formal Methods 1999.

[Lop,Sim,Don 00] Lopez N., Simonot M., Donzeau-Gouge V., *Deriving software specifications from event-based models*, in the proceedings of the 1st International Conference of B and Z users, edited by Jonathan P. Bowen, Steve Dunne, Andy Galloway, Steve King, Lecture Notes in Computer Science 1878, 209-229, Springer, 2000.

[Lop 02] Lopez N., *Spécification Formelle de Systèmes Complexes, Méthodes et Techniques"*, These CNAM, 2002.

[Morg 90] Morgan C., *Programming from Specifications*, Prentice-Hall International. 1990.

Automatic Verification of the IEEE-1394 Root Contention Protocol with KRONOS and PRISM ⋆

Conrado Daws [a,1], Marta Kwiatkowska [b], Gethin Norman [b]

[a] *CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands*
Conrado.Daws@cwi.nl

[b] *University of Birmingham, Birmingham B15 2TT, United Kingdom*
{M.Z.Kwiatkowska,G.Norman}@cs.bham.ac.uk

Abstract

We report on the automatic verification of timed probabilistic properties of the IEEE 1394 root contention protocol combining two existing tools: the real-time model-checker Kronos and the probabilistic model-checker Prism. The system is modelled as a probabilistic timed automaton. We first use Kronos to perform a symbolic forward reachability analysis to generate the set of states that are reachable with non-zero probability from the initial state, and before the deadline expires. We then encode this information as a Markov decision process to be analyzed with Prism. We apply this technique to compute the minimal probability of a leader being elected before a deadline, for different deadlines, and study the influence of using a biased coin on this minimal probability.

Key words: model checking, soft deadlines, probabilistic timed automata, IEEE 1394, root contention protocol

1 Introduction

The design and analysis of many hardware and software systems, such as embedded systems and monitoring equipment, requires detailed knowledge of their real-time aspects, in addition to the functional requirements. Typically, this is expressed in terms of *hard* real-time constraints; e.g. "after a fatal error, the system will be shut down in 45 seconds". In the case of safety-critical systems, it is essential to ensure that such constraints are *never* invalidated.

⋆ Supported in part by the EPSRC grant GR/N22960.
[1] Supported by an ERCIM Fellowship.

However, in other cases like multimedia protocols that perform in the presence of lossy media, such *hard deadlines* can be too restrictive. *Soft deadlines* are then a viable alternative in these cases. For example, a soft deadline of a multimedia system could be that "*with probability at least* 0.96, video frames arrive within 80 to 100 ms after being sent". Soft deadlines can also specify *fault-tolerance* and *reliability* properties such as "deadlock will not occur with probability 1", or "the message will be lost with probability at most 0.01".

Recent research [16,17] has set a theoretical framework for the specification and verification of timed probabilistic systems. Inspired by the success of real-time model-checkers such as Kronos [7] and Uppaal [18], the direction taken is that of automatic verification through *model checking*, adapting the formalisms and algorithms [1] for model-checking of timed systems to the case of timed probabilistic systems. Within this approach [2], timed probabilistic systems are modelled as *probabilistic timed automata* [16], i.e. timed automata with discrete probability distributions associated with the edges, and properties are specified in the logic PTCTL, which extends the quantitative branching temporal logic TCTL with a probabilistic operator. Due to the denseness of time, model checking algorithms rely on the construction of a finite quotient of the state space of the system, namely the region graph [16] or the forward reachability graph [17]. By adding the corresponding probability distributions to the transitions of the graph we obtain a Markov decision process (MDP). The probability with which a state of this MDP satisfies a property can then be calculated by solving an appropriate linear programming problem [6,5].

In this work we show how, based on these ideas, the real-time model-checker Kronos [7,12] and the probabilistic model-checker Prism [13,19] can be combined for the *automatic* verification of the root contention protocol of IEEE 1394, a timed and probabilistic protocol to resolve conflicts between two nodes competing in a leader election process. The property of interest is the minimal probability for electing a leader before a deadline. We first use Kronos [3] to perform a symbolic forward reachability analysis to generate the set of states that are reachable with non-zero probability from the initial state, and before the deadline expires. We then encode this information as a Markov decision process in the Prism input language. Finally, we compute with Prism the minimal probability of a leader being elected before a deadline, for different deadlines, and investigate the influence of using a biased coin on this minimal probability.

This article proceeds as follows. Section 2 introduces probabilistic timed automata and defines probabilistic reachability. In Section 3 we describe the features of Kronos and Prism used in our verification approach. The encoding of the reachability graph in Prism input language is explained in Section 4. Section 5 illustrates this approach with the verification of the root contention

[2] In this work we consider systems where only discrete probabilities arise.

[3] An experimental version, not yet distributed, that has been adapted to deal with probabilistic timed automata and generate the required output, is used.

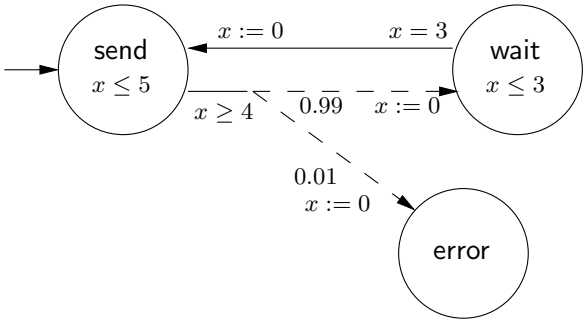


Fig. 1. An example of a probabilistic timed automaton PTA_1.

protocol of the IEEE 1394 standard. We conclude with Section 6.

2 Probabilistic Timed Automata

A timed automaton [2] is an automaton extended with *clocks*, variables with positive real values which increase uniformly with time. Clocks may be compared to positive integer time bounds to form *clock constraints* such as $(x \geq 2) \wedge (x \leq 5)$. There are two types of clock constraints: *invariants* labelling nodes, and *guards* labelling edges. The automaton may only stay in a node, letting time pass, if the clocks satisfy the invariant. When a guard is satisfied, the corresponding edge can be taken. Transitions are instantaneous, and can be labelled with *clock resets* of the form $x := 0$ meaning that upon entering the target node the value of clock x is set to 0. Probabilistic automata have probability distributions added to the edges, which model the likelihood of the action happening.

Example 2.1 *The probabilistic timed automaton* PTA_1 *of Figure 1 models a process which tries to* send *a packet after between 4 and 5 ms, and if successful* wait*s for 3 ms before trying to send another packet. The packet is sent with probability* 0.99 *and lost with probability 0.01 because of an* error*. Notice that edges belonging to a same distribution must be labelled with the same guard.*

2.1 Syntax

Clocks and valuations. Let the set $\mathcal{X}$ of *clocks* be a set of variables taking values from the time domain $\mathbf{R}_+$. A *clock valuation* is a point $v \in \mathbf{R}_+^{|\mathcal{X}|}$. The clock valuation $\mathbf{0} \in \mathbf{R}_+^{|\mathcal{X}|}$ assigns 0 to all clocks in $\mathcal{X}$. Let $v \in \mathbf{R}_+^{|\mathcal{X}|}$ be a clock valuation, $t \in \mathbf{R}_+$ be a time duration, and $X \subseteq \mathcal{X}$ a subset of clocks. Then $v + t$ denotes the *time increment* for v and t, and $v[X := 0]$ denotes the clock valuation obtained from $v \in \mathbf{R}_+^{|\mathcal{X}|}$ by resetting all of the clocks in X to 0 and leaving the values of all other clocks unchanged.

Zones. Let Z be the set of *zones* over $\mathcal{X}$, which are conjunctions of atomic constraints of the form $x \sim c$ and $x - y \sim c$, with $x, y \in \mathcal{X}$, $\sim \ \in \{<, \leq, \geq, >\}$, and $c \in \mathbf{N}$. A clock valuation v *satisfies* the zone ζ, written $v \models \zeta$, if and only

if ζ resolves to true after substituting each clock $x \in \mathcal{X}$ with the corresponding clock value $v(x)$. Let ζ be a zone and $X \subseteq \mathcal{X}$ be a subset of clocks. Then $\vec{\zeta}$ is the zone representing the set of clock valuations $v + t$ such that $v \models \zeta$ and $t \geq 0$, and $\zeta[X := 0]$ is the zone representing the set of clock valuations $v[X := 0]$ such that $v \models \zeta$.

Probability distributions. A *discrete probability distribution* over a finite set Q is a function $\mu : Q \to [0, 1]$ such that $\sum_{q \in Q} \mu(q) = 1$. Let $\mathsf{Dist}(Q)$ be the set of distributions over subsets of Q.

Definition 2.2 (Probabilistic timed automata.) *A* probabilistic timed automaton *is a tuple* $\mathsf{PTA} = (L, \mathcal{X}, \Sigma, I, \mathrm{P})$ *where:* L *is a finite set of* locations [4]*;* Σ *is a finite set of* labels*; the function* $I : L \to \mathrm{Z}$ *is the* invariant condition*; and the finite set* $\mathrm{P} \subseteq L \times \mathrm{Z} \times \Sigma \times \mathsf{Dist}(2^{\mathcal{X}} \times L)$ *is the* probabilistic edge relation*. An* edge *takes the form of a tuple* (l, g, X, l')*, where* l *is its source location,* g *is its* enabling condition*,* X *is the set of resetting clocks and* l' *is the destination location, such that* $(l, g, \sigma, p) \in \mathrm{P}$ *and* $p(X, l') > 0$*.*

2.2 Semantics

A state of a probabilistic timed automaton PTA is a pair (l, v) where $l \in L$ and $v \in \mathbf{R}_+^{|\mathcal{X}|}$ such that $v \models I(l)$. If the current state is (l, v), there is a nondeterministic choice of either letting *time pass* while satisfying the invariant condition $I(l)$, or making a *discrete* transition according to any probabilistic edge in P with source location l and whose enabling condition g is satisfied. If the probabilistic edge (l, g, σ, p) is chosen, then the probability of moving to the location l' and resetting to 0 all clocks in X is given by $p(X, l')$.

The semantics of probabilistic timed automata is defined in terms of transition systems exhibiting both nondeterministic and probabilistic choice, called probabilistic systems, which are essentially equivalent to Markov decision processes.

2.2.1 Probabilistic systems.

A *probabilistic system* $\mathsf{PS} = (S, Act, Steps)$ consists of a set S of *states*, a set Act of *actions*, and a *probabilistic transition relation* $Steps \subseteq S \times Act \times \mathsf{Dist}(S)$. A *probabilistic transition* $s \xrightarrow{a,\mu} s'$ is made from a state $s \in S$ by first nondeterministically selecting an action-distribution pair (a, μ) such that $(s, a, \mu) \in Steps$, and then by making a probabilistic choice of target state s' according to μ, such that $\mu(s') > 0$.

Definition 2.3 (Semantics of probabilistic timed automata.) *Given a probabilistic timed automaton* $\mathsf{PTA} = (L, \mathcal{X}, \Sigma, I, \mathrm{P})$*, the* semantics *of* PTA *is the probabilistic system* $[\![\mathsf{PTA}]\!] = (S, Act, Steps)$ *defined by the following.* (States) *Let* $S \subseteq L \times \mathbf{R}_+^{|\mathcal{X}|}$ *such that* $(l, v) \in S$ *if and only if* $v \models I(l)$*.*

[4] We sometimes identify an *initial location* $\bar{l} \in L$ represented graphically by an incoming arrow. In this case, the model starts in $\bar{l}$ with all clocks set to 0.

(Actions) *Let* $Act = \mathbf{R}_+ \cup \Sigma$. (Probabilistic transitions) *Let Steps be the least set of probabilistic transitions containing, for each state* $(l, v) \in S$*:*

Time transitions. *For each duration* $t \in \mathbf{R}_+$*, let* $((l, v), t, \mu) \in Steps$ *if and only if (1)* $\mu(l, v + t) = 1$*, and (2)* $v + t' \models I(l)$ *for all* $0 \leq t' \leq t$*.*

Discrete transitions. *For each probabilistic edge* $(l, g, \sigma, p) \in \mathrm{P}$*, let* $((l, v), \sigma, \mu) \in Steps$ *if and only if (1)* $v \models g$*, and (2) for each state* $(l', v') \in S$*:*

$$\mu(l', v') = \sum_{X \subseteq \mathcal{X} \,\&\, v' = v[X:=0]} p(X, l') \; .$$

2.3 Probabilistic Reachability

The behaviour of a probabilistic timed automaton is described in terms of the behaviour of its semantics, that is, the behaviour of a probabilistic system.

Paths. A *path* of a probabilistic system PS is a non-empty finite or infinite sequence of transitions $\omega = s_0 \xrightarrow{a_0,\mu_0} s_1 \xrightarrow{a_1,\mu_1} \cdots$. For a path ω and $i \in \mathbf{N}$, we denote by $\omega(i)$ the $(i+1)$th state of ω, and by $last(\omega)$ the last state of ω if ω is finite.

Adversaries. An *adversary* is a function A mapping every finite path ω to a pair $(a, \mu) \in Act \times \mathsf{Dist}(S)$ such that $(last(\omega), a, \mu) \in Steps$ [22]. Let Adv_{PS} be the set of adversaries of PS. For any $A \in Adv_{\mathsf{PS}}$, let $Path^A_{ful}$ denote the set of infinite paths associated with A. A probability measure $Prob^A$ over $Path^A_{ful}$ can then be defined following [11].

Definition 2.4 *Let* $\mathsf{PS} = (S, Act, Steps)$ *be a probabilistic system. Then the* reachability probability *with which a set* $F \subseteq S$ *of target states, can be reached from a state* $s \in S$*, for an adversary* $A \in Adv_{\mathsf{PS}}$*, is:*

$$ProbReach^A(s, F) \stackrel{\text{def}}{=} Prob^A\{\omega \in Path^A_{ful} \mid \omega(0) = s \,\&\, \exists i \in \mathbf{N} \,.\, \omega(i) \in F\} \; .$$

Furthermore, the maximal *and* minimal *reachability probabilities are defined respectively as*

$$MaxProbReach_{\mathsf{PS}}(s, F) \stackrel{\text{def}}{=} \sup_{A \in Adv_{\mathsf{PS}}} ProbReach^A(s, F)$$

$$MinProbReach_{\mathsf{PS}}(s, F) \stackrel{\text{def}}{=} \inf_{A \in Adv_{\mathsf{PS}}} ProbReach^A(s, F)$$

3 Verification with KRONOS and PRISM

Due to the denseness of time, the underlying semantic model of a (probabilistic) timed automaton is infinite, and hence effective decision procedures rely on building a finite quotient of the state space, e.g. the region graph or the forward reachability graph. This section describes the verification technique based on the generation of the forward reachability graph with Kronos, and

model checking the obtained graph encoded as a Markov decision process with Prism.

3.1 Forward Reachability with KRONOS

The forward reachability algorithm of Kronos proceeds by a graph-theoretic traversal of the reachable state space using a symbolic representation of sets of states, called *symbolic states* [8]. A symbolic state is a pair of the form $\langle l, \zeta \rangle$, with $l \in L$ and $\zeta \in \mathrm{Z}$, such that $\zeta \subseteq I(l)$; it represents all states (l, v) such that $v \models \zeta$. The traversal is based on the iteration of a *successor* operator in two alternating steps: first the computation of the *edge*-successors and then the computation of the *time*-successors of a symbolic state.

3.1.1 Edge Successors.

The edge-successor of $\langle l, \zeta \rangle$ with respect to an edge $e = (l, g, X, l')$ is

$$\mathsf{edge\_succ}(\langle l, \zeta \rangle, e) = \langle l', (\zeta \wedge g)[X := 0] \wedge I(l') \rangle$$

3.1.2 Time Successors.

The time-successor of $\langle l, \zeta \rangle$ is defined as

$$\mathsf{time\_succ}(\langle l, \zeta \rangle) = \langle l, \overrightarrow{\zeta} \wedge I(l) \rangle$$

Figure 2 shows the reachability graph obtained for the probabilistic timed automaton PTA_1 for a deadline of 15 ms, measured with an extra clock y. Since y is never reset, its value would increase indefinitely. To obtain a finite reachability graph, we need to apply the extrapolation abstraction of [8], which abstracts away the exact value of y when $y > 15$. Notice that this abstraction is exact with respect to reachability properties.

3.2 Model Checking Reachability Properties with PRISM

Prism [19] is a model checker designed to verify different types of probabilistic models: discrete-time Markov chains (DTMCs), Markov decision processes (MDPs) and continuous-time Markov chains (CTMCs). Properties to be checked are specified in probabilistic temporal logics, namely PCTL [6,5] if the model is a DTMC or an MDP, and CSL [4] in the case of a CTMC. We focus on the model checking of reachability properties on MDPs, since a (non-deterministic) probabilistic reachability graph belongs to this class of model, and deadline properties are specified as time bounded reachability properties.

3.2.1 Model Checking MDPs.

Model checking of Markov decision processes is based on the computation of the minimal probability $\mathsf{p}(\mathsf{s}, \Diamond \phi)$ or the maximal probability $\mathsf{P}(\mathsf{s}, \Diamond \phi)$ with which a state s satisfies a reachability formula $\Diamond \phi$. Then, a state s satisfies

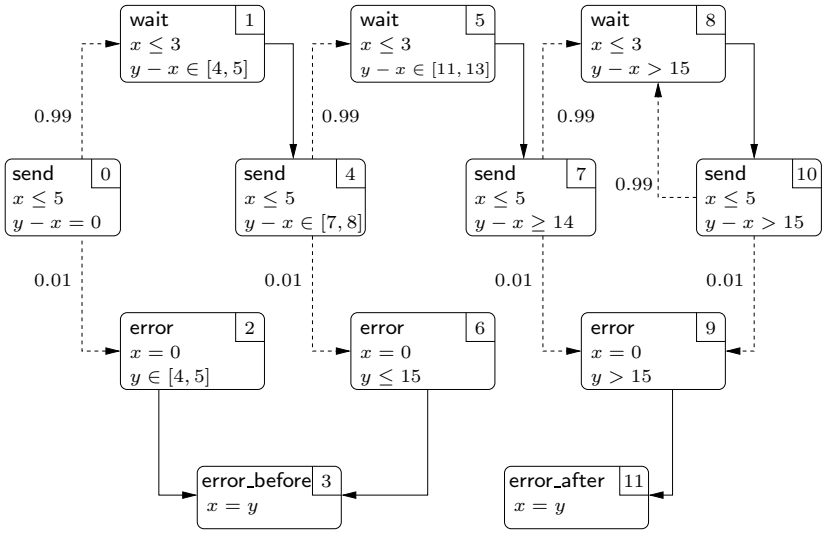


Fig. 2. Reachability graph of PTA_1

the PTCL formula $\mathcal{P}_{\leq\lambda}(\Diamond\phi)$ iff $\mathsf{P}(\mathsf{s}, \Diamond\,\phi) \leq \lambda$, and $\mathcal{P}_{\geq\lambda}(\Diamond\phi)$ iff $\mathsf{p}(\mathsf{s}, \Diamond\,\phi) \geq \lambda$. Maximal and minimal probabilities are computed by solving a linear programming problem [6,9]. The *iterative* algorithms implemented in Prism to solve this problem can combine different numerical computation methods with different data structures [13,14].

3.2.2 Model Checking PTAs.
We verify a PTA by model checking its probabilistic reachability graph using the following result [17]: the maximal probability computed on the reachability graph is an *upper bound* to the maximal probability defined on the semantic model of the probabilistic timed automaton. That is,

$$MaxProbReach_{\mathsf{PS}}(s, F) \leq \mathsf{P}(\mathsf{s}, \Diamond\,\phi_{\mathsf{F}}),$$

where ϕ_F is a formula characterizing the set of states F.

4 Encoding of a Reachability Graph in PRISM

The reachability graph obtained with Kronos is a list of symbolic states and transitions between them. In order to model-check probabilistic properties we must encode it as a Markov decision process using Prism's description language, a simple, state-based language, similar to Reactive Modules [3].

The behaviour of a system is described by a set of *guarded commands* of the form `[] <guard> -> <command>`. A *guard* is a predicate over variables of the system. A *command* describes a transition which the system can make if the guard is true, by giving a new value to *primed* variables as a function on the old values of *unprimed* variables. We consider two types of encoding of a reachability graph in this language.

4.1 Explicit Encoding

The first solution is a direct explicit encoding of the reachability graph using a single variable `s` whose value is the index of the state of the reachability graph. Transitions of the system are encoded by guarded commands such that the guard tests the value of `s` and the command updates it according to the transition relation of the reachability graph. For example, the encoding of the outgoing transitions from states $0, 4$ and 7, corresponding to location send in the reachability graph of Figure 2 is:

```
[] (s=0) -> 0.99:(s'=1) + 0.01:(s'=2)
[] (s=4) -> 0.99:(s'=5) + 0.01:(s'=6)
[] (s=7) -> 0.99:(s'=8) + 0.01:(s'=9)
```

This encoding generates a description the size of the reachability graph, which can grow drastically with the value of the deadline. Prism involves a model construction phase, during which the system description is parsed and converted into an MTBDD representation for further analysis. This phase can be extremely time consuming when the input file does not correspond to a modular and structured description of a system, such as with the explicit encoding. Thus, an encoding allowing for a more compact description of the system is needed.

4.2 Instances Encoding

States of a reachability graph correspond to several instances of locations of the timed automaton from which it was generated. We can then encode them with two variables, a location variable `l` and an instance variable `n` describing to which instance of the location it corresponds. For example, let $\texttt{l} = 0$ be the value of the location variable corresponding to send. Then, states $0, 4$ and 7 correspond to three different instances of this location, say $\texttt{n} = 0$, $\texttt{n} = 1$ and $\texttt{n} = 2$. Then, the outgoing transitions from states corresponding to send can be specified by:

```
[] (l=0)&(n=0) -> 0.99:(l'=1)&(n'=0) + 0.01:(l'=2)&(n'=0)
[] (l=0)&(n=1) -> 0.99:(l'=1)&(n'=1) + 0.01:(l'=2)&(n'=1)
[] (l=0)&(n=2) -> 0.99:(l'=1)&(n'=2) + 0.01:(l'=2)&(n'=2)
```

4.2.1 Relative compaction

The instance variable `n` is left unchanged by the command, meaning that the transition only affects the location variable for instances 0, 1 and 2. This is equivalent to write that $\texttt{n}' = \texttt{n}$, which can be omitted since, by default, a non updated variable takes its old value. Thus, the transitions above can be described more compactly in a single line:

```
[] (l=0)&(0<=n<=2) -> 0.99:(l'=1) + 0.01:(l'=2)
```

Since in a reachability graph a transition between two given locations can be repeated several times for different instances, this encoding allows us to specify them in a more compact manner. We will refer to this as the *relative compaction*, because it is based on specifying the updated value $\mathtt{n}'$ relative to its old value `n`.

The compaction algorithm is based on a traversal of the set of transitions of the reachability graph in order to find those which correspond to the same update command, and then describe them in a single line as a transition from multiple sates. Moreover, we combine different source states corresponding to the same location and successive numbers of instance, in a simple constraint where `n` is between two bounds, as in the example above.

4.2.2 Absolute compaction

In a reachability graph, we can encounter states which are the destination of many different transitions, such as state error_before ($\mathtt{l} = 3$, $\mathtt{n} = 0$) in the example of Figure 2. In this case, if we specify the updated value $\mathtt{n}'$ with its absolute value, an algorithm similar to the one above will allow us to describe all the incoming transitions in a single line. For example, the two transitions to error_before can be described by:

```
[] (l=2)&(0<=n<=1) -> 1:(l'=3)&(n'=0)
```

We will refer to this as the *absolute compaction*, because it is based on specifying the absolute value of $\mathtt{n}'$. Note that this compaction could also be applied to the explicit encoding. However, since in practise the relative compaction leads to a more compact description, compaction algorithms have only been implemented in the case of the instances encoding. Absolute compaction is especially interesting when used in combination with the relative one.

4.2.3 Combination

The heuristic implemented consists in first applying the relative compaction and then, for those transitions that couldn't be compacted, change the way the command updates the value of `n` from relative to absolute, and apply the absolute compaction.

5 Verification of the Root Contention Protocol

The IEEE 1394 High Performance serial bus is used to transport digitized video and audio signals within a network of multimedia systems and devices, such as TVs, PCs and VCRs. It has a scalable architecture, and it is hot-pluggable, meaning that devices can be added or removed from the network at any time, supports both isochronous and asynchronous communication and allows quick, reliable and inexpensive data transfer. It is currently one of the standard protocols for interconnecting multimedia equipment. The system uses a number of different protocols for different tasks, including a leader

election protocol, called *tree identify protocol*.

The tree identify protocol is a leader election protocol which takes place after a bus reset in the network, i.e. when a node (device or peripheral) is added to, or removed from, the network. After a bus reset, all nodes in the network have equal status, and know only to which nodes they are directly connected, so a leader must then be chosen. The aim of this protocol is to check whether the network topology is a tree and, if so, to construct a spanning tree over the network whose root is the leader elected by the protocol.

In order to elect a leader, nodes exchange "be my parent" requests with their neighbours. However, *contention* may arise when two nodes simultaneously send "be my parent" requests to each other. The solution adopted by the standard to overcome this conflict, called *root contention*, is both probabilistic and timed: each node will flip a coin in order to decide whether to wait for a short or for a long time for a request. The property of interest of the protocol is whether a leader is elected before a certain deadline, with a certain probability or greater.

5.1 The Model

The probabilistic timed automaton $\mathtt{I}_1^p$ in figure 3 is the abstract model of the root contention protocol considered in [15]. It is a probabilistic extension of the timed automaton $\mathtt{I}_1$ of [20] where each instance of bifurcating edges corresponds to a coin being flipped. For example, in the initial location start_start, there is a nondeterministic choice corresponding to node 1 (resp. node 2) starting the root contention protocol and flipping its coin, leading with probability 0.5 to each of slow_start and fast_start (resp. start_slow and start_fast). For simplicity, probability labels are omitted from the figure and probabilistic edges are represented by dashed arrows.

The timing constraints in $\mathtt{I}_1^p$ correspond to those specified in the updated standard IEEE 1394a. For instance, 360 ns corresponds to the transmission delay in the network if nodes are connected using a long wire. Other types of wire will have a different transmission delay, and hence can be verified by changing this value and re-running the experiments. Naturally, a lower delay results in a greater or equal probability of electing a leader before a deadline.

5.2 Verification

We first generate the reachability graph of the probabilistic timed automaton $\mathtt{I}_1^p$ until a deadline D is exceeded. To do this, we add an additional clock y, which measures the time elapsed since the beginning of the execution. Upon entering the location done, we test in *time zero* (clock x is reset in all incoming edges, and an invariant $x = 0$ forces the system to leave the location immediately) whether the deadline is exceeded or not, by adding two outgoing edges from done, one with the guard $y > D$ leading to a location done_after and one with the guard $y \leq D$ leading to a location done_before.

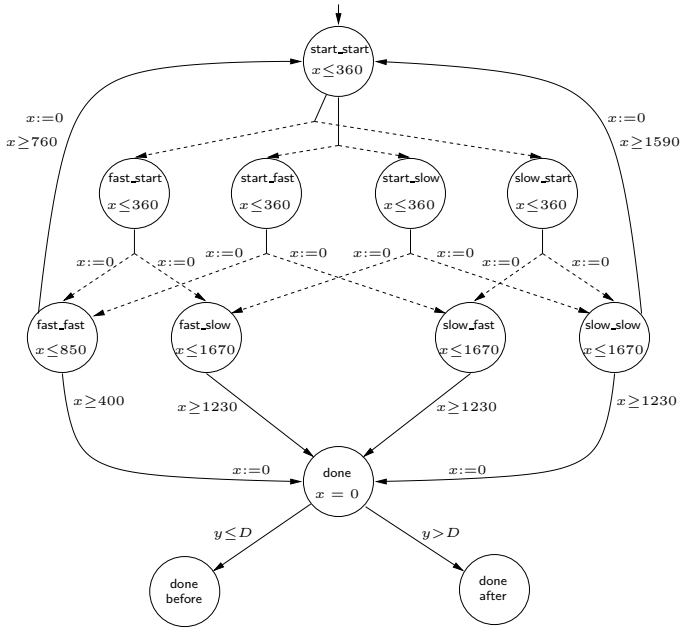


Fig. 3. Probabilistic timed automaton $\mathtt{I}_1^p$ modelling the root contention protocol.

Then, we specify the property of the root contention protocol we are interested in, namely, that a leader is elected *before* the deadline with *at least* a given probability. The PCTL formula that specifies this property is $\mathcal{P}_{\geq\lambda}(\Diamond\,(\mathsf{done} \wedge y \leq D))$, which cannot be verified with our technique because the probabilistic quantifier $\mathcal{P}_{\geq\lambda}$ is not of the correct form. However, it can be shown [15] that it is equivalent to the formula $\mathcal{P}_{<1-\lambda}(\Diamond\,\mathsf{done\_after})$ which can actually be verified on the reachability graph.

5.3 Experimental Results

In order to verify the property above, we compute the minimal probability for electing a leader before the deadline, for deadlines ranging from $4\mu s$ to $100\mu s$. These experiments were performed on a PC running Linux, with a 1400 MHz processor and 512 MB of RAM. Prism was used with its default options. Additional information can be found in [19].

Table 1 shows the results concerning the generation with Kronos of the reachability graph and of its encoding as an MDP. The first two columns give information about the generation of the reachability graph, its size in terms of the number of *states* and the *time* in seconds needed to generate it. The remaining columns show the size, in number of lines (i.e. transitions), of the MDP file generated by Kronos, for the different encodings we considered: explicit, instances with either absolute or relative compaction, and with both of them.

Figure 4 shows the evolution of the number of lines of the generated file for different values of the deadline. It demonstrates that the instances encoding allows for compactions which reduce drastically the number of lines of the

Table 1
Generation and encoding of the reachability graph

deadline	forw. reach.		explicit	instances	instances	instances
(μs)	states	time (s)		abs	rel	abs+rel
10	526	0.03	709	421	126	39
20	1876	0.09	2531	1501	368	72
30	4049	0.20	5466	3240	734	100
40	7034	0.46	9499	5629	1223	126
50	10865	1.23	14674	8694	1842	159
60	15511	2.74	20952	12412	2586	186
80	27296	8.94	36868	21841	4437	243
100	42401	22.29	57274	33926	6797	303

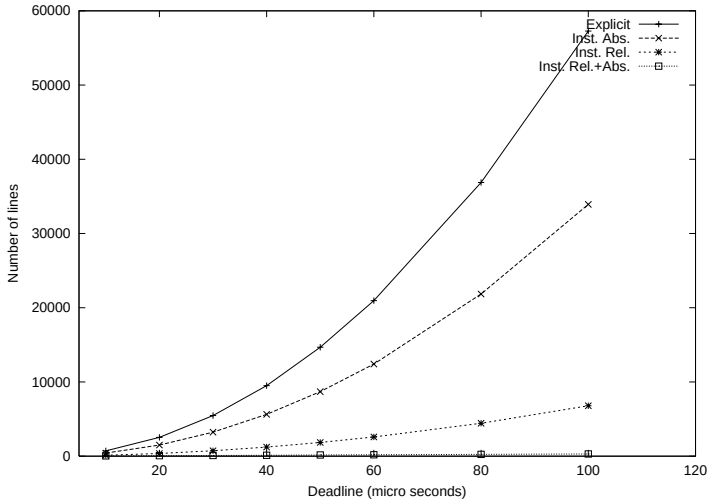


Fig. 4. Number of lines of the MDP

MDP file, and in the case where both relative and absolute compactions are considered, the number of lines grows less than linearly on the value of the deadline.

The experimental results concerning the verification with Prism are shown in Table 2. The left-most column shows the deadline used in the property, and the right-most column shows the minimum probability with which the system has reached a state where a leader is elected before the deadline. The results reflect the obvious fact that increasing the deadline increases the probability of a leader being elected. Notice that the same probability is computed for deadlines of more than $40\mu s$, meaning that the iterative method has converged, i.e. that the actual probabilities differ by less than $\epsilon = 10^{-6}$.

The remaining columns give information on the time performance of Prism in seconds, to build the model (columns labelled *model*) and to compute the probability (columns labelled *verif*), using the *explicit* encoding and the instances encoding with relative compaction (*inst+rel*) and with relative and absolute compaction (*inst+rel+abs*) .

Table 2
Time performances for model building and verification

deadline	explicit		inst+rel		inst+rel+abs		probability
(μs)	model (s)	verif (s)	model (s)	verif (s)	model (s)	verif (s)	
4	0.626	0.009	0.061	0.006	0.054	0.007	0.625
6	1.588	0.013	0.111	0.007	0.073	0.008	0.851562
8	5.654	0.018	0.195	0.008	0.140	0.008	0.939453
10	13.338	0.029	0.301	0.009	0.196	0.010	0.974731
20	190.971	0.098	3.038	0.025	1.303	0.026	0.999629
30	1037.892	0.309	14.672	0.056	4.969	0.058	0.999993
40	–	–	344.251	0.134	30.147	0.112	0.999998
50	–	–	1119.008	0.349	50.129	0.204	0.999998
60	–	–	3468.310	0.442	233.272	0.351	0.999998
80	–	–	–	–	814.035	0.729	0.999998
100	–	–	–	–	2861.889	1.744	0.999998

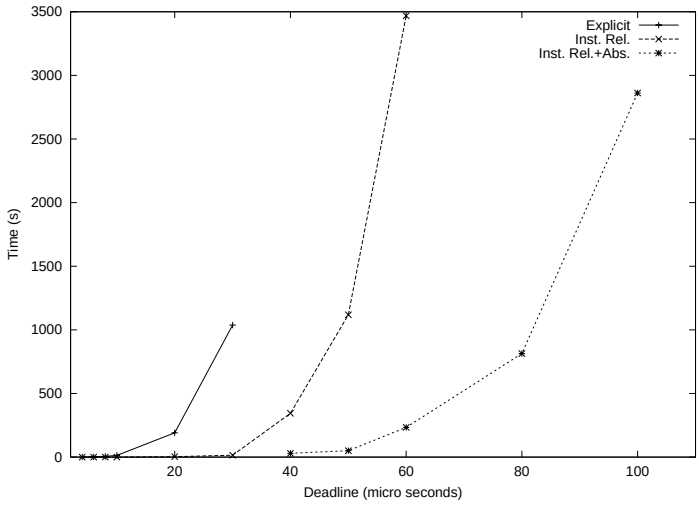


Fig. 5. Time to build the model

Compared to the previous attempt of verification [15] of the root contention protocol using HyTech [10], the generation of the reachability graph is no longer a problem, since it only took about 20 seconds to generate it for a deadline of $100\mu s$, whilst it took approximately 24 hours to generate it with HyTech for a deadline of $6\mu s$. Moreover, model checking of the probabilistic property took less than two seconds in the worst case. It is clear from this results that the bottleneck of this verification approach is now the model building phase of Prism, and the practical success of our verification approach depends on improving either the encoding or the model building algorithms.

Figure 5 shows the evolution of the time needed to build the model for different deadlines using different encodings. We can see that, although compactions improve the time needed to build the model, the latter still grows drastically with the value of the deadline, even when the size of the input file

Table 3
Probability of leader election with a biased coin.

fast	*slow*	$D = 6\mu s$	$D = 10\mu s$
.01	.99	0.039211	0.076886
.10	.90	0.330534	0.551770
.20	.80	0.554516	0.801000
.30	.70	0.704352	0.910950
.40	.60	0.799150	0.957090
.45	.55	0.830027	0.968230
.50	.50	0.851562	0.974731
.55	.45	0.864616	0.977771
.60	.40	0.869498	0.977795
.65	.35	0.865609	0.974558
.70	.30	0.850898	0.966919
.80	.20	0.768942	0.923030
.90	.10	0.544273	0.746829
.99	.01	0.076872	0.130600

grows linearly, because of the complexity of the guards after compaction.

5.4 RCP with a biased coin

We now study the influence of using a biased coin on the performance of the protocol. As noted in [21], a curious property of the protocol is that the probability for electing a leader before a deadline can be slightly increased if the probability to choose fast timing increases for both nodes. This is because, although the protocol will require more rounds to elect a leader, the time per round is lower when both processes select fast timing.

Table 3 gives the probability for electing a leader before $6\mu s$ or $10\mu s$, for different values of the probability of choosing *fast* or *slow* timing for both nodes. The results presented correspond to model checking the same property as before using the optimized instances encoding. Note that we don't need to compute the forward reachability for each case. Instead, since probabilities for choosing a fast or slow timings can be given as parameters in Prism description language, the same input file is used to perform probabilistic model checking, and only the actual values of the probabilities change.

6 Conclusions

We have presented an approach to the automatic verification of soft deadlines for timed probabilistic systems modelled as probabilistic timed automata. We use Kronos to generate the probabilistic reachability graph with respect to the deadline and encode it in the Prism input language. A probabilistic reachability property is then verified with Prism. We have successfully applied this

verification technique to the timed and probabilistic root contention protocol of the IEEE 1394. We have computed the minimal probability of electing a leader before different deadlines, and studied the influence of using a biased coin on this minimal probability.

The main obstacle we had to face was the encoding of the reachability graph in the Prism input language. The model checking algorithms of Prism are based on (MT)BDDs, so its input needs to be specified in a compact and structured manner. An explicit encoding of the reachability graph using a single variable to encode a state turned out to be infeasible even for small values of the deadline. The instances encoding using two variables, one corresponding to the location of the timed automaton, the other to the instance of this location in the reachability graph, allowed us to apply compaction techniques that helped overcoming this problem. However, it is not clear how general a solution this is. Finding a good encoding is then crucial.

Naturally, we need to validate this approach by applying it to other systems or protocols where timing and probabilistic aspects arise. In order to do this, a better integration of both tools is needed. A first step in this direction would be to implement the parallel composition of probabilistic timed automata so that we are able to model complex systems in a compositional way.

Acknowledgement

We thank Sergio Yovine for making Kronos' libraries available to us.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model checking continuous-time Markov chains by transient analysis. In *Proc. CAV'00*, LNCS, 2000.

[5] C. Baier and M. Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.

[6] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, *Proceedings of FSTTCS'95*, volume 1026 of *LNCS*, pages 499–513. Springer-Verlag, 1995.

[7] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 208–219. Springer-Verlag, 1996.

[8] C. Daws and S. Tripakis. Model–checking of real–time reachability properties using abstractions. In B. Steffen, editor, *Proc. TACAS'98*, volume 1384 of *LNCS*, pages 313–329. Springer-Verlag, 1998.

[9] L. de Alfaro. Computing minimum and maximum reachability times in probabilistic systems. In J. Baeten and S. Mauw, editors, *Proceedings of CONCUR'99*, volume 1664 of *LNCS*, pages 66–81. Springer Verlag, 1999.

[10] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1+2):110–122, 1997.

[11] J. G. Kemeny, J. L. Snell, and A. W. Knapp. *Denumerable Markov Chains*. Graduate Texts in Mathematics. Springer, 2nd edition, 1976.

[12] KRONOS web page. `http://www-verimag.imag.fr/TEMPORISE/kronos/`.

[13] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In J. B. T. Field, P. Harrison and U. Harder, editors, *Proc. TOOLS 2002*, volume 2324 of *LNCS*, pages 200–204. Springer, 2002.

[14] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. In J.-P. Katoen and P. Stevens, editors, *Proc. TACAS 2002*, volume 2280 of *LNCS*, pages 52–66. Springer, 2002.

[15] M. Kwiatkowska, G. Norman, and J. Sproston. Probabilistic model checking of deadline properties in the IEEE 1394 firewire root contention protocol. In *Proc. Int. Workshop on Application of Formal Methods to IEEE 1394 Standard*, 2001.

[16] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. volume 1601 of *LNCS*, pages 75–95. Springer-Verlag, 1999.

[17] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 286, 2002. To appear.

[18] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.

[19] PRISM web page. `http://www.cs.bham.ac.uk/~dxp/prism/`.

[20] D. Simons and M. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. *Springer International Journal of Software Tools for Technology Transfer*, 2001. To appear.

[21] M. Stoelinga. *Alea jacta est: verification of probabilistic, real-time and parametric systems*. PhD thesis, Univerisity of Nijmegen, 2002.

[22] M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *Proceedings FOCS'85*. IEEE Computer Society Press, 1985.

Wang Yi (Uppsala University, Sweden)

Synthesis of Verified Real Time Software

In this talk. I will present a unified model for timed systems to bridge scheduling, model checking and code synthesis. The technical contributions include a general notion of schedulability for automata and efficient algorithms for schedulability checking. TIMES is a tool developed based on these recent results and our past experience in developing the UPPAAL tool. It is designed for synthesis of verified software guaranteeing timing constraints. This talk will review recent development on TIMES and published results from papers.

References

Elena Fersman, Paul Pettersson, and Wang Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. Proceedings of TACAS 2002, LNCS 2280, pages 67-82

Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - A Tool for Modelling and Implementation of Embedded Systems. Proceedings of TACAS 2002, LNCS 2280, pages 460-464.

Specification and Analysis of the MPEG–2 Video Encoder with Timed-Arc Petri Nets [1]

Valentín Valero [2] Fernando L. Pelayo [3] Fernando Cuartero [4]
Diego Cazorla [5]

Departamento de Informática
Universidad de Castilla-La Mancha
Escuela Politécnica Superior de Albacete. 02071 - SPAIN

Abstract

Petri nets are a very suitable model for the description and analysis of concurrent systems. Several timed extensions of Petri nets have been defined to capture some additional aspects, concerning with the behaviour in time of the described systems. In this paper we illustrate the use of timed-arc Petri nets for the modelling of timed concurrent systems, using the MPEG–2 video encoder as an example. From the analysis of the model we conclude that the performance of the encoding process could be improved by introducing some minor changes on the encoder.

1 Introduction

Petri nets are a very suitable model for the description and analysis of concurrent systems. They have a graphical nature and they have a solid mathematical foundation supporting them. Furthermore, one of the main advantages of Petri nets is that they capture *true concurrency*, i.e., they are able to model the simultaneous execution of actions in the system. A Petri net consists of a set of places and transitions, as well as a set of arcs connecting places with transitions and transitions with places. Places and transitions are called the nodes of the net; then, when there is an arc connecting a pair of nodes (x, y), we say that x is a precondition of y and y is a postcondition of x. In order to capture the dynamic behaviour of the modelled system places are annotated

[1] This work has been supported by the CICYT project "Performance Evaluation of Distributed Systems",TIC2000-0701-C02-02.
[2] Email:valentin@info-ab.uclm.es
[3] Email:fpelayo@info-ab.uclm.es
[4] Email:fernando@info-ab.uclm.es
[5] Email:dcazorla@info-ab.uclm.es

with a number of *tokens*. Thus, markings are defined as functions from P to $\mathbb{N}$, where P is the set of places of the net.

However, Petri nets do not consider quantitative aspects in the specifications, like the time required for the execution of transitions. Then, in the last three decades, some effort has been made in order to include time in Petri nets. A survey of the different approaches to introduce time in Petri nets is presented in [5]. The first group of models assign time delays to transitions, either using a fixed and deterministic value [11,12] or choosing it from a probability distribution [3]. Other models use time intervals to establish the enabling times of transitions [10]. Some models introduce time on tokens [2,4,13]; here, tokens become classified into two different classes: available and unavailable ones. Available tokens are those that can be immediately used for firing a transition, while unavailable cannot. Occurrence of a transition has to wait for a certain period of time for these tokens to become available, although it is also possible for a token to remain unavailable forever (such tokens are said to be *dead*). More recently, Cerone and Maggiolo-Schettini [6] have defined a very general model (statically timed Petri nets), where timing constraints are intervals statically associated with places, transitions and arcs. Thus, models with timing constraints attached only to places, transitions or arcs can be obtained by considering particular subclasses of this general framework.

Timed-Arc Petri nets (TAPNs) [1,4,7,8,13] are a timed extension of Petri nets in which tokens have associated a non-negative real value indicating the elapsed time from its creation (*its age*), and arcs from places to transitions are also labelled by time intervals, which establish restrictions on the age of the tokens that can be used to fire the adjacent transitions. As a consequence of these restrictions some tokens may become *dead*, in the sense that they will never be available, since they are too old to fire any transitions in the future. The interpretation and use of Timed-Arc Petri nets can be obtained from a collection of processes interacting with one another according to a rendez-vous mechanism. Each process may execute either local actions or synchronization ones. Local actions are those that the process may execute without cooperation from another process, and thus in the Petri net model of the whole system they would appear as transitions with a single precondition place, while synchronization actions would have several precondition places, which correspond to the states at which every involved processes is ready to execute the action. Then, each time interval establishes some timing restrictions related to a particular process (for instance the time that a local processing may require). In consequence, the firing of a synchronization action can be done in a time window which depends on the age of the tokens on its precondition places.

Therefore, Timed-Arc Petri nets are a very appropriate model for the description of concurrent systems with time restrictions, such as manufacturing systems, real-time systems, process control and workflow systems. In this paper we show the use of TAPNs for the modelling of concurrent systems. As an illustration, we model the MPEG–2 Video Encoder by using TAPNs. The

ISO/IEC 13818–2 standard [9], commonly known as MPEG–2, is a standard intended for a wide range of applications, including Video–on–Demand (VoD), High Definition TV (HDTV) and video communications using broadband networks. The MPEG standards were designed with these two requirements:

- The need for a high compression, which is achieved by exploiting both spatial and temporal redundancies within an image sequence.
- The need for random access capability, which is obtained by considering a special kind of pictures (I pictures), which are encoded with no reference to other frames, only exploiting the spatial correlation in a frame.

Then, we have modelled this encoding algorithm with TAPNs, and from the analysis of the model we have concluded that a better performance could be obtained in the encoding process by introducing some minor changes into the encoder. Specifically, we have computed some bounds for the time required to encode each type of image of a video sequence, and we have concluded that some improvements can be introduced in the encoding process of the B-images.

The paper is structured as follows. In Section 2 we present Timed-Arc Petri nets and their semantics, in Section 3 we describe the MPEG–2 encoding algorithm and the corresponding TAPN that models it. Finally, the analysis of the algorithm and some conclusions are presented in Section 4.

2 Timed-Arc Petri Nets

We deal with timed-arc Petri nets, which have their tokens annotated with an age (a real value indicating the elapsed time from its creation) and arcs connecting places with transitions have a time interval associated with them. The interval limits the age of the tokens that are needed to fire the adjacent transition.

However, a transition is not forced to be fired when all its preconditions contain tokens with an adequate age, and the same is true even if these tokens are about to become too old for any transition to consume. More generally, in the model we consider[6] there is not any kind of urgency, what we can interpret in the sense that the model is *reactive*, as transitions will be only fired when the external context requires it. But then, it can be the case that the external context may lose the ability to fire a transition if some needed tokens become too old. Furthermore, it is possible that some tokens become *dead*, that is definitely useless because their increasing age will not allow the firing of any of their postcondition transitions in the future.

[6] Other proposals of timed-arc Petri nets [8] enforce the firing of transitions with an earliest and maximal firing rule.

Definition 2.1 Timed-arc Petri nets
We define a Timed-Arc Petri net (TAPN) as a tuple [7] $N = (P, T, F, times)$, where P is a finite set of *places*, T is a finite set of transitions ($P \cap T = \emptyset$), F is the *flow relation*, $F \subseteq (P \times T) \cup (T \times P)$, and *times* is a function that associates a closed time interval to each arc (p, t) in F, i.e.:

$$times : F \cap (P \times T) \longrightarrow \mathbb{R}_0^+ \times (\mathbb{R}_0^+ \cup \{\infty\})$$

When $times(p, t) = (x_1, x_2)$ we write $\pi_i(p, t)$ to denote x_i, for $i = 1, 2$. We will also say that $x \in times\,(p, t)$ if and only if $times(p, t) = (x_1, x_2)$ and $x_1 \leq x \leq x_2$.

As we previously mentioned, tokens are annotated with real values, so markings are defined by means of multisets over $\mathbb{R}_0^+$. More exactly, a marking M is a function:

$$M : P \longrightarrow \mathcal{B}(\mathbb{R}_0^+)$$

where $\mathcal{B}(\mathbb{R}_0^+)$ denotes the set of finite multisets (bags) of non-negative real numbers [8]. Thus, as usual, each place is annotated with a certain number of tokens, but each one of them has an associated non-negative real number (its *age*). We will denote the set of markings of N by $\mathcal{M}(N)$, and using classical set notation, we will denote the number of tokens on a place p at a marking M by $|M(p)|$.

As initial markings we only allow markings M such that for all p in P, and any $x > 0$ we have $M(p)(x) = 0$ (i.e., the *initial age* of any token is 0). Then, we define *marked Timed-Arc Petri nets* (MTAPN) as pairs (N, M), where N is a Timed-Arc Petri net, and M is an initial marking on it. As usual, from this initial marking we will obtain new markings, as the net evolves, either by firing transitions, or by time elapsing. In consequence, given a non-zero marking, even if we do not fire any transitions at all, starting from this marking we get an infinite reachability set of markings, due to the *token aging*.

A Timed-Arc Petri net with an arbitrary marking can be graphically represented by extending the usual representation of P/T nets with the corresponding time information. In particular we will use the age of each token to represent it. Therefore, MTAPNs have initially a finite collection of zero values labelling each place.

In Fig. 1 we show a MTAPN modelling a producer/consumer system, where we have represented by transition t_1 the action corresponding to the manufacturing process of the producer, which takes between 5 and 9 units of time, and by t_2 the action of including the generated object into the buffer. Notice that the initial tokens on p_5 represent the capacity of the buffer (3), and the

[7] We consider only arcs with weight 1 to simplify some definitions, but the extension to general arcs with greater weights is straightforward.
[8] Using classical notation, we consider that a multiset B over a set X is defined as a function $B : X \to \mathbb{N}$.

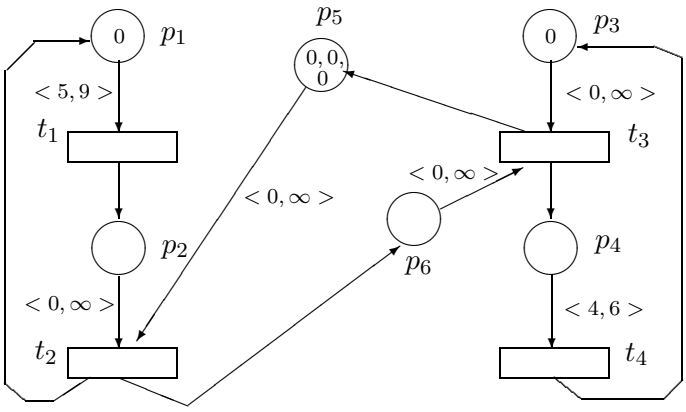


Fig. 1. Timed-Arc Petri net modelling a producer-consumer problem

arc connecting this place with t_2 is labelled by the interval $< 0, \infty >$, because these tokens can be consumed at any instant in the future. Tokens on p_6 represent the objects on the buffer which have not been yet consumed. Transition t_3 models the action of taking out an object from the buffer, which can occur at any instant. Finally, transition t_4 models the processing that makes the consumer for the objects extracted from the buffer, and this action takes between 4 and 6 units of time.

Let us observe that if the enabling time for the firing of one of these transitions (t_1 or t_4) expires, the system eventually becomes deadlocked, because we obtain a *dead* token either on p_1 or p_4.

Let us now see how we can fire transitions, and how we model the time elapsing.

Definition 2.2 Firing rule
Let $N = (P, T, F, times)$ be a TAPN, M a marking on it, and $t \in T$.

(i) We say that t is *enabled* at the marking M if and only if: $\forall p \in {}^\bullet t \ \exists x_p \in \mathbb{R}_0^+$ such that $M(p)(x_p) > 0 \wedge x_p \in times(p, t)$, i.e., on each precondition of t the marking has some token the age of which belongs to $times(p, t)$.

(ii) If t is enabled at M, it can be fired, and by its firing we reach a marking M', defined for each place p as follows:

$$M'(p) = M(p) - C^-(p, t) + C^+(t, p)$$

where both the subtraction and the addition operators work on multisets, and:

- $C^-(p,t) = \begin{cases} \{x_p\} & \text{if } p \in {}^\bullet t,\ x_p \in times(p,t) \text{ and } x_p \in M(p) \\ \emptyset & \text{otherwise} \end{cases}$
- $C^+(t,p) = \begin{cases} \emptyset & \text{if } p \notin t^\bullet \\ \{0\} & \text{otherwise} \end{cases}$

Thus, from each precondition place of t we remove a token fulfilling (i), and we add a new token with age 0 on each postcondition place of t.

As usual, we denote these evolutions by $M[t\rangle M'$, but it is noteworthy that these evolutions are in general non-deterministic, because when we fire a transition t, some of its precondition places could hold several tokens with different ages that could be used to fire it. Besides, we see that the firing of transitions does not consume any time. Therefore, in order to model the time elapsing we need the function *age*, defined below. By applying it, we age all the tokens of the net by the same time:

(iii) The function $age : \mathcal{M}(N) \times \mathbb{R}_0^+ \longrightarrow \mathcal{M}(N)$ is defined by:

$\forall M \in \mathcal{M}(N), \forall x, y \in \mathbb{R}_0^+, \forall p \in P :$

$$age(M,x)(p)(y) = \begin{cases} M(p)(y-x) & \text{if } y \geq x \\ 0 & \text{otherwise} \end{cases}$$

The marking obtained from M after x units of time without firing any transitions will be that given by $age(M,x)$.

Although we have defined the evolution by firing single transitions, this can be easily extended to the firing of *steps* or *bags* of transitions; those transitions that could be fired together in a single step could be also fired in sequence in any order, since no aging is produced by the firing of transitions. In this way we obtain step transitions that we denote by $M[R\rangle M'$. Finally, by alternating step transitions and time elapsing we can define a timed step semantics, where timed step sequences are those sequences $\sigma = M_0[R_1\rangle_{x_1} M_1 \ldots M_{n-1}[R_n\rangle_{x_n} M_n$, where M_i are markings, R_i multisets of transitions and $x_i \in \mathbb{R}_0^+$, in such a way that $M_i[R_{i+1}\rangle M'_{i+1}$ and $M_{i+1} = age(M'_{i+1}, x_{i+1})$. Note that we allow all the x_i be 0 in order to capture the execution in time zero of two causally related steps.

Then, given a MTAPN (N, M_0), we define $[M_0\rangle$ as the set of reachable markings on N starting from M_0, and we say that N is bounded if for every $p \in P$ there exists $n \in \mathbb{N}$ such that for all $M \in [M_0\rangle$ we have $|M(p)| \leq n$.

In a previous paper [13] we have shown that TAPNs have a greater expressiveness than PNs, even although TAPNs are not Turing complete, because they cannot correctly simulate a 2-counter machine. In that paper we proved that reachability is undecidable for TAPNs. Other properties that we have studied in a more recent paper [7] are coverability, boundedness and detection of dead tokens, which are all decidable for TAPNs. Decidability of coverability has been also proved in [1] for an extended version of TAPNs, in which all arcs can be annotated with bags of intervals in $\mathbb{N} \times \mathbb{N} \cup \{\infty\}$.

3 MPEG–2 Encoding Algorithm

MPEG standards were designed with two requirements in mind, namely, the need for a high compression, and the need for random access capability. These techniques exploit the fact that video sequences usually contain statistical re-

dundancies in both temporal and spatial directions. Thus, MPEG digital video coding techniques are statistical in nature. Specifically, the basic statistical property upon which MPEG compression techniques rely is inter-pixel region correlation. The contents of a particular pixel region can be predicted from nearby pixel regions within the same frame (intra-frame coding) or from pixel regions of a nearby frame (inter-frame coding).

Perhaps the ideal method for reducing temporal redundancies is one that tracks every pixel from frame to frame. However, this extensive search is computationally expensive. Under the MPEG standards, this search is performed by tracking the information within 16×16 pixels regions, called *macroblocks*. Given two contiguous frames, $frame(t)$ and $frame(t-1)$, for each macroblock in $frame(t)$, the encoder determines the best matching macroblock in $frame(t-1)$ and calculates the *motion vector*, which captures the macroblock translation information. Therefore, the *temporal redundancy reduction processor* generates a representation for $frame(t)$ by using the corresponding macroblock from $frame(t-1)$, and this representation only contains the motion vector and the prediction error (changes between both frames). This technique is called *motion compensated prediction*.

In order to reduce spatial redundancies a DCT (Discrete Cosine Transform) is used. With this coding process some subjective redundancies in the image are removed, on the basis of human visual criteria.

The combination of these two techniques described above are the key elements of the MPEG encoding process. Furthermore, in order to achieve the requirement of random access and high compression, the MPEG–2 standard specifies three types of compressed video frames: I pictures, P pictures and B pictures. I pictures (intracoded pictures) are coded with no reference to other frames, exploiting only spatial correlation in a frame. P pictures (predictive coded pictures) are coded by using motion compensated prediction of a previous I or P picture. Finally, B pictures (bidirectionally–predictive coded pictures) are obtained by motion compensation by using past and future reference frames (I or P pictures).

A group of consecutive I, P and B pictures constitute a structure called Group of Pictures (GoP). Therefore, a video sequence may be seen as a sequence of GoPs.

The block diagram of the MPEG encoder is depicted in Figure 2. In order to understand how the MPEG–2 encoder works, we will consider a typical GoP consisting on the frames IBBP. Despite the B pictures appearing before the P picture, the encoding order is IPBB because B pictures require both past and future frames as references.

The first frame in a GoP (I picture) is encoded in intra mode without references to any past or future frames. The DCT is applied to each macroblock and then it is uniformly quantized (Q). After quantization, it is encoded using a variable length code (VLC) and it is sent to the output buffer. At the same time the reconstruction (IQ) of all non-zero DCT coefficients belonging to one

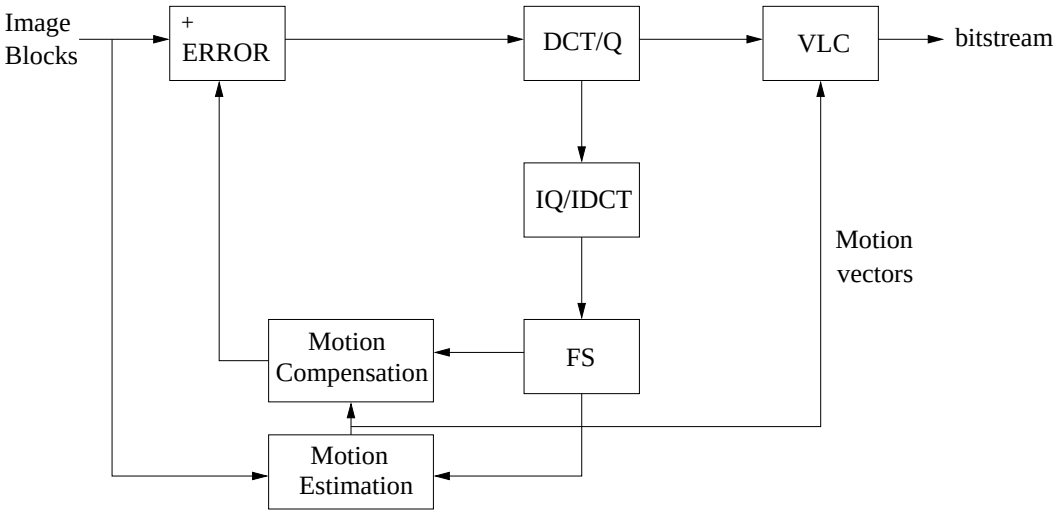


Fig. 2. Block diagram of the MPEG–2

macroblock and the Inverse DCT (IDCT) give us a compressed I picture which is stored temporarily in the *Frame Store (FS)*.

When the input is coded either as P or B pictures, the encoder does not code the picture macroblocks directly. Instead, it codes the prediction errors and the motion vectors. With P pictures, for each macroblock in the current picture, the motion estimation gives us the coordinates of the macroblock in the I picture that best matches its characteristics and thus, the motion vector may be calculated. The motion compensated prediction error is obtained by subtracting each pixel in a macroblock with its motion shifted counterpart in the previous frame. The prediction error and the motion vectors are coded (VLC) and sent to the output buffer. As in the previous case, a compressed P picture is stored in the Frame Store.

With B pictures, the motion estimation process is performed twice: for a past picture (I picture in this case), and for a future picture (P picture). Prediction errors and both motion vectors for each macroblock are coded (VLC) and sent to the output buffer. Notice that the compressed B pictures are not stored in the Frame Store, since they are not needed to calculate any other pictures.

3.1 Timed-Arc Petri Net modelling the MPEG–2

Figure 3 shows the Timed-Arc Petri net modelling the MPEG–2 encoding algorithm [9]. The left side figure describes the first part of the encoding algorithm, which corresponds to the generation of both I and P encoded pictures (but remember that even if P pictures are generated before the B pictures, they will appear the last in the final video sequence). Once the places *I-B1,*

[9] For simplicity, we omit in this picture the label of the arcs when they are labelled by $<0, \infty>$.

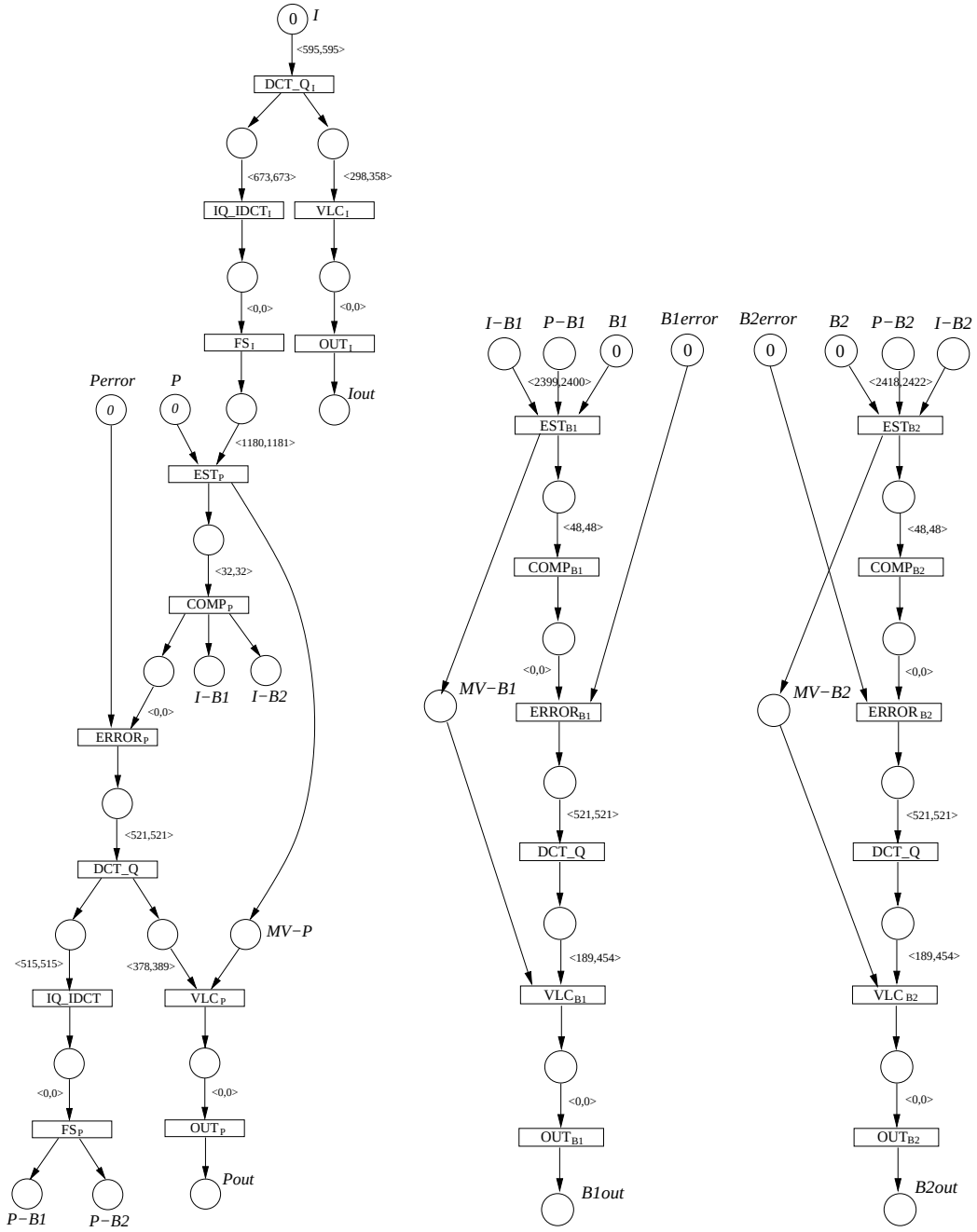


Fig. 3. TAPN modelling the MPEG–2

I-B2, P-B1, P-B2 are marked the second part of the net becomes activated (right part figure, where these places have been pictured again in that figure for a better readability), which models the *B1* and *B2* picture encoding process. *Iout* (respectively, *Pout*) represents the output of an encoded *I picture* (*P picture*), while places *B1out* and *B2out* represent the output of B1 and B2 pictures.

The time intervals that label the arcs connecting places with transitions have been obtained from several real measurements, by coding the so called

"Composed" video sequence. This experiment has been repeated a number of times, and the results being reported below are the minimum and the maximum of all these trials [10] . During these trials, no other operations were taking place in our experimental setup. The "Composed" video sequence (format PAL CCIR601, 720x576 pixels) is a representative video sequence which has several different motion levels, and we have encoded it by using a completely software-based MPEG–2 video encoder derived from that developed in Berkeley, which is freely available in the MPEG Home Page: http://www.mpeg.org.

In order to get the real values for the different elements of the encoder we have included some patches into the source code, which correspond with the beginning and the end of the elementary actions that we have described in the specification of the algorithm.

The real values thus obtained for I and P pictures reaching the output buffer are shown in Table 1 (measured values), as well as the times for encoding the complete GoP.

Table 1
Times for encoding the pictures

	Measured		TAPN	
Picture	Min	Max	Min	Max
I	893 ms	953 ms	893ms	953 ms
P	3688 ms	3738ms	3379ms	3391ms
GoP	11567 ms	12085 ms	6673ms	6962ms

4 Analysis and Conclusions

We may compute the times to reach every place of a TAPN, by constructing a state reachability graph. This graph can be constructed for bounded Timed-Arc Petri nets (see [13] for a detailed description); this construction is based on the fact that for every place p of a Timed-Arc Petri net we can find a maximal value $Max(p)$ for the age of its tokens to influence the activation of its postcondition transitions, because the tokens on that place with an age exceeding that maximal value can only be consumed by the firing of some transition $t \in p^\bullet$ for which $\pi_2(p,t) = \infty$.

Concretely, we may define:

$$Max(p) = Max\{\pi_i(p,t) \,|\, t \in p^\bullet,\ \pi_i(p,t) < \infty,\ i = 1,2\}$$

Obviously, in order to fire such a transition t the age of the involved token on p is unimportant once it exceeds $Max(p)$. This means that in order to construct

[10] These values were obtained in a Pentium II - 350MHz platform with 64MB RAM.

the state graph we can represent by the single value $1 + Max(p)$ the whole interval $[1 + Max(p), \infty]$.

In our case, with that state graph we may obtain a time interval for reaching every place of the TAPN, concretely we have obtained the results shown in the last two columns of Table 1 (TAPN values). Notice that the time required to reach the place *Iout* (respectively, *Pout*) is the time required to generate the I (respectively, P) picture. Furthermore, the time required for encoding the complete GoP is the time required to reach both places *B1out* and *B2out*.

We can observe that the most significant differences with the measured values appear in the times obtained for completing the GoP encoding. These differences are due to the important fact that the encoder only uses a single processor, whereas our TAPN model captures all the intrinsic parallelism of the encoding process, so that with the analysis of the TAPN we are obtaining the required times provided that we have as many processors as needed to take advantage of this parallelism. In our case two processors would be enough.

Consequently, the main conclusion of this analysis of the MPEG–2 encoder by using TAPNs is that the performance of the encoding algorithm can be improved by a factor of nearly 50% by using two processors. Moreover, in bigger GoPs this improvement factor increases a little more, since the major cost of the encoding process is due to the B-images, which can be encoded in parallel.

Acknowledgement

The authors would like to thank the reviewers for their comments and suggestions, which have helped to improve this paper significantly.

References

[1] P.A. Abdulla and A. Nylén. *Timed Petri Nets and BQOs*. In Proc. ICATPN 2001, Lecture Notes in Computer Science, vol. **2075**(2001), pp. 53–70.

[2] W.M.P. van der Aalst. *Interval Timed Coloured Petri Nets and their Analysis*. Lecture Notes in Computer Science, vol. **691** (1993), pp. 451–472.

[3] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte and A. Cumani. *On Petri Nets with Stochastic Timing*. Proc. of the International Workshop on Timed Petri Nets, IEEE Computer Society Press, pp. 80–87. 1985.

[4] T. Bolognesi, F. Lucidi and S. Trigila. *From Timed Petri Nets to Timed LOTOS*. Proc. of the Tenth International IFIP WG6.1 Symp. on Protocol Specification, Testing and Verification. 1990.

[5] Fred D.J. Bowden. *Modelling time in Petri nets*. Proc. Second Australia-Japan Workshop on Stochastic Models. 1996.

[6] Antonio Cerone and Andrea Maggiolo-Schettini. *Time-based expressivity of time Petri nets for system specification*. Theoretical Computer Science, vol. **216**(1999), pp. 1-53.

[7] D. de Frutos, V. Valero and O. Marroquín. *Decidability of Properties of Timed-Arc Petri Nets*. Proc. ICATPN 2000, Lecture Notes in Computer Science, vol. **1825** (2000), pp. 187–206.

[8] Hans-Michael Hanisch. *Analysis of Place/Transition Nets with Timed-Arcs and its Application to Batch Process Control*. Application and Theory of Petri Nets, LNCS vol. **691** (1993), pp:282–299.

[9] ISO/IEC 13818-2. *Draft International Standard Generic Coding of Moving Pictures and Associated Audio*. Recommendation H.262.

[10] P. Merlin. "A Study of the Recoverability of Communication Protocols". PhD. Thesis, Univ. of California. 1974.

[11] C. Ramchandani. "Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets". PhD. Thesis, Massachusetts Institute of Technology, Cambridge. 1973.

[12] J. Sifakis. *Use of Petri Nets for Performance Evaluation*. Proc. of the Third International Symposium IFIP W.G.7.3., Measuring, Modelling and Evaluating Computer Systems. Elsevier Science Publishers, pp. 75–93. 1977.

[13] V. Valero, D. de Frutos and F. Cuartero. *On Non-decidability of Reachability for Timed-Arc Petri Nets*. Proc. 8th Workshop on Petri Nets and Performance Models, PNPM'99, pp. 188–196. 1999.

Properties of the subtraction valid for any floating point system

Sylvie Boldo and Marc Daumas

Laboratoire de l'Informatique du Paralllisme
UMR 5668 - CNRS - ENS de Lyon - INRIA

1 Introduction

In 1974, Sterbenz [20] presented a theorem about the exact subtraction of two floating point numbers x and y when they are very close one from another, that is

$$\frac{y}{2} \leq x \leq 2y.$$

The theorem stating that $x - y$ is exact under the preceding condition was presented for any radix provided the hardware was accurate enough. More recently, other authors [8,10] presented similar results with an emphasis on didactic aspects.

We have recognized in [6] that Sterbenz's theorem is not a property of the computing hardware but rather a property of the floating point number representation. Given x and y, the question is to know whether or not $x - y$ can be represented in the working floating point system. This is clearly the key necessary condition for the implemented floating point subtraction $x \ominus y$ to return the exact result $x - y$.

With IEEE-like behavior, any floating point operation is cut down to two steps. An intermediate result is first computed to sufficient accuracy and then rounded. The designer must guarantee that the system always returns the result as if the infinitely precise mathematical operation were rounded. For example the subtraction is implemented as the composition of two mathematical functions, namely, the subtraction $(-)$ and the user specified rounding function $(\circ)$

$$x \ominus y = \circ(x - y).$$

The details of the implementation are not relevant to the user since knowing the rounding function is sufficient to deduce the value returned by any operation. Users usually expect the rounding function to be a monotonous (non decreasing) projection of the real numbers over the set of the machine floating point numbers. The later property implies that for any floating point number v,

$$\circ(v) = v.$$

Establishing Sterbenz's equality does not require any additional knowledge on the rounding function provided it is a projection.

In Section 2, we describe more precisely our formalization of the floating-point system. In Section 3 we discuss key properties of this system. The proposed representation is very redundant but we will see that any machine number has one single canonical representation that can be used in hardware. We will quickly present that a floating point system must handle denormal numbers in order to verify Sterbenz's theorem on very small numbers. Finally as the existence of a negation will become key to Sterbenz's theorem, we will present a strong necessary and sufficient condition for a generic number system to be stable by negation.

Section 4 presents our results about Sterbenz's theorem and relates them to real hardware implementations. We have focused on IEEE 754 standard implementations and on Texas Instrument TMS 320C3x series described in Section 5 with its SMJ military grade processes. The SMJ 320C3x circuits can be used in avionics and military applications such as the flight control primary or secondary computer (FCPC / FCSC) [14]. Past studies have proved that an automatic proof checker must be used for such critical systems [16]. This work ends with concluding remarks and perspectives for further developments.

2 Our formalization of a floating point system

All the results have been developed and validated using Coq [11]. It is a theorem checking system based on the Curry-Howard isomorphism. Systems like Coq allow the user to define new objects and to derive consequences of these definitions formally while checking every detail. The Coq tool is based on higher-order logic. With such an expressive logic, it is possible to state properties in their most general form. For example, universal quantification has been used to state properties that are true for an arbitrary rounding mode. Theorem provers have already been successfully used to mechanically check the correctness of floating-point algorithms [17,9], and with a strong emphasis for avionics [2].

We will present in this text the behavior of a generic floating point system in regard to Sterbenz's theorem. We define a generic floating point system from a mapping of $\mathbb{Z}^2$ onto $\mathbb{R}$

$$(n, e) \hookrightarrow n\beta^e$$

where β is a constant integer strictly greater than one called the radix of the floating point system. Later n will be called the mantissa and e the amplitude.

In Coq, the set is defined by the `float` type defined below in ASCII

```
Record float : Set := Float {
  Fnum: Z;
  Fexp: Z }.
```

and its value is obtained by using the `FtoR` function

```
Definition FtoR := [x : float]
   (Rmult (Fnum x) (powerRZ (IZR radix) (Fexp x))).
```

Two pairs are **equivalent** if they are mapped to the same real value. This equality will be noted as $=_{\mathbb{R}}$. Coq files are hardly understandable for a non-Coq user, theorems and definitions can be presented using a integrated pretty printer. For example:

Definition 2.1 FtoR := x : float $\longmapsto$ $\mathrm{Fnum}(x) \times \beta^{\mathrm{Fexp}(x)}$

All the quantities treated by a computer system must fit into a finite field, we focus our interest on pairs (n, e) such that n and e are bounded. For practical reasons, we do not use an upper bound on the amplitude and a **bounded floating point pair** is such that

$$n \in \{-N_i, \cdots, N_s\} \qquad \text{and} \qquad e \geq -E_i.$$

That is to say in Coq:

Definition 2.2 FboundedI := b : FboundI, d : float $\longmapsto$
$(-\mathrm{vNumInf}(b) \leq \mathrm{Fnum}(d)) \bigwedge (\mathrm{Fnum}(d) \leq \mathrm{vNumSup}(b))$
$\bigwedge (-\mathrm{dExp}(b) \leq \mathrm{Fexp}(d))$

A sectioning mechanism with implicit parameter management transforms Sterbenz's theorem with our floating point library so that it states that

> "*for any radix greater than one, for any floating-point system, for all floats x and y, if x and y are bounded, and if $\frac{y}{2} \leq x$ and $x \leq 2 \times y$ then there exists a bounded float z such that $z =_{\mathbb{R}} x - y$*".

Unfortunately, this assertion is false. For example, let the radix be two and the format such that the mantissa is between -1100_2 and 1111_2. Let x be $(1111_2, 0)$ and y be $(1110_2, 1)$. Both x and y are bounded, they are such that $\frac{y}{2} \leq x \leq 2 \times y$ but $x - y$ is -1101_2 and this value cannot be represented exactly in this floating point system. We will later give a list of necessary conditions for the assertion to be true.

Proofs are built interactively using high-level tactics that may solve some of the "easy" subgoals. We used pcoq [1]: a working environment for the Coq theorem prover with a nice graphical interface and the pretty printer.

At the end of each proof, Coq records a proof object that contains all the details of the derivation and ensures that the theorem is valid. The object can be double checked for life critical applications by a tool such as BindLib, a program independent of the Coq development.

The proofs for this work can be downloaded through the Internet at the address

`http://www.ens-lyon.fr/~sboldo/coq`.

They include the current development of the floating point library available at

`http://www-sop.inria.fr/lemme/AOC/`.

All the following theorems have been proved using this very general formalization. Unless explicitly specified the properties hold for any radix greater than one and any bound on the mantissa and the amplitude.

3 Basic properties of the set of bounded floating point numbers

3.1 Multiple representations

Contrary to IEEE-like behavior, the proposed library defines possibly many bounded floating point pairs with the same value. For example, the three radix two floating point pairs $(1100_2, 4)_2$, $(110_2, 5)_2$ and $(11_2, 6)_2$ share the same real value $3 \times 2^6 = 192$. This fact can be disturbing as one real value can be associated to many different bounded floating point pairs that do not have the same properties.

In order to retain common floating point behavior, we define a canonical pair for each bounded pair. This pair is meant to represent the actual fields stored in a computer that are associated to the number. A pair is **normal** if it is bounded and its amplitude cannot be reduced by multiplying the mantissa by the radix, that is

$$n \times \beta \notin \{-N_i, \cdots, N_s\}.$$

A pair is **denormal** if it is bounded and the amplitude reduction is blocked by the fact that it uses already the minimal accepted amplitude despite the mantissa being small enough to be multiplied by the radix. That is

$$n \times \beta \in \{-N_i, \cdots, N_s\} \qquad \text{and} \qquad e = -E_i.$$

Any bounded pair is equivalent to one unique pair either normal or denormal. The later pair is called the **canonical** representation. This fact is proved by several theorems. The first one, `FcanonicIUnique` states that if p and q are two canonical floating-point numbers such that $p =_{\mathbb{R}} q$ then p and q are syntactically equal (Leibniz's equality).

Other theorems prove the correctness of the `FnormalizeI` function defined below to construct the canonical representation from any bounded representation:

```
Fixpoint FNIAux [v, N, q : nat] : nat :=  Cases q of
 0 => 0
|(S q') => Cases
  (Zcompare (Zmult (Zpower_nat radix q') v) (Zmult radix N)) of
     INFERIEUR => q' | EGAL => q' | _ => (FNIAux v N q') end
```

```
 end.

Definition FNI := [q, N : nat]  (pred (FNIAux q N (S (S N)))).

Definition FnormalizeI :=
   [b : FboundI] [p : float]
    Cases (Zcompare ZERO (Fnum p)) of
      EGAL => (Float ZERO (Zopp (dExp b)))
     | INFERIEUR => (Fshift radix (min
           (FNI (absolu (Fnum p)) (vNumSup b))
           (absolu (Zplus (Fexp p) (dExp b)))) p)
     | SUPERIEUR => (Fshift radix (min
           (FNI (absolu (Fnum p)) (vNumInf b))
           (absolu (Zplus (Fexp p) (dExp b)))) p)
    end.
```

Expressing that the function is correct means that (i) if p is a bounded float, then the result FnormalizeI(p) is a bounded float (`FnormalizeIBounded`). It also means that (ii) the result is canonical (`FnormalizeIFcanonicI`) and such that (iii) the input pair p and the result pair are mapped to the same real value that is to say $p =_{\mathbb{R}}$ FnormalizeI(p) (`FnormalizeICorrect`). We omit these proofs as they are quite cumbersome but not difficult.

3.2 Negating a number

On IEEE-like number systems, the mantissa is stored with separate sign and magnitude, therefore $N_i = N_s$. This fact is not true on all floating point systems. Some hardware designers decided to use two's complement to store the mantissa as this is the case for Texas Instrument TMS 320C3x [21].

A bounded floating point number p can be negated if there exists another bounded floating point number q such that $q =_{\mathbb{R}} -p$. As we will see, almost any number can be negated even on systems based on the TMS 320C3x digital signal processors. The only two cases where a number cannot be negated cause either an overflow as the opposite of the least represented number is larger than the biggest number allowed in the number system or an underflow as the opposite of the least represented positive normal number is larger than the biggest negative number allowed in the number system. The second case would not occur on systems that handle denormal numbers.

The following theorem checked with Coq (`FoppBounded` and `FoppBoundedInv`) answers any question about negating a number. The cases study for a system that does not handle denormal numbers and for the upper bound on the amplitude are treated separately (`FoppBoundedExp`).

Theorem 3.1 *On a floating point system bounded by N_i, N_s and E_i with*

$N_i \neq N_s$, any bounded pair can be negated to a bounded float if and only if

$$|N_i - N_s| = 1 \qquad \textit{and} \qquad \beta \mid \max(N_i, N_s).$$

Without loss of generality, we assume that $N_s > N_i$. As a consequence, any pair (n, e) with $n \in \{-N_i, \cdots, N_i\}$ can be easily negated by negating its mantissa. The pairs (n, e) with $n \in \{N_i + 1, \cdots, N_s\}$ can only be negated by manipulating the amplitude. Therefore, β should divide all the $n \in \{N_i + 1, \cdots, N_s\}$. That is possible only for $N_i + 1 = N_s$ if β divides N_s. On the contrary, if N_s is a multiple of β and $N_i = N_s - 1$, any bounded pair can be negated to find another bounded pair.

We have also proved that the negation is the only opposite on a system that handles denormal numbers: if $x \oplus y = 0$, then y is the negation of x. Rephrasing [13] we first prove that the distance between two floating point numbers is at least β^{-E_i} then we conclude in the `OppositeIUnique`:

Theorem 3.2 *On a floating point system bounded by N_i, N_s and E_i, let P be any rounding mode and x and y be two bounded floats. If $y \neq_{\mathbb{R}} -x$ and z is a rounded result of $x + y$ then $|z| \geq \beta^{-E_i}$.*

3.3 Usual definitions of radix complement

Radix complement may or may not be used depending on the convention for negative numbers defined by the author for radix $\beta > 2$. The three common definitions are equivalent when $\beta = 2$.

The interpretation where the sign digit is -1 when $\beta - 1$ is stored in the most significant digit leads to the bounds

$$N_i = \beta^{p-1} \qquad \text{and} \qquad N_s = (\beta - 1) \cdot \beta^{p-1} - 1$$

with p bits of mantissa ($p > 1$). If $\beta > 2$, $N_s - N_i > 1$ and some bounded pairs cannot be negated without rounding.

Some authors use the previous convention but restrict the leading digit to 0 or $\beta - 1$. In this case,

$$N_i = \beta^{p-1} \qquad \text{and} \qquad N_s = \beta^{p-1} - 1,$$

so any pair can be negated.

When the interpretation is read modulo β^p and the digits are balanced evenly with possibly an additional digit to the negative set, the bound are

$$N_i = \left\lfloor \frac{\beta^p}{2} \right\rfloor \qquad \text{and} \qquad N_s = \left\lceil \frac{\beta^p}{2} \right\rceil - 1.$$

If β is odd, the set is evenly balanced and $N_i = N_s$. If β is even, $N_i = N_s + 1$ and β divided N_i since $p > 1$.

As a conclusion, it seems natural to prefer a sign-magnitude or a two's complement notation for the mantissa. We will see in Section 5 that all the existing implementations use one of these two classes.

3.4 Denormal numbers

The number system that we have just defined handles denormal pairs (gradual underflow) as this helps write more robust codes [7]. Sterbenz's theorem cannot be true if denormal numbers are not allowed. Let λ be the lowest positive normal number. Its value is

$$\lambda = \left(\left\lfloor \frac{N_s}{\beta} \right\rfloor + 1 \right) \times \beta^{-E_i}$$

and the following floating point number is

$$\lambda^+ = \left(\left\lfloor \frac{N_s}{\beta} \right\rfloor + 2 \right) \times \beta^{-E_i}.$$

The quantities λ and λ^+ verify $\lambda^+/2 \leq \lambda \leq 2\lambda^+$ and

$$\lambda^+ - \lambda = \beta^{-E_i}$$

that is a denormal number. This example shows that without allowing denormal numbers, the subtraction of x and y under the conditions of Sterbenz's theorem may not be represented.

3.5 Lexicographic order

Many authors, including [3], have recognized that it is a nice feature for lexicographic order of the floating point pairs to coincide with the order of the represented real values. As this fact is not necessary trivial in a generic floating point system, we establish the two following `LexicoPosCanI` and `LexicoCanI` theorems.

Theorem 3.3 *On a floating point system bounded by N_i, N_s and E_i, for any canonical pair (n_x, e_x) representing x and any bounded pair (n_y, e_y) representing y*

$$0 \leq x \leq y \qquad \textit{implies} \qquad e_x \leq e_y.$$

Theorem 3.4 *On a floating point system bounded by N_i, N_s and E_i with $|N_i - N_s| \leq 1$, for any canonical pair (n_x, e_x) representing x and any bounded pair (n_y, e_y) representing y*

$$|x| < |y| \qquad \textit{implies} \qquad e_x \leq e_y.$$

The difference between the preceding theorems and the usual IEEE like situation arises from the fact that the magnitude of a floating point pair may

not be represented or may use another amplitude. We establish the following corollary.

Corollary 3.5 *On a floating point system bounded by N_i, N_s and E_i with $|N_i - N_s| \leq 1$, for any canonical pair (n_x, e_x) representing x and any bounded pair (n_y, e_y) representing y*

$$e_x < e_y \qquad \textit{implies} \qquad |x| \leq |y|.$$

This means that when $|N_i - N_s| \leq 1$, our floating point system behaves like a IEEE compliant implementation as far as lexicographical order is concerned.

When $|N_i - N_s| > 1$, we have a very different behavior. Here is an example that also shows that the bound on the difference $N_i - N_s$ is tight. We define a binary notation with the mantissa between -1001_2 and 111_2. The pairs $(100_2, 1)_2$ and $(-1001_2, 0)_2$ are canonical yet their magnitudes and their amplitudes are not in the same order. This cannot happen in IEEE compliant systems or on the TMS 320C3x.

4 Sterbenz's theorem

4.1 A first very general theorem

It is amazing to realize that the following theorem is true whatever the radix and the bounds N_i and N_s. Moreover, the proof has been upgraded automatically by the Coq proof checker from the previous proof `SterbenzAux` presented in [6] that was supposed to work only when $N_i = N_s$.

Theorem 4.1 *On a floating point system bounded by N_i, N_s and E_i with no assumption of a relation between N_i and N_s, for any bounded pairs (n_x, e_x) and (n_y, e_y) representing x and y such that*

$$y \leq x \leq 2y,$$

the difference $x - y$ can be represented by a bounded pair (n, e). Furthermore, the bounded mantissa n and the bounded amplitude e can be defined as

$$n = n_x \beta^{e_x - \min(e_x, e_y)} - n_y \beta^{e_y - \min(e_x, e_y)}$$
$$e = \min(e_x, e_y).$$

On a floating point system where any bounded pair can be negated without rounding such as presented section 3.2, Sterbenz theorem `SterbenzOppI` stated below is proved by applying twice Theorem 4.1.

Theorem 4.2 *On a floating point system bounded by N_i, N_s and E_i where any bounded pair can be negated to another bounded pair, for any bounded pairs x and y such that*

$$\frac{y}{2} \leq x \leq 2y,$$

the difference $x - y$ can be represented by a bounded pair.

We prove the theorem correct when $y/2 \leq x \leq y$ by applying Theorem 4.1 to $X = y$ and $Y = x$ so that $X - Y = -(y - x)$ can be represented by a bounded pair.

4.2 Other systems

Although we presented in section 3 that it is most desirable to use a number system with a few natural properties including the fact that every bounded pair can be negated without rounding, we present now the `SterbenzI` very generic theorem. The details of the proof are available on the Internet.

Theorem 4.3 *On a floating point system bounded by N_i, N_s and E_i where $|N_i - N_s| \leq \delta$, for any canonical pair (n_x, e_x) representing x and any bounded pair (n_y, e_y) representing y such that*

$$\frac{y + \delta\beta^{\min(e_x,e_y)}}{2} \leq x \leq 2y,$$

the difference $x - y$ can be represented by a bounded pair.

5 Concluding remarks

5.1 Overview of existing hardware implementing sign magnitude

Most general purpose widely available processors use a sign magnitude representation. Some books [10,15] even present the sign magnitude notation as the natural floating point notation. This notation is in use in the well-studied IEEE-754 compliant hardware. Some IBM systems use radix 10 [4] and a few of them retain a radix 16 compatibility mode [19]. Yet most systems use radix 2. For all these systems, Sterbenz's equality holds with all the natural properties presented in this work.

The properties presented here were scattered in the litterature as most of them have been published over the time. Yet the main motivations of this work was to compare the most common general purpose IEEE 754 based behavior to other implementations such as IEEE 854 compatible circuits, almost IEEE 854 behavior and non IEEE behavior as we see in the following.

5.2 Texas Instrument's two's complement notation

Texas Instrument uses in its TMS 320C3x the two's complement notation for the mantissa. This notation was also in use in Honeywell 6080N computer [18]. A different notation with the same mantissa range is studied in a exercise of [12]. This number system is well suited as all the natural properties (stability through negation, existence and unicity of an opposite and lexicographic order of the pairs) are still true and Sterbenz's equality holds. We do not have any

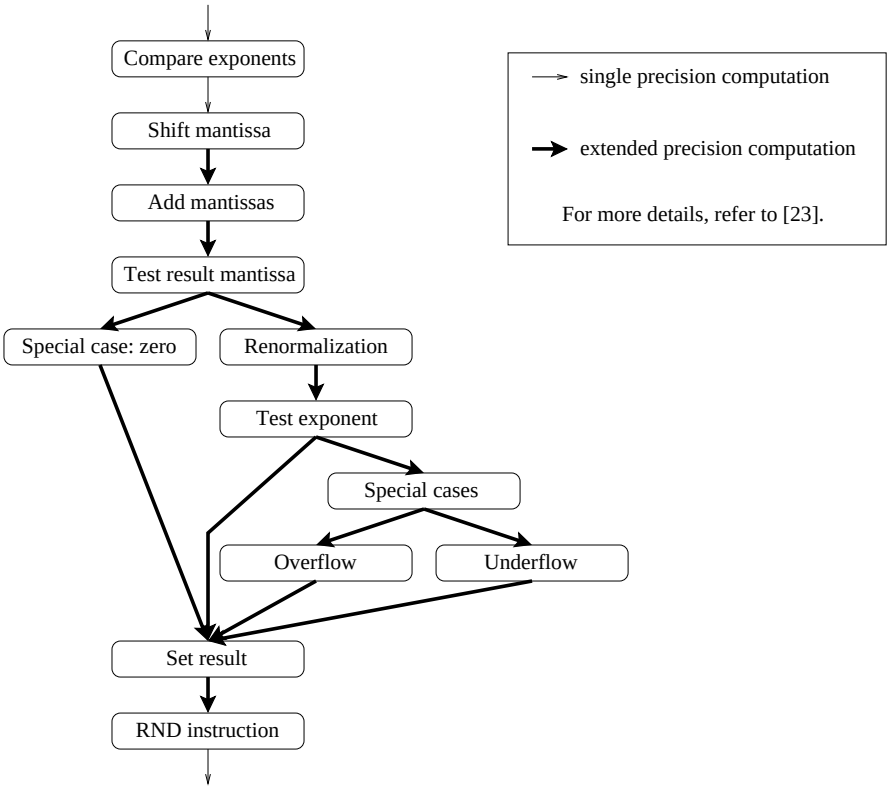


Figure 1. Flowchart for floating point addition followed by a RND instruction

knowledge of a working floating point unit that uses neither sign-magnitude nor the two's complement notation for the mantissa encoding.

The following theorem (`ReductRange` and `ReductRangeInv`) can be used to deduce that the set of the represented numbers is almost identical with an IEEE-compatible unit and the TMS 320C3x. Should Texas Instrument decide to implement denormal pairs and precise rounding the unit could be functionnaly IEEE-compliant.

Theorem 5.1 *The set of the real numbers represented on a floating point system bounded by $N_i > 1$, $N_s > 1$ and E_i is identical to the set of the numbers represented on a system bounded by N_i, $N_s - 1$ and E_i (respectively $N_i - 1$, N_s and E_i) if and only if*

$$\beta \mid N_s \qquad \textit{(respectively} \quad \beta \mid N_i).$$

All the theoretical results presented in this text prove that under a few assumptions there exists a bounded float that is the exact result of the subtraction. We have to look at the way the addition/subtraction is performed by the TMS 320C3x to be sure that this exact result is really returned by the unit.

Figure 1 presents a simplified version of the flowchart of the addition. This operation is first performed on extended precision and then rounded. The

mantissa of one of the inputs is possibly shifted depending on the actual value of the difference of the amplitudes. The result of the addition of the mantissas lies in 30 bits. That means that 8 additional bits are used for the intermediate result. Adding a new case to the result of [10], we see that one guard bit is sufficient for Sterbenz's theorem to hold even using a different notation than the IEEE-like sign-magnitude for the mantissa. On the contrary, the Sterbenz's theorem does not hold if the user manipulates extended numbers rather than single precision numbers. In this case, the operation is performed without any guard bit and the result is not necessarily found by the floating point unit.

If no exception is triggered, the mantissa is accurate enough to hold the exact result and the result before rounding is the expected exact result. As this result was proved to be bounded, the rounding does not change it and the final result is exact.

We deduce immediately from $y/2 \leq x \leq 2 \times y$ that $x - y \in [-y/2; y]$ so no overflow can occur. If the exact result is a denormal number, the TMS 320C3x returns 0 as this processor does not handles such numbers.

In this text, we have shown that the floating point number system used for the TMS 320C3x defines almost the same real values as the system of an IEEE-compliant processor with a very different interpretation for the mantissa field. We have also shown that gradual underflow and correct rounding would be very sensible in such a system although neither was implemented. Finally, we have proved some very useful result about the TMS 320C3x such as Sterbenz's theorem provided no underflow occurred.

5.3 On automatic proof checking

Without a strong incentive on formal analysis of the TMS 320C3x, such work would probably not have been carried out. It has been made possible by the very formal and generic development of the proofs used in Coq. Odds are that such conclusions would scarcely be trusted if they were not checked by an automatic proof checker since the proofs are very technical and prone to many small mistakes that would not have been ruled out by experimental knowledge.

We will continue to investigate natural properties of floating point number systems as they lead us to necessary conditions on the number systems. In the case of this work, Sterbenz's equality and the possibility to negate a number are also key to analyze numerical software behavior such as [5].

6 Acknowledgments

We wish to thank Laurence Rideau and Laurent Thry for they work on the initial development of the Coq theory library.

References

[1] Amerkad, A., Y. Bertot, L. Rideau and L. Pottier, *Mathematics and proof presentation in Pcoq*, in: *Proceedings of Proof Transformation and Presentation and Proof Complexities*, Siena, Italy, 2001.
URL
`http://www-sop.inria.fr/lemme/Laurence.Rideau/proof-pco\%q.ps.gz`

[2] Carreño, V. A. and P. S. Miner, *Specification of the IEEE-854 floating-point standard in HOL and PVS*, in: *1995 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, 1995, supplemental Proceedings.
URL `http://shemesh.larc.nasa.gov/fm/ftp/larc/vac/hug95.ps`

[3] Coonen, J. T., *Specification for a proposed standard for floating point arithmetic*, Memorandum ERL M78/72, University of California, Berkeley (1978).

[4] Cowlishaw, M. F., E. M. Schwarz, R. M. Smith and C. F. Webb, *A decimal floating point specification*, in: N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th Symposium on Computer Arithmetic*, Vail, Colorado, 2001, pp. 147–154.
URL `http://computer.org/proceedings/arith/`

[5] Daumas, M. and P. Langlois, *Additive symmetric: the non-negative case*, Theoretical Computer Science (2002).

[6] Daumas, M., L. Rideau and L. Théry, *A generic library of floating-point numbers and its application to exact computing*, in: *14th International Conference on Theorem Proving in Higher Order Logics*, Edinburgh, Scotland, 2001, pp. 169–184.
URL `http://link.springer.de/link/service/series/0558/bibs/2\%152/21520169.htm`

[7] Demmel, J., *Underflow and the reliability of numerical software*, SIAM Journal on Scientific and Statistical Computing **5** (1984), pp. 887–919.

[8] Goldberg, D., *What every computer scientist should know about floating point arithmetic*, ACM Computing Surveys **23** (1991), pp. 5–47.
URL `http://www.acm.org/pubs/articles/journals/surveys/1991-\%23-1/p5-goldberg/p5-goldberg.pdf`

[9] Harrison, J., *Floating point verification in HOL light: the exponential function*, Technical Report 428, University of Cambridge Computer Laboratory (1997).
URL `http://www.cl.cam.ac.uk/users/jrh/papers/tang.ps.gz`

[10] Higham, N. J., "Accuracy and stability of numerical algorithms," SIAM, 1996.
URL `http://www.ma.man.ac.uk/~higham/asna.html`

[11] Huet, G., G. Kahn and C. Paulin-Mohring, *The Coq proof assistant: a tutorial: version 6.1*, Technical Report 204, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France (1997).

URL ftp://ftp.inria.fr/INRIA/publication/publi-pdf/RT/RT-02\%04.pdf

[12] Knuth, D. E., "The Art of Computer Programming: Seminumerical Algorithms," Addison-Wesley, 1997, third edition.

[13] Kulisch, U., *Rounding near zero*, in: *4th Real Numbers and Computers Conference*, Dagstuhl, Germany, 2000, pp. 23–29.

[14] Laurent, O., P. Michel and V. Wiels, *Using formal verification techniques to reduce simulation and test effort*, in: *International Symposium of Formal Methods Europe*, Berlin, Germany, 2001, pp. 465–477.
URL http://link.springer.de/link/service/series/0558/papers\%/2021/20210465.pdf

[15] Overton, M. J., "Numerical Computing with IEEE Floating Point Arithmetic," SIAM, 2001.
URL http://www.siam.org/catalog/mcc07/ot76.htm

[16] Rushby, J. and F. von Henke, *Formal verification of algorithms for critical systems*, in: *Proceedings of the Conference on Software for Critical Systems*, New Orleans, Louisiana, 1991, pp. 1–15.
URL http://www.acm.org/pubs/articles/proceedings/soft/12508\%3/p1-rushby/p1-rushby.pdf

[17] Russinoff, D. M., *A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor*, LMS Journal of Computation and Mathematics **1** (1998), pp. 148–200.
URL http://www.onr.com/user/russ/david/k7-div-sqrt.ps

[18] Schryer, N. L., *A test of computer's floating-point arithmetic unit*, Technical report 89, AT&T Bell Laboratories (1981).
URL http://cm.bell-labs.com/cm/cs/cstr/89.ps.gz

[19] Schwarz, E. M., R. M. Smith and C. A. Krygowski, *The S/390 G5 floating point unit supporting hex and binary architectures*, in: I. Koren and P. Kornerup, editors, *Proceedings of the 14th Symposium on Computer Arithmetic*, Adelaide, Australia, 1999, pp. 258–265.
URL http://computer.org/proceedings/arith/0116/0116toc.htm

[20] Sterbenz, P. H., "Floating point computation," Prentice Hall, 1974.

[21] Texas Instruments, "TMS320C3x — User's guide," (1997).
URL http://www-s.ti.com/sc/psheets/spru031e/spru031e.pdf

Simple and Efficient Translation from LTL Formulas to Büchi Automata

Xavier Thirioux [1]

IRIT - LIMA, 2 rue Camichel, 31071 Toulouse, France,
Xavier.Thirioux@enseeiht.fr

Abstract

We present a collection of simple on-the-fly techniques to generate small Büchi automata from Linear Time Logic formulas. These techniques mainly involve syntactic characterizations of formulas, and yet allow efficient computations. Thus heavily relying on such proof-theoretic issues, we can omit the classical formula pre-simplification step, and also simulation-based post-simplification steps (aka model-theoretic issues).

Although closely related to other similar recent works in the same topic, our ideas have led to an implementation that performs significantly better than some of the best available tools, such as Wring or LTL2BA. We compare our tool BAOM ("Büchi Automata Once More") with others, on formulas commonly found in the literature, and on randomly generated testbenchs.

Key words: Linear Time Logic. Büchi automata. Tableaux-based method. Syntactic characterizations of formulas.

Introduction

This paper describes several new techniques to implement an efficient translation from Linear Time Logic (LTL) specifications to Büchi automata. Our prime motivation was to implement a new symbolic BDD-based [2] model-checker [BCM+92] based upon Linear Time Logic specifications for synchronous programs, especially those written in Esterel. This work extends previous works on the Xeve model-checker [Bou97].

In our symbolic framework, a well-known solution to achieve this goal is first to translate (the negation of) any given LTL formula into an observer (a finite state machine) that is plugged into the original design we want to check. Then, by symbolic forward image computations of the product system, find

[1] This work was partially supported by the SYNTEL RNRT project, France.
[2] BDD stands for Binary Decision Diagram.

an execution that meets the fairness side conditions imposed by the formula, the so-called Büchi conditions. Though the principle is rather simple, an efficient implementation that avoids exponential blow-ups during translation of a formula into an observer is difficult to achieve. Recently, some promising new algorithms have been found to lower the size of automata. The resulting tools, namely LTL2BA, Wring and EqLTL (see respectively [GO01,SB00,EH00]) seemingly outperform SPIN [Hol97].

Our main contribution is to provide a new implementation (called BAOM) that behaves linearly (in space and time) for many more formulas (especially formulas containing fairness constraints), where previous algorithms exhibit exponential blow-ups. Moreover, we obtain automata that are in the average smaller than with any other method. Our prototype directly builds a Generalized Büchi Automaton from a Linear Temporal Logic formula, without the need for intermediate data-structures. For matters of efficiency, our transitions are labelled with BDDs rather than with single atomic propositions. Also, our automata are generalized in a mixed sense, i.e. we have fairness conditions on states as well as on transitions, depending upon the shape of the formula. Notice that our ideas are primarily concerned with on-the-fly optimizations based upon syntactic relations between formulas. We also introduce a notion of "merged" states in our automata, in order to factorize sub-tableaux when possible and reduce further the number of states.

Nevertheless, our algorithm widely borrows ideas from recent tableaux methods, from recent techniques around alternating automata, and also from syntactic relations between formulas. Actually, we include the syntactic simplification rules of [SB00] on-the-fly during the tableaux generation, and generalize in this way similar rules of [DGV99] as well as [GO01]. Some of these rules can also be seen as an cheap alternative to the "boolean optimization" paradigm of [SB00], which is a general solution to remove redundant and complementary sub-formulas occurring in tableaux. In our case, this simplification may introduce new fairness constraints on transitions, as in [GO01].

Related works come in many flavours, but are principaly concerned with improvements of the tableau method described in [VW94] and [GPVW95]. In [EH00] the authors present an algorithm in three steps : first a rewriting step, followed by a standard translation and finally a simulation-based optimization. In [SB00], the same kind of techniques are applied, yet with totally different rewriting rules and with simulation relations that can be computed more efficiently. In [GO01], the authors also reuse the same set of rewriting rules as in [SB00], and consider very simple on-the-fly simplification rules that avoid fixpoint computations necessary in simulation-based methods. The simplification process, though simple, is still efficient due to a specific translation based upon alternating automata where fairness constraints exclusively concern transitions.

Finally, all these works point out the ability to simplify in some cases the

fairness constraints of the SCCs[3] of the generated automata, for instance by the recursive removing of the unfair terminal SCCs.

The roadmap : section 1 introduces preliminary notions and reminds the original tableaux method. Section 2 presents our new revisited tableaux algorithm that builds a first version of a Büchi automaton from a LTL formula. This algorithm splits in four parts :

(i) Use modified tableaux rules to generate a basic automaton (section 2.1);

(ii) Normalize and simplify transitions (section 2.2);

(iii) Detect unfair SCCs and simplify the automaton (section 2.3);

(iv) Merge transition-equivalent states (section 2.4).

Thereafter, section 3 presents a classical post-simplification phase to reduce the number of states. Finally, we show in section 4 some promising results of our prototype and compare them with other similar tools, and then we conclude in section 5.

1 Preliminaries

We define here a variant of Büchi automata, also called Generalized Büchi Automata, with multiple acceptance conditions on the states as well as on the transition edges. Labels are located on the transition edges, and are boolean formulas (denoted below as $\mathcal{B}(AP)$) built from a set of atomic propositions AP([4]). We make use of a data structure to represent the edges (thus replacing the traditional δ function) because we actually need to distinguish between different transition edges with different fairness constraints and compatible labels. Notice that we treat uniformly state and transition fairness.

Definition 1.1 *A GBA is a five-tuple :*

$$\mathcal{A} = \langle AP, Q, Q_0, E, \mathcal{F} \rangle$$

where AP is the set of atomic propositions, Q is the finite set of states, $Q_0 \subseteq Q$ is the set of initial states, $E \subseteq Q \times \mathcal{B}(AP) \times Q$ is the set of edges, and $\mathcal{F} \subseteq \mathcal{B}(Q \cup E)$ is the set of acceptance conditions, expressed as logical constraints. A (generalized) transition function $\delta \in \mathcal{B}(AP) \rightarrow 2^Q \rightarrow 2^Q$ can be recovered from E as :

$$\delta(p, qs) = \{q' \mid \exists q, l, q'.q \in qs \wedge \langle q, l, q' \rangle \in E \wedge \models p \rightarrow l\}$$

As p and l are encoded as BDD, we can easily decide whether $\models p \rightarrow l$ holds or not.

A run of $\mathcal{A}$ is an infinite sequence $\sigma = \langle q_0, i_0, t_0 \rangle; \langle q_1, i_1, t_1 \rangle; \ldots$ where $q_k \in Q$, $t_k \in E$ and $i_k \subseteq AP$, such that for all $k \geq 0$:

$$t_k = \langle q_k, l_k, q_{k+1} \rangle \wedge i_k \models l_k$$

[3] SCC stands for Strongly Connected Component.

[4] For X a set of ground terms, $\mathcal{B}(X)$ denotes its boolean closure.

A run σ is accepting if for each $F \in \mathcal{F}$, we have infinitely many k's such that :

$$t_{\mathrm{k}}, q_{\mathrm{k+1}} \models F$$

Finally, an automaton $\mathcal{A}$ accepts an infinite word of input events $i = i_0, i_1, \ldots$ over $(2^{AP})^{\omega}$, whenever there exists an accepting run of $\mathcal{A}$:

$$\sigma = \langle q_0, i_0, t_0 \rangle; \langle q_1, i_1, t_1 \rangle; \ldots$$

Its language $\mathcal{L}(\mathcal{A})$ is the set of infinite words it accepts.

Multiple initial states and multiple acceptance conditions are not mandatory, but are considered here only for convenience with respect to the overall model-checking process in which the translation step occurs.

Definition 1.2 *The linear time logic (LTL) is built from propositional logic by adding temporal operators, yielding the following syntax :*

$$\begin{aligned} LTL ::= \; & AP \\ | \; & \mathcal{B}(LTL) \\ | \; & LTL \; \mathsf{U} \; LTL \\ | \; & LTL \; \mathsf{R} \; LTL \\ | \; & \bigcirc \; LTL \end{aligned}$$

$\bigcirc$ is the "next-time" operator, U is the (strong) "until" operator and R is the "release" operator. R and U are dual of each other. Usual $\Box$ ("always") and $\Diamond$ ("eventually") operators are defined as $\Box\phi = False \; \mathsf{R} \; \phi$ and $\Diamond\phi = True \; \mathsf{U} \; \phi$.

We briefly recall here the standard tableaux method [VW94,GPVW95], as we use it as a basis for our own extension. Each state of the automaton denotes and identifies a LTL formula in negative normal form, i.e. where negation has been pushed down the parse tree of the formula. Then, from a given state, the transition function is computed by means of semantic expansion rules. These rules consist in applying from left to right the following equalities to the state-formula, through the expansion function Exp :

$$\begin{aligned} Exp(prop) &= prop \\ Exp(\bigcirc\phi) &= \bigcirc\phi \\ Exp(\phi \vee \psi) &= Exp(\phi) \vee Exp(\psi) \\ Exp(\phi \wedge \psi) &= Exp(\phi) \wedge Exp(\psi) \\ Exp(\phi \; \mathsf{U} \; \psi) &= Exp(\psi \vee (\phi \wedge \bigcirc(\phi \; \mathsf{U} \; \psi))) \\ Exp(\phi \; \mathsf{R} \; \psi) &= Exp(\psi \wedge (\phi \vee \bigcirc(\phi \; \mathsf{R} \; \psi))) \end{aligned}$$

Thus, starting with a formula ϕ, we first expand it and then put it into disjunctive normal form. Each conjunctive term $\psi = \psi_1 \wedge \psi_2 \ldots$ will constitute a next state. According to the above rules, each ψ_{i} is either an atomic proposi-

tion ap_{i} or a next-time formula $\bigcirc\theta_{\mathrm{i}}$. Hence, the ψ-state will be labelled by a formula $\theta_1 \wedge \theta_2 \ldots$ whereas the transition edge from the ϕ-state to the ψ-state will be labelled by a propositional formula $ap_1 \wedge ap_2 \ldots$.

As usual, transitions labelled with unsatisfiable propositions are removed, thus removing unreachable states as well.

The initial states are built from expansion of the root formula. Multiple initial states can be avoided if the initial formula is not expanded. This may increase or decrease the number of states, depending upon the formula (see theorem 2.13).

Finally, as for the Büchi acceptance conditions, for each $\phi \mathsf{U} \psi$ occurring in state-formulas, there exists an acceptance formula $Fair_{\phi \mathsf{U} \psi}$ on states :

$$Fair_{\phi \mathsf{U} \psi} \ = \ \bigvee \{q \in Q \mid \phi \mathsf{U} \psi \notin q \ \vee \ \psi \in q\}$$

Then, for this particular kind of formulas on states, the condition for a run to be accepted boils down to the following statement : for each set $Fair_{\phi \mathsf{U} \psi}$, we have infinitely many q_k's such that $q_k \models Fair_{\phi \mathsf{U} \psi}$.

2 Tableaux method revisited

In the remainder, we propose different steps aiming at reducing the size of automata. All these improvements are relative to a pervasive automaton $\mathcal{M} = \langle AP, Q, Q_0, E, \mathcal{F} \rangle$ assumed at each step to be the result of previous transformations. To ease the description of our method, we define a notion of substitution on sets as :

$$S[e' \mid e] = \begin{cases} (S \setminus \{e\}) \cup \{e'\} \text{ if } e \in S \\ S \text{ else} \end{cases}$$

2.1 Expanding tableaux rules

When designing this new algorithm, our main goal was to obtain small and deterministic automata from a standard tableaux-based method.

We now define a temporal approximation of a LTL formula, driven by a positive integer, denoted as $\lceil \phi \rceil^d$. This approximation is a formula representing exactly the finite d-prefixes of infinite words identified by ϕ.

Definition 2.1 *For any $\phi \in$LTL under negative normal form and $d \geq 0$, we*

define the function $\lceil\phi\rceil^d$ *as below :*

$$\begin{aligned}
\lceil ap\rceil^d &= ap\\
\lceil \phi \vee \psi\rceil^d &= \lceil\phi\rceil^d \vee \lceil\psi\rceil^d\\
\lceil \phi \wedge \psi\rceil^d &= \lceil\phi\rceil^d \wedge \lceil\psi\rceil^d\\
\lceil \phi \ \mathsf{U}\ \psi\rceil^d &= \lceil\psi \vee (\phi \wedge \bigcirc(\phi \ \mathsf{U}\ \psi))\rceil^d\\
\lceil \phi \ \mathsf{R}\ \psi\rceil^d &= \lceil\psi \wedge (\phi \vee \bigcirc(\phi \ \mathsf{R}\ \psi))\rceil^d\\
\lceil \bigcirc\phi\rceil^0 &= True\\
\lceil \bigcirc\phi\rceil^{d+1} &= \bigcirc\lceil\phi\rceil^d
\end{aligned}$$

Lemma 2.2 *For any* $\phi \in LTL$ *and any* $d \geq 0$*, we have :* $\phi \Rightarrow \lceil\phi\rceil^d$ *and also* $\phi \vee (\lceil\neg\phi\rceil^d \wedge \psi) \Leftrightarrow \phi \vee \psi$*.*

Proof (sketch) The implication is proved by structural induction on ϕ. The equivalence then follows.

We can now modify the expansion rules taking into account this finite approximation. Indeed, given any integer d, we can use the following new rules for U and R, where $\neg\psi$ and $\neg\phi$ are put in negative normal form :

$$Exp(\phi \ \mathsf{U}\ \psi) = Exp(\psi \vee (\phi \wedge (\lceil\neg\psi\rceil^d \wedge \bigcirc(\phi \ \mathsf{U}\ \psi))))$$

$$Exp(\phi \ \mathsf{R}\ \psi) = Exp(\psi \wedge (\phi \vee (\lceil\neg\phi\rceil^d \wedge \bigcirc(\phi \ \mathsf{R}\ \psi))))$$

Theorem 2.3 *For any* $\phi \in LTL$ *and any* $d \geq 0$*, let* $\mathcal{M}_\mathrm{d}$ *be the automaton produced using revised expansion rules, then* $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}_\mathrm{d})$*.*

Proof (sketch) The automata are produced according to semantic expansion rules so that they exactly accept words denoted by state-formulas. Then lemma 2.2 is used to convert revised formulas to standard ones, proving that $\mathcal{M}$ and $\mathcal{M}_\mathrm{d}$ accept the same language.

For our specific usage, using a great value of d may increase the number of different states, as the $\lceil.\rceil^d$ operator generates new next-time sub-formulas. Though, due to the extra constraints upon transitions introduced by the prefix formulas, this may also increase the deterministic flavour of the resulting automaton, that is we obtain at an early stage many more incompatible transitions, which later on we won't have to compare (see theorem 2.9).

After some conclusive experiments showing that the average number of states tend to increase as d does, we decided for the time being to restrain our choice to $d = 0$, leading to a good balance between a small overhead and a better overall performance. For instance, with formulas under the form $\bigwedge_{\mathrm{i}=1\ldots\mathrm{N}} \Box\Diamond p_\mathrm{i}$, regarding N as a parameter, our method can save upto an exponential number of states with respect to the Wring tool, or proceed exponentially faster than the LTL2BA tool (we obtain the same automaton in

this case [5]).

We now assume that each state denotes a conjunctively interpreted set of formulas, instead of a single conjunctive formula.

Let $\phi \leq \psi$ be a relation of syntactic implication between two formulas, similar to the ones presented in [SB00] and [DGV99]. This relation will greatly help us in reducing the size of automata. Notice that this relation doesn't expand temporal operators, so that its computational cost is moderate.

Definition 2.4 *For any $\phi, \psi \in LTL$, we define the relation $\phi \leq \psi$ as the smallest fixpoint of the following rules :*

$False \leq \psi$	$\phi \leq True$
$\phi_1 \vee \phi_2 \leq \psi \Leftarrow \phi_1 \leq \psi \wedge \phi_2 \leq \psi$	$\phi \leq \psi_1 \wedge \psi_2 \Leftarrow \phi \leq \psi_1 \wedge \phi \leq \psi_2$
$\phi \leq \psi_1 \vee \psi_2 \Leftarrow \phi \leq \psi_1 \vee \phi \leq \psi_2$	$\phi_1 \wedge \phi_2 \leq \psi \Leftarrow \phi_1 \leq \psi \vee \phi_2 \leq \psi$
$\phi_1 \mathsf{R} \phi_2 \leq \psi \Leftarrow \phi_2 \leq \psi$	$\phi \leq \psi_1 \mathsf{U} \psi_2 \Leftarrow \phi \leq \psi_2$
$\phi_1 \mathsf{R} \phi_2 \leq \psi_1 \mathsf{R} \psi_2 \Leftarrow \phi_1 \leq \psi_1 \wedge \phi_2 \leq \psi_2$	$\phi_1 \mathsf{U} \phi_2 \leq \psi_1 \mathsf{U} \psi_2 \Leftarrow \phi_1 \leq \psi_1 \wedge \phi_2 \leq \psi_2$

This definition allows us to remove weak formulas from states, and therefore to reduce in many cases the number of different states, by the mean of the following theorem.

Theorem 2.5 *Let $q = \{\phi_1, \phi_2, \ldots, \phi_n\} \in Q$ be a state such that $\phi_1 \leq \phi_2$. Let q' denote the set $\{\phi_2, \ldots, \phi_n\}$. Then $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$ with the automaton $\mathcal{M}' = \langle AP, Q', Q'_0, E', \mathcal{F}' \rangle$ defined below :*

- $Q' = Q[q' \mid q]$
- $Q'_0 = Q_0[q' \mid q]$
- $E' = \{\langle q_{\text{src}}, l, q_{\text{dst}}[q' \mid q] \rangle \mid \langle q_{\text{src}}, l, q_{\text{dst}} \rangle \in E\}$
- $\mathcal{F}' = \left\{ \begin{array}{l} \{\mathcal{F}air_\phi[q' \mid q] \mid \mathcal{F}air_\phi \in \mathcal{F}\} \quad \textit{if } \phi_1 \neq \llcorner \mathsf{U} \lrcorner \\ \mathcal{F}[\mathcal{F}air'_{\phi_1} \mid \mathcal{F}air_{\phi_1}] \quad \textit{else, with} \\ \quad \mathcal{F}air'_{\phi_1} = \mathcal{F}air_{\phi_1}[q' \mid q] \wedge (\neg q' \bigvee \{\langle q_{\text{src}}, l, q_{\text{dst}} \rangle \in E \mid q_{\text{dst}} \neq q\}) \end{array} \right\}$

Proof (sketch) As our implication relation is easily proved to be sound, the two automata accept the same language, disregarding fairness constraints. In the case the removed formula ϕ_1 is an until formula and thus involves a change in acceptance conditions, we report the fairness of ϕ_1 onto all the incoming transition edges of state q'. Hence we mimic the standard situation where fairness is on state q.

[5] As shown in test cases presented in section 4.

2.2 Normalizing transitions

Once the outgoing transition edges from a given state are built, we proceed with a normalization step in which we factor transitions. This factorization is possible (and simple) because we use BDDs to represent transition labels.

Definition 2.6 *Assuming that $\phi \in LTL$ is such that $\mathcal{F}air_\phi \in \mathcal{F}$ and $q, q' \in Q$, we define the following global fair (and unfair) labeling functions between two states :*

$$l_\phi(q,q') = \bigvee\{l \mid t = \langle q, l, q'\rangle \in E \wedge t, q' \models \mathcal{F}air_\phi\}$$
$$l_{\text{unfair}}(q,q') = \bigvee\{l \mid t = \langle q, l, q'\rangle \in E \wedge \forall \mathcal{F}air_\phi \in \mathcal{F}.t, q' \not\models \mathcal{F}air_\phi\}$$

With these functions, we can define our normalization step.

Theorem 2.7 *Let us define the automaton $\mathcal{M}' = \langle AP, Q, Q_0, E', \mathcal{F}'\rangle$:*

- $E' = \{\langle q, l, q'\rangle \mid l = l_\phi(q,q') \vee l = l_{\text{unfair}}(q,q')\}$
- $\mathcal{F}' = \{\mathcal{F}air'_\phi \mid \mathcal{F}air_\phi \in \mathcal{F}\}$ *with*
  $\mathcal{F}air'_\phi = \bigvee\{t \wedge q' \mid t = \langle q, l, q'\rangle \in E' \wedge l = l_\phi(q,q')\}$

Then $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Proof (sketch) This normalization may reduce the number of transition edges between two states[6] but has no effect on the language of $\mathcal{M}$ since we factorize edges with respect to fairness constraints. The only case to which we must pay attention is when a given transition edge is fair regarding at least two different constraints. Then we must split it into (at least) two different edges with the same label, each satisfying only one fair constraint. But for any accepted word, if an original edge of $\mathcal{M}$ would be triggered infinitely often, the resulting edges of $\mathcal{M}'$ could also be triggered infinitely often, each in turn. The converse also holds. So finally $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Notice that in the above theorem, we can indeed easily simplify the fairness constraints, in order to keep only transition fairness. Actually, defining :

$$\mathcal{F}air'_\phi = \bigvee\{t \mid t = \langle q, l, q'\rangle \in E' \wedge l = l_\phi(q,q')\}$$

would also yield the same result. But we decided to keep both kinds of fairness information because this allows to implement simpler algorithms in our prototype.

The next step consists in trying to determinize transitions from any given state, i.e. to modify their labels so that their pairwise intersection becomes empty. This usually leads to a lesser number of (smaller) transition edges, and allow in practice further simplifications (see theorem 2.13). Besides, the structure of the automaton is then more easily amenable to efficient model-checking algorithms. Actually, the deterministic flavour of an automaton is a

[6] After this operation, there always exist less than $|\mathcal{F}|+1$ different transition edges between any two states.

salient feature in symbolic model-checking, because it appears to have a great influence on efficiency of partitioned states space exploration algorithms for instance.

Definition 2.8 *We classically extend the notion of implication between formulas to an implication between states. For any $q, q' \in Q$, we define :*

$$q \leq q' \ = \ \forall \phi' \in q'. \exists \phi \in q. \phi \leq \phi'$$

Theorem 2.9 *Let us consider $t_1 = \langle q, l_1, q_1 \rangle \in E$ and also $t_2 = \langle q, l_2, q_2 \rangle \in E$. Now assume $q_1 \leq q_2$, $l_1 \wedge l_2 \neq False$ and :*

$$\forall \mathcal{F}air_\phi \in \mathcal{F}. t_1, q_1 \models \mathcal{F}air_\phi \ \Rightarrow \ t_2, q_2 \models \mathcal{F}air_\phi$$

Then the automaton $\mathcal{M}' = \langle AP, Q, Q_0, E', \mathcal{F}' \rangle$ defined below :

- *$E' = E[t'_1 \mid t_1]$ with $t'_1 = \langle q, l_1 \wedge \neg l_2, q_1 \rangle$*
- *$\mathcal{F}' = \mathcal{F}$*

is such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$. We remove the new transition t'_1 from E' and set $\mathcal{F}' = \{\mathcal{F}air_\phi[False \mid t'_1] \mid \mathcal{F}air_\phi \in \mathcal{F}\}$ if $l_1 \wedge \neg l_2 = False$ holds.

Proof (sketch) Following definitions 2.4 and 2.8, $q_1 \leq q_2$ implies $\mathcal{L}(q_1) \subseteq \mathcal{L}(q_2)$. So, we can safely remove from t_1 the input events that are common with t_2. The fairness constraints don't need to be changed (but in case of mere removal) since it is easier to reach q_2 through t_2 than to reach q_1 through t_1 w.r.t. fairness constraints, and $l_1 \vee l_2 = (l_1 \wedge \neg l_2) \vee l_2$. Hence, if we had an accepted word passing from q to q_1 through $l_1 \wedge l_2$, we know that it would also be accepted via q_2.

2.3 Detecting unfair SCCs

Our last but one on-the-fly step can simplify the fairness constraints on states, by early detecting of certain unfair SCCs, i.e. SCCs where at least one fairness constraint is never satisfied for any of its states, or transient SCCs, i.e. SCCs with only one state and no self loop. Besides, we define a syntactic under-approximation of unfair and transient SCCs.

Definition 2.10 *For $q \in Q$, we define the following* FairLoop *and* Unstable *predicates*[7] *:*

$$\begin{aligned} \text{Unstable}(q) \ &= \ \exists \phi \in q. \phi \neq True \wedge \neg \text{FairLoop}(\phi, q) \\ \text{FairLoop}(\phi, q) \ &= \ \exists \psi = \psi_1 \mathsf{R} \psi_2 \in q. \phi = \psi \vee \phi \prec \psi_2 \end{aligned}$$

Lemma 2.11 *For any $q \in Q$, such that* Unstable(q)*, then the SCC of q is either transient or unfair.*

[7] For any formulas ϕ and ψ, $\phi \prec \psi$ denotes the subterm relation, but for negated atoms. That is, for p an atomic proposition, we have $p \not\prec \neg p$.

Proof (sketch) By contradiction. Assume the SCC of q is fair. Then, following the expansion rules, each non-R formula ϕ must be (transitively) generated by the right-hand side ψ of a R formula, which are the only fair looping operators in LTL. Hence, because we don't change the shape of formulas when expanding them, it is necessary to check $\phi \prec \psi$. Therefore, Unstable(q) does not hold. As a conclusion, an accepted run cannot remain stuck in the SCC of an unstable state.

Theorem 2.12 *Let q be an unstable state, we change the fairness constraints by defining the automaton $\mathcal{M}' = \langle AP, Q, Q_0, E, \mathcal{F}' \rangle$ as :*

- $\mathcal{F}' = \{\mathcal{F}air_\phi[False \mid q] \mid \mathcal{F}air_\phi \in \mathcal{F}\}$

Then we have $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Proof (sketch) We can safely remove fairness information relative to an unstable state and its incoming transition edges, since an accepted word cannot visit it infinitely often, as proved by lemma 2.11. So, $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

2.4 Merging states

Last, but not least, we define the notion of merged states. It consists in merging some target states of some transitions with the same labels and fairness constraints, making a compound state. This also applies to the initial states that can be merged as one single state.

Theorem 2.13 *Let us consider $t_1 = \langle q, l_1, q_1 \rangle \in E$ and also $t_2 = \langle q, l_2, q_2 \rangle \in E$. Now assume we have $l_1 = l_2$([8]) and :*

$$\forall \mathcal{F}air_\phi \in \mathcal{F}. t_1, q_1 \models \mathcal{F}air_\phi \;\Leftrightarrow\; t_2, q_2 \models \mathcal{F}air_\phi$$

Then the automaton $\mathcal{M}' = \langle AP, Q', Q_0, E', \mathcal{F}' \rangle$ defined below :

- $Q' = Q \cup \{q_{12}\}$
- $E' = (E \setminus \{t_1, t_2\}) \cup \{t_{12}\}$ *with* $t_{12} = \langle q, l_1, q_{12} \rangle$
- $\mathcal{F}' = \{\mathcal{F}air_\phi[False \mid \{t_1, t_2\}] \vee \mathcal{F}air_\phi[t_{12} \wedge q_{12} \mid t_1 \wedge q_1] \mid \mathcal{F}air_\phi \in \mathcal{F}\}$

is such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Proof (sketch) As we ensure that transitions as well as target states have the same impact on fairness, we can merge them without modifying the set of accepted words. Notice that the original target states are not removed, but if they become unreachable. Then, $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

A merged state is defined as the set of its components. In theorem 2.13, we have thus $q_{12} = \{q_1, q_2\}$. Ordinary states can also be defined as singleton sets. So from now on we move up a level and assert that a state indeed represents a set of sets of formulas.

[8] These labels are identical up to BDD normalization.

It seems likely that states accessible through the same label have something in common, and that it may be worth trying to identify them. We only consider identical labels, as if we would consider a more relaxed constraint (for instance labels with a non empty intersection), this could create exponentially more new states. In practice, it seems that most of the time interesting compound states are created, and not too many of them.

Nevertheless, it may happen that some merged state in the automaton is subsumed by its components, existing as states on their own. Then the merged state is there superfluous and can be safely removed.

The initial state, put in DNF, can also play the role of a merged state. By the following theorem, it can be split as any other real merged state in order to reduce the overall number of states.

For merged states to be split back, we have to define the notion of subsumption.

Definition 2.14 *For any set of sets of formulas (not necessarily a actual state) S, we define what it means to be subsumed by states of $\mathcal{M}$, with the following predicate :*

$$\begin{aligned} &\text{Subsumed}(S) &&\Leftarrow \exists q \in Q.S = q \\ &\text{Subsumed}(S_1 \cup S_2) &&\Leftarrow \text{Subsumed}(S_1) \wedge \text{Subsumed}(S_2) \end{aligned}$$

Theorem 2.15 *Let us consider $q = q_1 \cup \ldots \cup q_n \in Q$. Assume* Subsumed$(q)$ *holds. Then the automaton $\mathcal{M}' = \langle AP, Q', Q'_0, E', \mathcal{F}' \rangle$ defined below :*

- $Q' = Q \setminus \{q\}$
- $Q'_0 = \begin{cases} (Q_0 \setminus \{q\}) \cup \{q_1\} \cup \ldots \cup \{q_n\} & \text{if } q \in Q_0 \\ Q_0 & \text{else} \end{cases}$
- $E' = E \setminus \{\langle q_{\text{src}}, l, q \rangle \in E\}$
- $\mathcal{F}' = \{\mathcal{F}air_\phi[False \mid q] \mid \mathcal{F}air_\phi \in \mathcal{F}\}$

is such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Proof (sketch) We remove a state q that is exactly subsumed by others as stated in definition 2.14, and redirect its incoming edges towards its components, which together recognize the same language as q. Henceforth $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

This theorem can be applied on-the-fly or later when the automaton is totally built. We chose to use it as soon as possible, in order to reduce the complexity of the post-simplification phase, but it may be worth postponing its use so as to process all merged states once and for all. Intermediate choices are currently being experimented too.

3 Post-simplification

As for automata post-simplification phase, we remove (all but one of) identical states, which is a pertaining step in all related works. Indeed, we haven't explored more general simulation relations, because they sometimes tend not to lend themselves to efficient computations.

Theorem 3.1 *Let us consider two states $q_1, q_2 \in Q$, with the same outgoing transition edges, i.e. such that for any $n = 1, 2$ and $\overline{n} = 3 - n$:*

$$\forall t_{\mathrm{n}} = \langle q_{\mathrm{n}}, l_{\mathrm{n}}, q' \rangle \in E. \exists t_{\overline{\mathrm{n}}} = \langle q_{\overline{\mathrm{n}}}, l_{\overline{\mathrm{n}}}, q' \rangle \in E.$$
$$l_{\mathrm{n}} = l_{\overline{\mathrm{n}}} \wedge \forall \mathcal{F}air_\phi \in \mathcal{F}. t_{\mathrm{n}}, q' \models \mathcal{F}air_\phi \;\Leftrightarrow\; t_{\overline{\mathrm{n}}}, q' \models \mathcal{F}air_\phi$$

Without loss of generality, we assume that whenever at least one of q_1 or q_2 is initial, then it is q_1. Then the automaton $\mathcal{M}' = \langle AP, Q', Q'_0, E', \mathcal{F}' \rangle$ defined below :

- $Q' = Q \setminus \{q_2\}$
- $Q'_0 = Q_0 \setminus \{q_2\}$
- $E' = \{\langle q_{\mathrm{src}}, l, q'_{\mathrm{dst}} \rangle \mid \langle q_{\mathrm{src}}, l, q_{\mathrm{dst}} \rangle \in E \wedge q'_{\mathrm{dst}} = q_{\mathrm{dst}}[q_1 \mid q_2]\}$
- $\mathcal{F}' = \{\mathcal{F}air_\phi[\langle q_1, l, q_{\mathrm{dst}} \rangle \mid \langle q_2, l, q_{\mathrm{dst}} \rangle \in E][q_1 \mid q_2] \mid \mathcal{F}air_\phi \in \mathcal{F}\}$ [9]

is such that $\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$.

Proof (sketch) This is a classical operation in automata theory.

4 Experiments

The following examples have all been tested on a 400MHz bi-pentiumII PC, with 256 Mo.

We include first some tests taken from [EH00,GO01] (see figure 1), showing the relative performances of Wring, LTL2BA and BAOM. For the sake of simplicity, we decided to present only results of the best available tools, because they constantly outperform other tools like SPIN for instance. Likewise, elapsed time is most of the time not shown, because all these tools usually achieve the translation within 5 seconds, which is largely acceptable. Yet, there exist pathological formulas such that time consumption is then exponential. In this case, we precisely show elapsed time or indicate that a crash had occurred or timeout (10 hours) had expired (†).

Then, we show some results for 3 collections of 1000 randomly generated formulas, processed with LTL2AUT, Wring and BAOM. Temporal and boolean operators are drawn with the same probability (see figure 2).

Finally, in order to (loosely) compare BAOM to LTL2BA, though we had only a restricted access to the LTL2BA tool via a web page, we use a testbench

[9] In the above theorem, the term $\mathcal{F}air_\phi[\langle q_1, l, q_{\mathrm{dst}} \rangle \mid \langle q_2, l, q_{\mathrm{dst}} \rangle \in E]$ means that the substitution is performed for all $\langle q_2, l, q_{\mathrm{dst}} \rangle \in E$.

for our tool generated with the same hypothesis as a similar testbench for LTL2BA presented in [GO01] (see figure 3). We have randomly drawn 1000 generated formulas with 10 nodes and 3 atoms.

Our prototype is entirely written in OCaML [DU], and thus time comparisons with other tools issued from similar works is hardly relevant due to the extreme diversity of implementation languages (and computers). For instance, the SPIN tool [Hol97] as well as the LTL2BA tool [GO01] and the LTL2AUT tool [DGV99] are written in C, whereas the EqLTL tool [EH00] is written in ML, and the Wring tool [SB00] in Perl.

Notice that transient memory requirements are not mentionned here due to impracticability of measures. Besides, because our transformations are almost all applied on-the-fly, memory consumption in our case is linearly bound to the size of the resulting automaton.

LTL formulas	states(time in seconds)		
	Wring	LTL2BA	BAOM
examples from [EH00]			
$p\mathsf{U}(q \wedge \Box r)$	3	2	2
$p\mathsf{U}(q \wedge \bigcirc(r\mathsf{U}s))$	5	3	3
$p\mathsf{U}(q \wedge \bigcirc(r \wedge (\Diamond(s \wedge \bigcirc\Diamond(t \wedge \bigcirc\Diamond(u \wedge \bigcirc\Diamond v))))))$	13	7	7
$\Diamond(p \wedge \bigcirc\Box q)$	3	2	2
$\Diamond(p \wedge \bigcirc(q \wedge \bigcirc\Diamond r))$	6	4	4
$\Diamond(q \wedge \bigcirc(p\mathsf{U}r))$	5	3	3
$\Diamond\Box p \vee \Diamond\Box q$	4	3	3
$\Box(p \rightarrow q\mathsf{U}r)$	3	2	2
$\Diamond(p \wedge \bigcirc\Diamond(q \wedge \bigcirc\Diamond(r \wedge \bigcirc\Diamond s)))$	9	5	5
$\bigwedge_{i=1\ldots5} \Box\Diamond p_i$	31(195)	1	1
$(p\mathsf{U}(q\mathsf{U}r)) \vee (q\mathsf{U}(r\mathsf{U}p))$	4	5	2
$(p\mathsf{U}(q\mathsf{U}r)) \vee (q\mathsf{U}(r\mathsf{U}p)) \vee (r\mathsf{U}(p\mathsf{U}q))$	4	7	2
$\Box(p \rightarrow q\mathsf{U}(\Box r \vee \Box s))$	4	4	4
examples from [GO01]			
$\neg((\bigwedge_{i=1\ldots10} \Box\Diamond p_i) \rightarrow \Box(q \rightarrow \Diamond r))$	†	2(36000)	2(44)
$\neg(p_1\mathsf{U}(p_2\mathsf{U}(\ldots\mathsf{U}p_8)\ldots)$	†	8(1200)	8

Fig. 1. Examples excerpt from [EH00,GO01].

	10 nodes 3 atoms		15 nodes 3 atoms		20 nodes 5 atoms	
method	states	time	states	time	states	time
LTL2AUT	6698	127s	11086	453s	25528	2740s
Wring	4043	203s	4830	534s	7748	1973s
BAOM	3026	3.45s	3318	6.5s	4723	40s

Fig. 2. Comparison between LTL2AUT, Wring and BAOM.

method	formulas	avg. time	max. time	avg. states	max. states
LTL2BA	200	0.01	0.04	4.51	39
BAOM	1000	0.003	0.09	3.06	16

Fig. 3. Loose comparison between LTL2BA and BAOM.

5 Conclusion

We have succeeded in devising an efficient algorithm, based upon syntactic considerations, with techniques designed to be used on-the-fly. This shows that a careful examination of parse trees of formulas can lead to similar or better results than *a posteriori* simulation-based methods. In real-life applications, efficiency is also due to the heavy use of BDDs in our data-structures, but this advantage doesn't really show up in our test cases due to the small number of atomic propositions. The main original factors of improvement over other similar tools are the introduction of finite d-prefixes in our revised expansion rules and also the fairness paradigm we have developed in conjunction with the syntactic implication between formulas. Yet, a more careful study of the relationship between values of d, shape of formulas and size of resulting automatas should obviously be carried out. As for the merging states techniques, it appears to have an impact in case of quasi-redundant or incompatible sub-formulas (this is often the case for randomly generated formulas), but don't usually come into play for short hand-written specifications.

Notice that the current implementation of our tool hasn't been specifically geared towards efficiency since it was developped in a purely functional setting (except for the BDD package). More efficient techniques such as hash-caching (as used in BDD algorithms) should be developped in order to deal with parse trees of very large formulas.

Indeed, for the time being the good ratio between more involved syntactic algorithms (being under examination) and practical efficiency is not clearly worked out, all the more because previous works mainly focused upon model-theoretic methods. We claim nevertheless that the syntactic level can offer

much more information, with a reasonable cost.

The benchmarks we have conducted tend to support this claim, though these tests mostly concern random formulas. In order to get interesting benchmarks, we would like to test only "sensible" specifications (i.e. used in industrial contexts for instance), which seems to be a delicate task, being known that a large database of such specifications is not yet available.

Acknowledgement

The author would like to thank kindly Robert de Simone for fruitful discussions and collaboration.

References

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10e20 states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Bou97] Amar Bouali. Xeve: an esterel verication environment (version v1.3). Technical Report RT-0214, INRIA Sophia-Antipolis, December 1997.

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *International Conference on Computer Aided Verification*, pages 249–260, 1999.

[DU] Documentation and User's. The objective caml system release 3.02.

[EH00] Kousha Etessami and Gerard J. Holzmann. Optimizing buchi automata. In *International Conference on Concurrency Theory*, pages 153–167, 2000.

[GO01] Paul Gastin and Denis Oddoux. Fast ltl to bchi automata translation, 2001.

[GPVW95] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient bchi automata from ltl formulae. In *International Conference on Computer Aided Verification*, pages 53–65, 2000.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

Liveness Checking as Safety Checking

Armin Biere, Cyrille Artho, Viktor Schuppan

Computer Systems Institute, ETH Zentrum RZ H, CH-8092 Zürich, Switzerland

Abstract

Temporal logic is widely used for specifying hardware and software systems. Typically two types of properties are distinguished, safety and liveness properties. While safety can easily be checked by reachability analysis, and many efficient checkers for safety properties exist, more sophisticated algorithms have always been considered to be necessary for checking liveness. In this paper we describe an efficient translation of liveness checking problems into safety checking problems. A counter example is detected by saving a previously visited state in an additional state recording component and checking a loop closing condition. The approach handles fairness and thus extends to full LTL.

1 Introduction

Model Checking [12] is one of the most successful approaches for verifying temporal specifications of hardware and software systems. System properties are specified in temporal logic [13] for which various formalisms exist. Typically two types of properties are distinguished, safety and liveness [19]. In practical applications, safety properties are prevalent. Therefore very efficient algorithms and tools have been devised for checking safety properties. Still the specification of most systems contains liveness parts. We describe a generic translation procedure that takes a system with a liveness specification and translates it into a new system, for which a safety property is valid iff the liveness property in the original system holds.

The main motivation is to enable existing tools and techniques to check liveness which were originally supposed to work on safety properties only. For instance sequential ATPG (automatic test pattern generation) [22] can be used to check simple classes of temporal formulae [3], but general liveness properties have been out of reach. The same applies to STE (symbolic trajectory evaluation) [26,9], though a generalized version of STE has been published that can handle all ω-regular properties [27]. Both technologies have been in use in industry for over a decade [22,6] and efficient implementations exist.

For symbolic model checking [21] there is a vast literature on optimizations which are only applicable to safety. Frontier set simplification [7], dense

[23] and prioritized [8] reachability analysis all try to speed up BDD-based reachability calculation, but have not been adapted to handle liveness so far.

Forward model checking [17,15,2] is an attempt to improve on backward based symbolic model checking by visiting reachable states only and catching bugs as early as possible. It is motivated by the observation that checking safety properties amounts to reachability analysis. Forward model checking tries to use forward image calculations exclusively. Since we are able to translate liveness into safety we expect to have the same benefits without changing the model checking algorithms.

Kupferman and Vardi have developed an approach to simplify automaton-based model checking of safety properties by searching for finite violating prefixes [18]. With our translation we follow a similar goal by reducing liveness properties to safety properties and thus enabling the application of a much wider range of verification algorithms.

Our translation is structural. It respects the hierarchy of the system and can easily be applied, even manually, on the design entry level, eg in a Hardware Description Language. This is particularly useful if a tool does not include other model checking algorithms beside safety checking, and, as it is usually the case in a commercial setting, there is no access to the source code.

The basic idea is borrowed from explicit on-the-fly model checking [14] and bounded model checking [1]: a counter example to a liveness property in a finite system is *lasso-shaped*, it consists of a prefix that leads to a loop. As in [1] the major challenge is how to detect the loop. In our translation a loop is found by saving a previously visited state and later checking whether the current state already occurred.

For simple liveness properties, the result of the translation is not much larger than the original model checking problem. We also show how to handle more complicated liveness properties, for instance involving the until operator. By adding fairness constraints the technique can be extended to full LTL.

The next section elaborates on various examples to establish an intuitive understanding of the translation. In Sect. 3 we introduce the necessary formal background. We precisely define the translation in Sect. 4, prove its correctness and compare the complexity of the original and the resulting model checking problems. In the same section we mention how to extend our translation to handle fairness and LTL. Preliminary experiments in Sect. 5 show the feasibility of our approach, and Sect. 6 concludes.

2 Intuition

A counter example trace to a simple liveness property $\mathbf{AF}p$ is an infinite path where the body p of the liveness property holds nowhere, or equivalently $\neg p$ holds along the whole path. Since we restrict our models to be finite, such a trace can always be assumed to be lasso-shaped as depicted in Fig. 1. It consists of a prefix that leads to a loop, starting at the loop state s_l. From

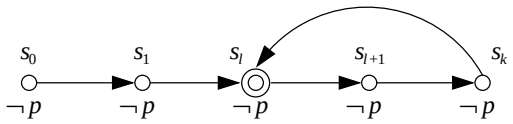


Fig. 1. A generic lasso-shaped counter example trace for $\mathbf{AF}p$.

every infinite trace in a finite model we can construct a lasso-shaped trace by closing the loop as soon as a state occurs the second time. Note that $\neg p$ still holds along the constructed path.

This observation is at the core of various model checking algorithms. Examples are explicit state algorithms for Büchi Automata [14] and unfolding liveness properties in bounded model checking [1]. With this restriction we only need to search for lasso-shaped counter examples. This is used in both translations discussed in this paper. The first translation, shown for illustration only, is called *counter based translation*. It extends the well known technique to check for *bounded liveness* only, but is not of practical value. Our main contribution is the *state recording translation*. It produces a state machine that may save at any time a previously reached state. Both translations do not only modify the property to be checked but also add additional checking components to the model while still maintaining bisimulation equivalence [12].

2.1 Counter Based Translation

In model checking applications it is often observed that a liveness property $\mathbf{AF}p$ can further be restricted by adding a bound k on the number of steps within which the body p has to hold. The bound is either given in the specification or may be determined by manual inspection. A bounded liveness property $\mathbf{AF}^k p$ is defined as

$$\mathbf{AF}^k p \equiv \mathbf{A}(p \vee \mathbf{X}p \vee \cdots \vee \mathbf{X}^k p), \text{ with } \mathbf{X}^i p \equiv \underbrace{\mathbf{X} \cdots \mathbf{X}}_{i-\text{times}} p \tag{1}$$

and clearly $\mathbf{AF}^k p$ implies $\mathbf{AF}p$. The reverse direction is also true if the bound is chosen large enough, in particular as large as the number of states $|S|$ in the model, since all states are reachable in $|S|$ steps.

A trivial translation would just exchange $\mathbf{AF}p$ by $\mathbf{AF}^k p$ with k the number of states. However, the expansion of $\mathbf{AF}^k p$ in (1) results in a very large formula, especially in the context of symbolic model checking. To avoid this explicit expansion, our counter based translation adds a counter to the model which counts the number of states reached so far. Now it only remains to check, whenever the counter reaches the number of states of the original model, that p was found to hold in at least one state reached so far. This latter property can be checked by attaching a boolean flag to the model that remembers whether p was satisfied in the past. This last step is property dependent.

As a first example we use a modulo 4 counter with initial state `0`. In Fig. 2 an SMV program [21] and a state graph of the counter are shown. While

```
MODULE main
VAR state : -1..3;
DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;
SPEC AF found
```

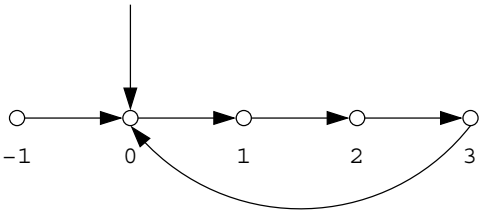


Fig. 2. A modulo 4 counter with unreachable state `-1`.

all states are reachable from `-1`, that state itself is unreachable because the counter wraps back to `0`.

The state space of the example encompasses five states. After five transitions we check whether `found` stayed false all the way from the initial state. In this case a counter example is found. Otherwise, the liveness property is valid, since every potential lasso-shaped trace of length 5 contains a state in which `found` holds.

This allows for a trivial translation of the liveness property into a safety property. The model is extended with a boolean variable `live`, which denotes whether `found` has already been true. A variable `counter` counts the number of states.

The left column of Fig. 3 shows the translated specification. The liveness property `AF found` translates into the safety property `AG (finished -> live)`. This translation is extremely inefficient, because it always requires traversing five states.

2.2 State Recording Translation

Instead of conservatively searching as long as required in the worst case, the search should terminate whenever a previously seen state s_l is traversed. Each time such a loop has been found, the liveness property p has to hold for at least one state visited before. Otherwise we have a counter example (see Fig. 1). Because state space traversal is memoryless, there is no way of explicitly expressing that property p must have been true at an earlier time as soon as we reach state s_l a second time.

The new model needs a way of "saving" a previously seen state for detecting a loop. Since we do not know beforehand whether we will see the current state again later, we use an oracle `save` that tells the model whether the current state is assumed to be the first state of a loop. To prevent overwriting the copy, another variable `saved` is used. After s_k, the last state of the loop, s_l is encountered again (see Fig. 4). At that time, the predicate `looped` becomes

```
MODULE main
VAR state   : -1..3;
    counter : 0..5;
    live    : boolean;


DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;

  init(counter) := 0;
  next(counter) :=
    case
      counter < 5 :
        counter + 1;
      1 : counter;
    esac;
  init(live) := 0;
  next(live) :=
    live | found;
DEFINE
  finished := counter = 5;
SPEC AG (finished -> live)
```

(a) counter

```
MODULE main
VAR state : -1..3;
    loop  : -1..3;
    live  : boolean;
    save  : boolean;
    saved : boolean;
DEFINE
  found := state = -1;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      state = 3 : 0;
      1 : state + 1;
    esac;
  init(saved) := 0;
  next(saved) :=
    saved | save;
  next(loop) :=
    case
      !saved & save : state;
      1 : loop;
    esac;
  init(live) := 0;
  next(live) := live | found;
DEFINE
  looped :=
    saved & state = loop;
SPEC AG (looped -> live)
```

(b) safe

Fig. 3. A counter based translation and our new translation of the liveness property.

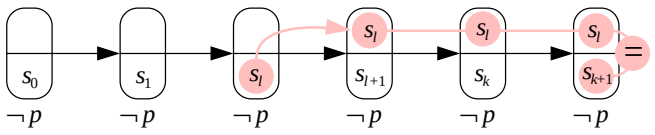


Fig. 4. Loop checking for $\mathbf{AF}p$ counter examples as reachability.

true, and property p must have been fulfilled at least once.

As visualized in Fig. 4, the loop closing condition `looped` checks, whether the current state has been visited earlier. Correspondingly Fig. 4 shows one more state than Fig. 1. Therefore `live` and `saved` should not refer to the current state. Their purpose is to remember whether `found` and respectively `save` were true in the past. In particular their initial value should be false.

2.3 Translation of fairness into safety

Fairness properties can also be translated using the same methodology. Figure 5 shows an example with two tasks t_0 and t_1 that count from 0 to 7 each. At each step, only one task is allowed to take its turn. The liveness property, stating that each task eventually arrives at state 7, can only be fulfilled if the turns are taken in a fair manner, i.e. each task eventually gets its turn.

In order to include fairness in our example, we define a new property `fair`. It records whether the fairness property is true at least once within each loop. Because `save` and `saved` are global, they are shared with the task modules.

3 Preliminaries

Our notation follows [12]. Let A be a set of *atomic propositions*. A *Kripke structure* K, or simply *model*, wrt. A is defined as $K = (S, I, T, L)$ with S a finite set of states, $I \subseteq S$ a set of initial states, $T \subseteq S \times S$ its transition relation, and $L : S \to 2^A$ a labelling function. We assume that the set of initial states is non-empty and the transition relation is total, i.e. for every state $s \in S$ there exists a state $s' \in S$ with $(s, s') \in T$. We write $T(s, s')$ whenever $(s, s') \in T$ and similarly $I(s)$.

As temporal logic we use a subset of CTL* with the next time operator **X**, and the eventuality operator **F**. We do not treat the until operator **U** or further operators in detail, since our translation works for full LTL which includes these operators. We will only consider the universal path quantifier **A**. The propositional operators are conjunction ($\wedge$) and negation ($\neg$). We also add the propositional constants $\{0, 1\}$.

The set of CTL* formulae is made of two types of formulae, *path formulae* Φ and *state formulae* Ψ. All atomic propositions $p \in A$ are state formulae, which can always be coerced to path formulae. Negation maintains the type of the argument. The same applies to conjunction if the types of the arguments match. Otherwise their conjunction is a path formula. Temporal operators are applied to state formulae. A path formula may be prefixed by a path quantifier to obtain a state formula.

Semantics for path formulae are defined wrt. *paths*, where the set Π of paths of K is the union of all finite and infinite sequences $\pi = (s_i)$ with $s_i \in S$ and $T(s_i, s_{i+1})$ for $0 \leq i < |\pi|$. The length $|\pi|$ of π is defined as the number of transitions. We only consider non-empty paths. We write $\pi(i)$ for s_i and π^n for the sequence (s_{i+n}), which is the same as the original path with its first n states removed. Let $p \in A$, $\phi, \phi_1, \phi_2 \in \Phi$ and $\psi, \psi_1, \psi_2 \in \Psi$. The validity of

```
MODULE task(id,turn)
VAR
  state : 0..7;


ASSIGN
  init(state) := 0;
  next(state) :=
    case
      turn = id &
      state < 7
        : state + 1;
      1 : state;
    esac;
DEFINE
  found := state = 7;

FAIRNESS
  turn = id




MODULE main
VAR
  turn  : 0..1;
  t0 : task(0,turn);
  t1 : task(1,turn);



DEFINE
  found :=
    t0.found &
    t1.found;




SPEC
  AF found
```

(a) live

```
MODULE task(id,turn,save,saved)
VAR
  state : 0..7;
  loop  : 0..7;
  fair  : boolean;
ASSIGN
  init(state) := 0;
  next(state) :=
    case
      turn = id &
      state < 7
        : state + 1;
      1 : state;
    esac;
DEFINE
  found := state = 7;
  looped := saved & state = loop;
ASSIGN
  init(fair) := 0;
  next(fair) := fair |
    id = turn & (save | saved);
  next(loop) :=
    case
      save & !saved : state;
      1 : loop;
    esac;

MODULE main
VAR
  turn  : 0..1;
  t0    : task(0,turn,save,saved);
  t1    : task(1,turn,save,saved);
  save  : boolean;
  saved : boolean;
  live  : boolean;
DEFINE
  found  := t0.found  & t1.found;
  looped := t0.looped & t1.looped;
  fair   := t0.fair   & t1.fair;
ASSIGN
  init(saved) := 0;
  next(saved) := saved | save;
  init(live)  := 0;
  next(live)  := live | found;
SPEC
  AG (looped & fair -> live)
```

(b) safe

Fig. 5. Hierarchical translation of liveness with fairness into pure safety.

state and path formulae for $s \in S$ and infinite $\pi \in \Pi$ are defined as follows:

$s \models \mathbf{A}\ \phi$	iff $\pi \models \phi$ for all $\pi \in \Pi$ with $\pi(0) = s$	$s \models p$	iff $p \in L(s)$
$s \models \psi_1 \wedge \psi_2$	iff $s \models \psi_1$ and $s \models \psi_2$	$s \models \neg\psi$	iff $s \not\models \psi$
$\pi \models \phi_1 \wedge \phi_2$	iff $\pi \models \phi_1$ and $\pi \models \phi_2$	$\pi \models \neg\phi$	iff $\pi \not\models \phi$
$\pi \models \mathbf{F}\ \phi$	iff there exists $i \geq 0$ with $\pi^i \models \phi$	$\pi \models \mathbf{X}\ \phi$	iff $\pi^1 \models \phi$
$\pi \models \psi$	iff $\pi(0) \models \psi$		

We will also use other boolean operators, such as disjunction ($\vee$) and implication ($\rightarrow$). The temporal operator globally $\mathbf{G}$ is defined as $\mathbf{G}\ \phi \equiv \neg\mathbf{F}\neg\phi$ and the existential path quantifier as $\mathbf{E}\ \phi \equiv \neg\mathbf{A}\neg\phi$.

A path π is *initialized* wrt. a given model $K = (S, I, T, L)$ iff $\pi(0) \in I$. Then a CTL* formula f is valid for K iff $\pi \models f$ for all initialized infinite paths π. Model checking determines the validity of f for K. Two model checking problems $P = (K, f)$ and $P' = (K', f')$ are equivalent iff $K \models f \ \Leftrightarrow K' \models f'$.

The first two steps of our translation in Sect. 4 produce equivalent model checking problems, proved by bisimulation equivalence. Two models $K = (S, I, T, L)$ and $K' = (S', I', T', L')$ over the same set of atomic propositions are bisimulation equivalent iff there exists a relation $\sim\ \subseteq S \times S'$ with the following properties: Let $s \in S$ and $s' \in S'$ with $s \sim s'$. First the labelling has to match, that is $L(s) = L'(s')$. Second for all $t \in S$ with $T(s, t)$ there has to exist $t' \in S'$ with $T'(s', t')$ and $t \sim t'$. Finally, for all initial states $s \in I$ there has to be an initial state $s' \in I'$ with $s \sim s'$. The dual properties have to hold as well.

The complexity of the original model checking algorithm [11] for simple properties, such as $\mathbf{AF}p$ and $\mathbf{AG}p$, is linear in the size of the model K. Particularly it is linear in the number of states $|S|$ and the number of transitions $|T|$. In the case of on-the-fly model checking [14] the complexity can further be restricted to be linear in the number of reachable states $|R|$ with $R = \{\pi(i) \mid i \geq 0, \pi \in \Pi, \pi \text{ initialized}\}$. For symbolic model checking with BDDs [21] the number of (reachable) states is less important than the number of fixpoint iterations. This number is bounded by the *diameter* d which is defined as the maximal *distance* $\delta(s, t)$ between two states $s, t \in S$ with

$$\delta(s, t) = \min \{\ k\ \mid\ \pi \in \Pi,\ \ |\pi| = k,\ \ \pi(0) = s,\ \ \pi(k) = t\}$$

In BFS reachability analysis the number of iterations can further be restricted to the maximal distance r, called *radius*, of all reachable states to some possibly varying initial state. In backward fixpoint computations, which are the traditional way of checking liveness properties, we can introduce a similar notion of a *backward radius* which is the number of backward iterations after which the fixpoint is reached. The backward radius depends not only on the model but also on the property. Note that backward and forward radius are

not related. For instance, an inductive invariant p has a backward radius of one when checking $\mathbf{AG}p$, independent of the size of the model. In practice pure backward model checking is usually outperformed by forward model checking [17] or a restricted version of backward model checking in which the approximations in the fixpoint computation are restricted to the pre-computed set of reachable states.

4 Translation

In this section we precisely describe our state recording translation on an abstract level and prove its correctness. The application to a concrete model description language such as the SMV input language used for the experiments is left to the reader. We also do not treat the counter based translation formally. In the second part of the section we discuss the efficiency of our translation by comparing size and diameter of the original and the translated model. In the last part we describe the extention to fairness and LTL.

4.1 Correctness

Let $K = (S, I, T, L)$ be a Kripke structure and $\mathbf{AF}p$ be the liveness property we want to check. As a first step we construct $K_\perp = (S_\perp, I_\perp, T_\perp, L_\perp)$, with $S_\perp = S \times (S \cup \{\perp\})$, and $I_\perp = I \times \{\perp\}$. The new transition relation is defined as

$$T_\perp((s,t),(s',t')) \quad \Leftrightarrow \quad T(s,s') \ \wedge \ (t' = t \ \vee \ (t = \perp \ \wedge \ t' = s)) \tag{2}$$

which operates on the first state component like the original transition relation. In the second state component a previously reached original state may be recorded, nondeterministically, but at most once (see also Fig. 4). Therefore $T_\perp$ is monotonic in the second state component for the order $\leq_\perp \subseteq (S \cup \{\perp\})^2$ with $s \leq_\perp t$ iff $s = t$ or $s = \perp$. The new labelling is obtained as $L_\perp = L \circ \rho$ using the projection function ρ operating on pairs with $\rho((s,t)) = s$.

We further assume that $\perp$ is a new state that does not already occur in S. In essence our translation simulates the original behavior of K without introducing dead ends, maintaining the labelling of the states. Therefore we can prove that K and $K_\perp$ are bisimulation equivalent under the bisimulation $\sim \subseteq S \times S_\perp$, with $s \sim s_\perp \Leftrightarrow \rho(s_\perp) = s$. To prove that $\sim$ is a bisimulation we use $\lambda_\perp \colon S \to S_\perp$ defined as $\lambda_\perp(s) = (s, \perp)$ and extend both $\lambda_\perp$ and the projection function ρ to paths in the natural way. Then we can easily check that $\pi \sim \lambda_\perp(\pi)$ and $\rho(\pi_\perp) \sim \pi_\perp$ for all paths. These functions provide the necessary witnesses for the existential quantifiers in the requirements for $\sim$ being a bisimulation.

Lemma 4.1 *K and $K_\perp$ are bisimulation equivalent.*

The next step adds a flag that remembers whether p has ever been valid on the path so far. The result is $K_p = (S_p, I_p, T_p, L_p)$ with $S_p = S_\perp \times \{0, 1\}$,

$I_p = I_\perp \times \{0\}$, and $T_p((s,x),(s',x'))$ iff

$$T_\perp(s,s') \ \wedge \ (p \in L_\perp(s) \ \rightarrow \ x' = 1) \ \wedge \ (p \notin L_\perp(s) \ \rightarrow \ x' = x)$$

The rest is defined as in the first step. Again T_p is monotonic in the second state component, in this case for the order of natural numbers restricted to $\{0,1\}$. Note, that K_p depends on the property being checked. Similar reasoning as before with a slightly more complex $\lambda_p : S_\perp \rightarrow S_p$ and a transitivity argument gives the following Lemma.

Lemma 4.2 *K and K_p are bisimulation equivalent.*

Since validity of CTL* formulae is preserved under bisimulation equivalence [4,12], we obtain the equivalence of $(K, \mathbf{AF}p)$ and $(K_p, \mathbf{AF}p)$. The final step in our translation consists of adding a new atomic proposition q with

$$q \in L_p(\ ((s,t),x)\ ) \quad \Leftrightarrow \quad s = t \ \rightarrow \ x = 1 \tag{3}$$

This definition shows the correctness of our translation.

Theorem 4.3 *$(K, \mathbf{AF}p)$ and $(K_p, \mathbf{AG}\ q)$ are equivalent.*

Proof. What remains to be shown is the equivalence of $\mathbf{EG}\neg p$ and $\mathbf{EF}\ \neg q$ in K_p. First assume $K_p \models \mathbf{EG}\neg p$. Then there exists an infinite initialized path $\pi \in \Pi_p$ with $p \notin L_p(\pi(i))$ for all $i \geq 0$. Since the number of states of S_p is finite, there have to exist indices $k \geq l \geq 0$ with $\pi(k+1) = \pi(l)$. Let $\pi(i) = (\ (s_i,t_i),x_i\ )$ for $i \geq 0$ and define $\pi'(i) = (\ (s_i,t'_i),x_i\ )$ with $t'_i = \perp$ for $0 \leq i \leq l$ and $t'_i = s_l$ for $l < i \leq k+1$.

Clearly π' is an initialized legal path of K_p. By definition we have $s_{k+1} = t'_{k+1} = s_l$ and $x_i = 0$ for $0 \leq i \leq k+1$, since $p \notin L_p(\pi'(j)) = L(s_j) = L_p(\pi(j))$ for $0 \leq j \leq k$. From (3) we get $q \notin L_p(\pi'(k+1))$ and π' proves to be a witness for $\mathbf{EF}\neg q$, assuming π' is extended to an infinite path in the obvious way. Note that T_p is total since our translation does not introduce dead ends.

For the reverse direction assume $\mathbf{EF}\neg q$ holds. Without loss of generality we find an initialized path $\pi \in \Pi_p$ with $|\pi| = k+1$ and $\pi(k+1) \models \neg q$. With $\pi(i) = (\ (s_i,t_i),x_i\ )$ we deduce from (3) that $s_{k+1} = t_{k+1}$ and $x_{k+1} = 0$. From the monotonicity of $T_\perp$ in its second state component, we obtain an l with $0 < l \leq k$, such that $\perp = t_0 = \ldots = t_l$ and $s_l = t_{l+1} = \ldots = t_{k+1}$. Now we construct an infinite path π' with $\pi'(i) = (\ (s'_i,t'_i),x_i\ )$ as follows: for $0 \leq i \leq k$ we simply set $\pi'(i) = \pi(i)$. If $i > k$ we define $t'_i = t_{k+1}$, $x'_i = x_{k+1}$ and $s'_i = s_{l+c}$ with $c = (i-l) \bmod (k+1-l)$. From the monotonicity of T_p in its second state component, we have $x_{k+1} = \ldots = x_0 = 0$, which implies $s_i \models \neg p$ for $0 \leq i \leq k$. Since these original states determine the non-validity of p for every $\pi'(i)$, and π' is a legal initialized infinite path, it serves as witness for $\mathbf{EG}\neg p$.□

4.2 Complexity

Our objective was to enable checking liveness properties with techniques and tools previously only used for reachability calculation or safety checking. The

impact of our translations on the complexity for model checking or reachability calculation is quite reasonable.

As sketched with the example of Fig. 5, the size of a non-canonical symbolic description in program code, increases only by a small constant factor. The counter based translation will produce very large counter examples. Therefore we restrict the discussion to the state recording translation.

In global (explicit) model checking [11] the complexity is governed by the number of states, which increases quadratically:

$$|S_p| = 2 \cdot |S_\perp| = 2 \cdot |S| \cdot (|S| + 1) = O(|S|^2)$$

In the case of on-the-fly (explicit) model checking [14] only the size of the reachable state space R_p is of interest. A reachable state $(s,t) \in R_\perp$ of $K_\perp$ either contains $\perp$ as second component t, or t is reachable in K since only reachable states are recorded. Therefore $R_\perp$ is bounded by $|R| \cdot (|R| + 1)$. This bound is tight: a modulo n counter, like the model in Fig. 2 for $n = 4$, has $|R_\perp| = n \cdot (n + 1)$ reachable states. If $n = 4$ then every combination of $\{0, \ldots, 3\} \times \{\perp, 0, \ldots 3\}$ can be reached. Further introducing the p-recording flag at most doubles the number:

$$|R_p| \leq 2 \cdot |R_\perp| \leq 2 \cdot |R| \cdot (|R| + 1) = O(|R|^2)$$

Regarding symbolic model checking with BDDs [21] we have two results. First we relate the size of reduced ordered BDDs for the transition relation of K, $K_\perp$ and K_p. Assuming S is encoded with $n = \lceil \log_2 |S| \rceil$ state bits, we can encode $S_\perp$ with $2n + 1$ boolean variables. It is important to interleave the boolean variables for the first and second component. Otherwise the size of the BDD for the term $(t' = t \vee (t = \perp \wedge t' = s))$ in (2) may explode. With an interleaved order it is linear in n with a factor of approx. 11. The factor has been determined empirically for large state spaces. Thus the size of the BDD for $T_\perp$ can be bounded by $11 \cdot n$ the size of the BDD for T by using the fact from [5] that computing any boolean binary operation on BDDs will produce a BDD of size that is linear with factor 1 in the size of the argument BDDs. Finally, the size of the BDD for T_p compared to the size of the BDD for $T_\perp$ may increase by a linear factor in the size of the BDD representing the set of states in which p holds, which in practice is usually very small.

Similar calculations for the set of initial states show that the size of BDDs representing K_p can be bound to be linear in the size of the BDDs representing K, linear in the number of state bits, and linear in the size of the BDD representing the set of states in which p holds. These *static* bounds do not say anything about the size of the BDDs in the fixpoint iterations. To measure the *dynamic* complexity we determine bounds on the diameter and radius, which also serve as bounds on the maximal number of fixpoint iterations. Note that the counter based translation has a radius at least as large as the number of states in the original system, which makes traditional symbolic reachability

analysis impractical even for medium sized problems. One important observation is that the state recording translation produces a much smaller diameter d_p and radius r_p:

Theorem 4.4 $d_p \leq 4 \cdot d + 3$ *and* $r_p \leq r + 3 \cdot d + 3$

Proof. Let $\pi \in \Pi_\perp$ be a finite path with $\pi(i) = (s_i, t_i)$ and $|\pi| = k$. Since $T_\perp$ is monotonic in the second component we have to distinguish two cases. If first $t_0 = \ldots = t_k$, then $\delta_\perp(\pi(0), \pi(k)) = \delta(s_0, s_k) \leq d$, since all paths in K can be extended to legal paths in $K_\perp$ by adding a fixed non changing second state component. In the second case there exists an l with $0 \leq l < k$ with $t_0 = \ldots = t_l = \perp$ and $t_{l+1} = \ldots = t_k = s_l$ (cf Fig. 4). Now we have two sub-paths with constant second state component as in the first case and obtain

$$\delta_\perp(\pi(0), \pi(k)) \;\leq\; \delta(s_0, s_l) + 1 + \delta(s_{l+1}, s_k) \;\leq\; 2 \cdot d + 1$$

which also subsumes the bound of the first case and thus $d_\perp \leq 2 \cdot d + 1$. To determine the bound for the radius we additionally assume that π is initialized. Then $\delta(s_0, s_l) \leq r$ and we obtain $r_\perp \leq r + d + 1$. With the same reasoning, since T_p is monotonic in the second state component as well, we derive $d_p \leq 2 \cdot d_\perp + 1$ and $r_p \leq r_\perp + d_\perp + 1$. By substitution we derive the desired inequalities. □

Unfortunately, there are examples where r is much smaller than d and for reachability analysis in K_p we still have to perform more than d fix point iterations. A modulo n counter as in Fig. 2 without the `-1` state becomes such an example if we allow all states to be initial states. Then we have $d = n - 1$, $r = 0$, but $d_\perp = 2 \cdot n - 1$ and $r_\perp = n$, which is already larger than d. The number of backward iterations necessary to check a liveness property in the original model could also be very large.

4.3 Fairness and LTL

Our translation is able to incorporate fairness. A fairness constraint is simply a subset of S. A path π is called *fair* wrt. *one* fairness constraint $F^i \subseteq S$ iff some state in F^i occurs infinitely often on π. If π is fair, then π is infinite, written $|\pi| = \infty$. Formally we add a fifth component F to a model, where F is a possibly empty list of fairness constraints $F = (F^1, \ldots, F^m)$. Then a path is fair for K iff it is fair wrt. every F^i. The semantics of models with fairness constraints is defined as in the unfair case, except that all paths are required to be fair. Bisimulation with fairness is defined by expanding the transition based definition stated above to whole fair paths as in [12]: the additional requirement is that for all fair paths $\pi \in \Pi$ there exists a fair path $\pi' \in \Pi'$ with $\pi \sim \pi'$, where $\pi \sim \pi'$ iff $\pi(i) \sim \pi'(i)$ for all $i \geq 0$. To handle a fair Kripke structure $K(S, I, T, L, F)$ we construct $K_p(S_p, I_p, T_p, L_p, F_p)$ where S_p, I_p, T_p, and L_p are defined as above and F is extended to

$$F_p = (F^1 \times (S \cup \{\perp\}) \times \{0, 1\}, \ldots, F^m \times (S \cup \{\perp\}) \times \{0, 1\}).$$

We define $K_p^F = (S_p^F, I_p^F, T_p^F, L_p^F)$ with $S_p^F = S_p \times \{0,1\}^m$ and $I_p^F = I_p \times \{(0,\ldots,0)\}$ by replacing each fairness constraint F^i with a state bit that remembers whether a loop state in F^i has been reached. Let L_p^F be the natural extension of L_p as before. Let (s,t,x,v), $(s',t',x',v') \in S_p^F$ with $s,s' \in S$, $t,t' \in S \cup \{\bot\}$, $x,x' \in \{0,1\}$ and $v,v' \in \{0,1\}^m$. The transition relation T_p^F is satisfied for (s,t,x,v) and (s',t',x',v') as current and next state iff

$$T_p(\ ((s,t),x)\ ,\ ((s',t'),x')\ ) \wedge$$

$$\textstyle\bigwedge_{i=1}^{m}\ (v'(i) = v(i)\ \vee\ (t' \neq \bot\ \wedge\ s \in F^i\ \wedge\ v'(i) = 1))$$

which is again monotonic in the new fairness components of the state space. We further add a new atomic proposition q_F with

$$q_F \in L_p^F\ (\ ((s,t,x,v)\ )\ \Leftrightarrow\ (v(1) = \ldots = v(m) = 1)\ \rightarrow\ q \in L_p((s,t),x)$$

where q is defined as for K_p. We can prove a correctness result like before, now including fairness.

Theorem 4.5 *$(K, \mathbf{AF}p)$ and $(K_p^F, \mathbf{AG}\ q_F)$ are equivalent.*

The number of added state bits grows linearly in the number m of fairness constraints. This directly corresponds to the increase in size of the input for symbolic model checking. The state space K_p^F itself grows exponentially. So does the diameter and the radius. The approach seems to be feasible, at least for explicit model checking, only for a small number of fairness constraints. However, checking $\mathbf{AG}\ q_F$ will always find shortest counter examples.

An alternative approach counts the number of fairness constraints satisfied sofar, similar to the well known translation of generalized Büchi automata into ordinary Büchi automata. It produces a liveness property with a single fairness constraint, which in turn is translated into a safety property. This approach is more space efficient. It requires only logarithmic additional state bits. However it fails to generate counter example traces of minimal length. In addition, it is not clear how this *binary* encoding performs for symbolic model checking versus the *one-hot* encoding discussed before.

Since generalized Büchi automata and thus LTL [14] can be translated into fair Kripke structures, our translation also applies to LTL model checking in general. Additionally it is possible to derive special translation rules for other standard LTL operators. For example to handle $p_1 \mathbf{U}\ p_2$ we use

$$p_1 \mathbf{U}\ p_2\ \equiv\ (p_1\ \mathbf{U}_{\text{weak}}\ p_2)\ \wedge\ \mathbf{F}p_2$$

where the *weak until* operator $p_1\ \mathbf{U}_{\text{weak}}\ p_2$ is defined to be valid for a path iff p_1 does not stop to hold before p_2 holds or p_1 holds along the whole path. By adding a state bit that remembers whether p_2 was fulfilled already, the weak until can easily be transformed into a simple safety property. Then the

	check true			check false			counterexample false		
n	live	count	safe	live	count	safe	live	count	safe
4	8 (4+ 4)	9 (9+0)	8 (8+0)	5 (4+1)	5 (5+0)	4 (4+0)	7 (3+ 4)	5 (0+5)	4 (0+4)
8	16 (8+ 8)	17(17+0)	16(16+0)	9 (8+1)	9 (9+0)	8 (8+0)	15 (7+ 8)	9 (0+9)	8 (0+8)
12	24(12+12)	25(25+0)	24(24+0)	13(12+1)	13(13+0)	12(12+0)	23(11+12)	13(0+13)	12(0+12)
16	32(16+16)	33(33+0)	32(32+0)	17(16+1)	17(17+0)	16(16+0)	31(15+16)	17(0+17)	16(0+16)

Table 1
Counters

eventuality $\mathbf{F}p_2$ is translated into a safety property as well, with our original translation. Finally, we check both safety properties simultaneously.

5 Experiments

In this section, we show the results of our translation applied to various examples, both theoretical and "real world" ones. Each table is divided into three main parts: the left part, with the iterations needed for the correct model, the middle part, where the model is incorrect, and the right part, which shows the iterations needed to compute a counter example for the incorrect model. The three main parts are further split up into one column for each different approach: *live* for the conventional liveness approach, *count* for the counter based approach (not used in the FireWire example), and *safe* for our state recording translation. For each version, the number of overall, forward, and reverse iterations is shown.

5.1 Simple Counters

In the case of a simple counter in Table 1 all approaches perform linearly in the number of iterations wrt. the model size. Computing the counter example, however, requires nearly twice as many iterations with the live version as opposed to our method.

For the counters used in Table 2 the desired state `n` can be reached from any state in one step. There are only two iterations needed to complete a loop, and n backward iterations to reach all possible predecessors. With the counter based approach, $n + 1$ iterations are required to enumerate enough states, and another iteration to reach state `n`. Our approach requires a constant number of five iterations for a correct model: One iteration to reach all possible successor states; from those states, a second iteration to reach state `n`. The third iteration reaches the initial state `0` again, from which two more iterations are required to prove the liveness within the loop.

The false example requires two iterations for the loop, and with the live version, another backward iteration for the initial state as a predecessor. The counter based approach is very inefficient. The counter example analysis shows a similar behavior.

	check true			check false			counterexample false		
n	live	count	safe	live	count	safe	live	count	safe
4	6 (2+ 4)	6 (6+0)	5 (5+0)	3 (2+1)	5 (5+0)	2 (2+0)	3 (1+2)	5 (0+5)	2 (0+2)
8	10 (2+ 8)	10 (10+0)	5 (5+0)	3 (2+1)	9 (9+0)	2 (2+0)	3 (1+2)	9 (0+9)	2 (0+2)
12	14 (2+12)	14 (14+0)	5 (5+0)	3 (2+1)	13 (13+0)	2 (2+0)	3 (1+2)	13 (0+13)	2 (0+2)
16	18 (2+16)	18 (18+0)	5 (5+0)	3 (2+1)	17 (17+0)	2 (2+0)	3 (1+2)	17 (0+17)	2 (0+2)

Table 2
Skipping Counters

5.2 IEEE 1394 FireWire – Tree Identify Protocol

IEEE 1394 (FireWire) [16] is a protocol for a serial high-speed bus widely used to interconnect multimedia devices and PCs. To ensure correct functioning of the protocol the nodes connected to an IEEE 1394 bus are required to form a tree. The Tree Identify Protocol is executed each time the bus configuration changes to verify this condition and to elect a unique leader who has extended responsibilities in later phases of the protocol. In previous work [25,24] we have verified several properties of the Tree Identify Protocol with SMV.

The single most important property to be verified in the tree identify phase is the designation of a leader before the next phase of the protocol is reached. This property was checked in our experiments for both the original (correct) version of the model from [24] and a version with a bug preventing the successful completion of the protocol. In the SMV input language it is formulated for 2 nodes as follows

```
AF (node[0].root | node[1].root | timeout | known_problems)
```

where `root`, `timeout` and `known_problems` are state properties. Separate safety properties are used to ensure that neither `timeout` nor `known_problems` have occurred. Once verified these conditions could be removed from the model and are not included in the performance figures given here.

During the run of the protocol two nodes might be left competing to become root. In this case a sub-protocol is invoked to resolve this situation, called root contention. Both contending nodes non-deterministically choose to wait for either a short or a long time before continuing. If the nodes chose differently one of them will become root. Otherwise the sub-protocol is repeated. A fairness condition ensures that the two nodes will make a different choice at some point.

Most of the steps in the translation process described in Sect. 4 have been automated. For the translation a flat model is generated with NuSMV [10]. Additional variables are introduced to record the saved state, to represent the oracle, and to keep track whether each fairness condition has been true on the loop. Simple liveness properties of the form `AF p` are also translated automatically. More complicated properties need to be reformulated by the user either by using the automata based approach or by simple transformations

		check true		check false		cex false	
n	p	live	safe	live	safe	live	safe
2	2	74 (19 + 55)	24 (24 + 0)	34 (19 + 15)	13 (13 + 0)	132 (13 + 119)	13 (0 + 13)
2	3	74 (19 + 55)	24 (24 + 0)	35 (19 + 16)	13 (13 + 0)	132 (13 + 119)	13 (0 + 13)
2	4	78 (19 + 59)	24 (24 + 0)	36 (19 + 17)	13 (13 + 0)	132 (13 + 119)	13 (0 + 13)
3	2	76 (21 + 55)	23 (23 + 0)	36 (21 + 15)	11 (11 + 0)	67 (10 + 57)	11 (0 + 11)
3	3	77 (21 + 56)	23 (23 + 0)	37 (21 + 16)	11 (11 + 0)	67 (10 + 57)	11 (0 + 11)
3	4	77 (21 + 56)	23 (23 + 0)	37 (21 + 16)	11 (11 + 0)	67 (10 + 57)	11 (0 + 11)
4	2	129 (31 + 98)	36 (36 + 0)	52 (31 + 21)	19 (19 + 0)	215 (19 + 196)	19 (0 + 19)

Table 3
Leader election in the Tree Identify Protocol - iterations

		check true				check + cex false			
		live		safe		live		safe	
n	p	time	memory	time	memory	time	memory	time	memory
2	2	0.85	66941	4.19	397030	1.12	103299	2.64	282859
2	3	1.93	201680	11.07	782574	2.65	215169	6.82	595756
2	4	4.71	443947	28.22	1296088	5.45	402535	16.00	944482
3	2	11.33	699222	39.45	1946866	7.59	718910	12.09	772508
3	3	76.05	3777278	283.07	9578242	53.60	3678676	86.82	4217925
3	4	450.72	29220542	1567.67	31759998	259.51	19588279	554.39	14364650
4	2	357.30	14001693	1376.18	35547502	204.82	12500473	644.18	24864717

Table 4
Leader election in the Tree Identify Protocol

similar to the one we presented for the until operator in Sect. 4.3. Finally, an improved variable order is generated. To allow for a fair comparison the live model was also flattened before checking.

We used Cadence SMV [20] on a Pentium III-800 running Linux 2.2.19. Execution time and memory usage were limited to 1 hour and 1 GB respectively. Since an optimized variable order was provided explicitly, dynamic reordering had been disabled. In separate runs we checked that dynamic reordering produces comparable orders. Note that, enabling dynamic reordering would have increased runtimes dramatically.

Configurations with 2 – 4 nodes and 2 – 4 ports were checked. Table 3 shows the number of iterations. Table 4 lists execution time in seconds and memory usage in peak number of BDD nodes. Combinations of nodes and ports not shown could not be handled within the given time and memory bounds.

In each case, the safe version requires much fewer overall iterations than the live version. Only for the correct model the safe version needs more forward iterations than the live version. While run time and memory usage for the safe version of the correct model is up to 6 times higher than for the live version, the relation improves in the buggy case.

6 Conclusion

In this paper we presented a translation that allows to check liveness properties by checking safety properties. Our main contributions can be summarized as follows:

(i) For commercial or proprietary safety checking tools it may not be feasible for the user to change the algorithms. Our technique allows to apply such tools to liveness, which were supposed to check safety properties only.

(ii) The experiments indicate that our technique is comparable with specialized algorithms. Additionally we are able to find counter example traces of minimal length.

(iii) With our translation theoretical results on safety checking can be lifted to liveness checking. Therefore special treatment of liveness properties can only be justified by experiments or additional complexity results.

The main open question is how the number of state bits introduced by our translation can further be reduced. We also want to apply the method to liveness checking with sequential ATPG and STE.

References

[1] Biere, A., A. Cimatti, E. Clarke and Y. Zhu, *Symbolic model checking without BDDs*, in: *TACAS*, 99.

[2] Biere, A., E. Clarke and Y. Zhu, *Multiple state and single state tableaux for combining local and global model checking*, in: *Correct System Design (Recent Insights and Advances)*, number 1710 in LNCS, 2000.

[3] Boppana, V., S. Rajan, K. Takayama and M. Fujita, *Model checking based on sequential ATPG*, in: *CAV*, 99.

[4] Browne, M., E. Clarke and O. Grumberg, *Characterizing finite Kripke structures in propositional logic*, Theoretical Computer Science **59** (1988).

[5] Bryant, R., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **35** (1986).

[6] Bryant, R. and C.-J. Seger, *Formal verification of digital circuits using symbolic ternary system models*, in: *CAV*, 1990.

[7] Burch, J., E. Clarke, D. Long, K. McMillan and D. Dill, *Symbolic model checking for sequential circuit verification*, IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems **13** (1994).

[8] Cabodi, G., P. Camurati and S. Quer, *Improved reachability analysis of large finite state machines*, in: *ICCAD*, 1996.

[9] Chou, C.-T., *The mathematical foundation of symbolic trajectory evaluation*, in: *CAV*, 1999.

[10] Cimatti, A., E. Clarke, F. Giunchiglia and M. Roveri, *NuSMV: a new symbolic model verifier*, in: *CAV*, 99.

[11] Clarke, E. and A. Emerson, *Design and synthesis of synchronization skeletons using branching time temporal logic*, in: *IBM Workshop on Logics of Programs*, 1981.

[12] Clarke, E., O. Grumberg and D. Peled, "Model Checking," MIT Press, 1999.

[13] Emerson, A., *Temporal and modal logic*, in: *Handbook Theoretical Computer Science: Volume B, Formal Methods and Semantics* (1995).

[14] Gerth, R., D. Peled, M. Vardi and P. Wolper, *Simple on-the-fly automatic verification of linear temporal logic*, in: *15th Workshop on Protocol Specification, Testing, and Verification* (1995).

[15] Henzinger, T., O. Kupferman and S. Qadeer, *From pre-historic to post-modern symbolic model checking*, in: *CAV*, 1998.

[16] IEEE, "IEEE Standard for a High Performance Serial Bus. Std 1394-1995, and Supplement 1394a-2000," (1995, 2000).

[17] Iwashita, H. and T. Nakata, *CTL model checking based on forward state traversal*, in: *ICCAD*, 1996.

[18] Kupferman, O. and M. Vardi, *Model checking of safety properties*, in: *CAV*, 1999.

[19] Lamport, L., *Proving the correctness of multiprocess programs*, IEEE Transactions on Software Engineering **3** (1977).

[20] McMillan, K., *Cadence SMV*, @ `http://www-cad.eecs.berkeley.edu/~kenmcmil/smv`.

[21] McMillan, K., "Symbolic Model Checking: An Approach to the State Explosion Problem," Kluwer Academic Publishers, 1993.

[22] Niermann, T. and J. Patel, *Hitec: A test generation package for sequential circuits*, in: *EURODAC*, 1991.

[23] Ravi, K. and F. Somenzi, *High density reachability analysis*, in: *ICCAD*, 1995.

[24] Schuppan, V. and A. Biere, *Verifying the IEEE 1394 FireWire Tree Identify Protocol with SMV* Submitted.

[25] Schuppan, V. and A. Biere, *A simple verification of the Tree Identify Protocol with SMV*, in: *IEEE 1394 (FireWire) Workshop*, 2001.

[26] Seger, C.-J. and R. Bryant, *Formal verification by symbolic evaluation of partially-ordered trajectories*, Formal Methods in System Design **6** (1995).

[27] Yang, J. and C.-J. Seger, *Introduction to generalized symbolic trajectory evaluation*, in: *ICCD*, 2001.

Stuttering-Insensitive Automata for On-the-fly Detection of Livelock Properties

Henri Hansen [1], Wojciech Penczek [2], Antti Valmari [3]

[1,3] *Tampere University of Technology,*
Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND

[2] *Institute of Computer Science, PAS*
Ordona 21, 01-237 Warsaw, POLAND

Abstract

The research examines liveness and progress properties of concurrent systems and their on-the-fly verification. An alternative formalism to Büchi automata, called *testing automata*, is developed. The basic idea of testing automata is to observe *changes* in the values of state propositions instead of the values. Therefore, the testing automata are able to accept only stuttering-insensitive languages. Testing automata can accept the same stuttering-insensitive languages as (state-labelled) Büchi automata, and they have at most the same number of states. They are also more often deterministic. Moreover, on-the-fly verification using testing automata can often (but not always) use an algorithm performing only one search in the state space, whereas on-the-fly verification with Büchi automata requires two searches. Experimental results illustrating the benefits of testing automata are presented.

1 Introduction

In this research we examine liveness and progress properties (see e.g. [11, Chapter 4.2]) of concurrent systems and their on-the-fly verification. On-the-fly verification has the significant benefit that the analysis of an erroneous behaviour is possible when only a fragment of the state space has been generated. This is widely known, and is supported by the measurements in Section 5. The most well-known general-purpose algorithm suitable for on-the-fly

[1] Email: hansen@cs.tut.fi
[2] Email: penczek@ipipan.waw.pl
[3] Email: ava@cs.tut.fi

verification is presented in [3]. It is based on a double search of the state space. The property under inspection is often expressed as a Büchi automaton. From the point of view of the algorithm, the local state of the Büchi automaton is a part of the global state, and as such has an impact on the total number of reachable states.

The basis of this research is the observation that a significant proper subset of liveness properties can be verified by an alternative on-the-fly algorithm. The algorithm does only one search in the state space. It is described as Algorithm 3.4 in [17] but we also present it in Section 3.3. It searches for cycles that represent non-progress. (Another algorithm is the one in [8, pp. 235–237], but unlike the algorithm in [17], it requires a double state space search.) When the verified property can be expressed in a form suitable for the single-search algorithm, the algorithm has a tendency to find an error sooner than the algorithm of [3], as the measurements in Section 5 show.

In this research we develop an alternative formalism to Büchi automata, called *testing automata*, which makes it possible to use the algorithm of [17] in many verification tasks. The basic idea of testing automata is to observe *changes* in the values of state propositions instead of the values. Because of this, the testing automata are able to accept only stuttering-insensitive languages. They can accept the same stuttering-insensitive languages as Büchi automata. They never need more states than state-labelled Büchi automata. Deterministic testing automata accept strictly more stuttering-insensitive languages than Büchi automata.

Many verification researchers find limiting oneself to stuttering-insensitive languages not a serious disadvantage. It may even be seen as a benefit [5,10]. For example, the results in [13] would have been easier to derive, if the authors did not have to bother with the fact that even when the language accepted by a Büchi automaton is stuttering-insensitive, individual local states may be sensitive to stuttering. Confining to stuttering-insensitive languages also expands the possibilities to reduce the automaton before use.

In Section 2 Büchi automata and how they are used in verification are presented. In Section 3 the same is done for testing automata. In Section 4, the relationship between Büchi automata and testing automata is explored. Among other things, a construction is given to transform a Büchi automaton that accepts a stuttering-insensitive language into a testing automaton. In Section 5 some experimental results are given illustrating the benefits of the algorithm in [17].

2 Büchi automata

We will use Büchi automata whose states are labelled rather than the transitions. The intended interpretation is that the automaton has a set of propositions whose truth values depend on the state, and the label of a state indicates the propositions that evaluate to True in that state. Translations between

state-labelled and transition-labelled Büchi automata are straightforward, see [12].

2.1 Definitions

A *Büchi automaton* is a 6-tuple

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}})$$

where

- S is a finite set. Its elements are called *states*.
- Π is a finite set. Its elements are called *propositions*.
- val is a function from S to 2^{Π}. Its elements are called *valuations*.
- $\Delta \subseteq S \times S$. Its elements are called *transitions*.
- $\hat{S} \subseteq S$. Its elements are called *initial states*.
- $F_{\mathsf{inf}} \subseteq S$. Its elements are traditionally called acceptance states but to avoid confusion later, we call them *infinite acceptance states*.

From now on, let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}})$ be a Büchi automaton.

A *run* of $\mathcal{B}$ is an infinite sequence $s_0 s_1 s_2 \cdots \in S^\omega$ such that

- $s_0 \in \hat{S}$,
- $\forall i : (s_i, s_{i+1}) \in \Delta$.

A run is *accepting* if and only if $s_i \in F_{\mathsf{inf}}$ holds for infinitely many values of i.

The *language* $\mathcal{L}(\mathcal{B})$ accepted by the Büchi automaton is the set of infinite sequences $P_0 P_1 P_2 \cdots$ s.t. there is an accepting run $s_0 s_1 s_2 \cdots$, where $P_i = val(s_i)$ for $i \geq 0$.

We say that a language $\mathcal{L}$ is *stuttering-insensitive* iff $P_0 P_1 P_2 \cdots \in \mathcal{L} \Leftrightarrow P_0^{i_0} P_1^{i_1} P_2^{i_2} \cdots \in \mathcal{L}$ for every $i_0 > 0, i_1 > 0, \ldots$. Here X^i denotes a string that consists of i copies of X. A Büchi automaton is stuttering-insensitive iff the language it accepts is so.

As we have seen already, for a Büchi automaton $\mathcal{B}$, the language $\mathcal{L}(\mathcal{B}) \subseteq (2^{\Pi})^\omega$. For certain purposes it is useful to extend the Büchi automata formalism so that they can accept languages $\mathcal{L} \subseteq (2^{\Pi})^\omega \cup (2^{\Pi})^*$. Therefore, we will discuss two additional kinds of acceptance states, $F_{\mathsf{fin}} \subseteq S$ and $F_{\mathsf{dl}} \subseteq S$ called *finite* and *deadlock acceptance* states, respectively. Let

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$$

be a Büchi automaton extended in this way. A *run* is defined like above, except that now it may also be finite.

An infinite sequence $P_0 P_1 P_2 \cdots$ is *accepted* if at least one of the two conditions below holds:

(i) $\mathcal{B}$ has an infinite run $s_0 s_1 s_2 \cdots$ such that $\forall i : val(s_i) = P_i$, and $s_0 s_1 s_2 \cdots$

is accepting in the above-defined sense (i.e. $s_i \in F_{\mathsf{inf}}$ for infinitely many i).

(ii) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_k$ for some $k \geq 0$ such that $\forall i \leq k : val(s_i) = P_i$, and $s_k \in F_{\mathsf{fin}}$.

A finite sequence $P_0 P_1 P_2 \cdots P_n$ is *accepted* if at least one of the two conditions below hold:

(iii) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_k$, for some $k \leq n$ such that $\forall i \leq k : val(s_i) = P_i$, and $s_k \in F_{\mathsf{fin}}$.

(iv) $\mathcal{B}$ has a finite run $s_0 s_1 s_2 \cdots s_n$ such that $\forall i \leq n : val(s_i) = P_i$ and $s_n \in F_{\mathsf{dl}}$.

It turns out that the set F_{fin} does not increase the accepting power of Büchi automata, but it has other benefits in verification. The F_{fin}-acceptance can find counterexamples at least as fast as other methods and detection of such counterexamples can be trivially integrated to other methods. It is also sometimes easier or more natural to express properties directly using F_{fin} states than first encoding them as LTL formulas.

Theorem 2.1 *For every Büchi automaton with $F_{\mathsf{fin}} \neq \emptyset$ there is a Büchi automaton with $F_{\mathsf{fin}} = \emptyset$ that accepts the same language.*

Proof. Let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a Büchi automaton, with $F_{\mathsf{fin}} \neq \emptyset$. We construct another automaton $\mathcal{B}' = (S', \Pi, val', \Delta', \hat{S}', F'_{\mathsf{inf}}, \emptyset, F'_{\mathsf{dl}})$ that accepts the same language.

In the construction, we create a clique of states that are all both deadlock and infinite acceptance states, redirect all transitions leading to a finite acceptance state into a state of the clique with the same valuation as the finite acceptance state, and remove all finite acceptance states. This does not affect runs that do not visit such states, and any run that does is accepting in both the automata by definition.

- $S_1 = S - F_{\mathsf{fin}}$, $S_2 = 2^{\Pi}$, and $S' = S_1 \cup S_2$.
- $\Delta_1 = \Delta \cap (S_1 \times S_1)$, $\Delta_2 = \{ (s, val(s')) \mid (s, s') \in \Delta \wedge s \in S_1 \wedge s' \in F_{\mathsf{fin}} \}$ and $\Delta_3 = S_2 \times S_2$, and $\Delta' = \Delta_1 \cup \Delta_2 \cup \Delta_3$.
- $val'(s) = val(s)$ whenever $s \in S_1$, and $val'(P) = P$ when $P \in S_2$.
- $\hat{S}' = (\hat{S} \cap S_1) \cup \{ P \mid \exists s \in F_{\mathsf{fin}} \cap \hat{S} : val(s) = P \}$.
- $F'_{\mathsf{inf}} = (F_{\mathsf{inf}} \cap S_1) \cup S_2$.
- $F'_{\mathsf{dl}} = (F_{\mathsf{dl}} \cap S_1) \cup S_2$.

□

The F_{dl}, on the other hand, is meaningful only when dealing with finite sequences. In such cases it is necessary.

We say that a Büchi automaton is *deterministic* iff the following holds: $\forall s, s_1, s_2 \in S : ((s, s_1) \in \Delta \wedge (s, s_2) \in \Delta) \Rightarrow (val(s_1) \neq val(s_2) \vee s_1 = s_2)$.

2.2 Verification with Büchi automata

We define a *system* as a tuple $(S_\mathsf{S}, \Pi, val_\mathsf{S}, \Delta_\mathsf{S}, \hat{S}_\mathsf{S})$, where S_S is a set of states, Π is a set of propositions, $val_\mathsf{S} : S_\mathsf{S} \longrightarrow 2^\Pi$ is a function that assigns to each state of the system a set of propositions, $\Delta_\mathsf{S} \subseteq S_\mathsf{S} \times S_\mathsf{S}$ is a transition relation, and $\hat{S}_\mathsf{S} \subseteq S_\mathsf{S}$ is a set of initial states.

A Büchi automaton $\mathcal{B} = (S_\mathsf{B}, \Pi, val_\mathsf{B}, \Delta_\mathsf{B}, \hat{S}_\mathsf{B}, F_\mathsf{inf}, F_\mathsf{fin}, F_\mathsf{dl})$, representing the negation of a tested property, is used in combination with the system. We consider the product $System \parallel \mathcal{B} = (S, \text{“}\rightarrow\text{”}, \hat{S})$, where $S = \{(s,t) \mid s \in S_\mathsf{S} \wedge t \in S_\mathsf{B} \wedge val_\mathsf{S}(s) = val_\mathsf{B}(t)\}$, “$\rightarrow$” $\subseteq S \times S$ with $(s,t) \rightarrow (s',t')$ iff $(s,s') \in \Delta_\mathsf{S} \wedge (t,t') \in \Delta_\mathsf{B}$, and $\hat{S} = S \cap (\hat{S}_\mathsf{S} \times \hat{S}_\mathsf{B})$.

A state $(s,t) \in S$ is called *reachable* iff there are states $(s_0,t_0), \ldots, (s_n,t_n)$ in S such that $(s_0,t_0) \rightarrow (s_1,t_1) \rightarrow \cdots \rightarrow (s_n,t_n) \wedge (s_0,t_0) \in \hat{S} \wedge (s,t) = (s_n,t_n)$.

In on-the-fly verification, we construct only the reachable part of the product starting from the initial states. We call this part the state space. Furthermore, we can stop immediately after a *counterexample* has been established. This counterexample can be of the following three types:

(i) An infinite sequence of states $(s_0,t_0)(s_1,t_1)\cdots$ such that $(s_0,t_0) \in \hat{S}$ and $\forall i \geq 0 : (s_i,t_i) \rightarrow (s_{i+1},t_{i+1})$ and for infinitely many $i \geq 0$: $t_i \in F_\mathsf{inf}$. If S is finite, this means in practice that a cycle reachable from an initial state is found where at least one state of F_inf occurs.

(ii) A state (s,t) such that it is reachable, $t \in F_\mathsf{dl}$, and there is no s' such that $(s,s') \in \Delta_\mathsf{S}$. That is, a deadlock of the system is reachable where the testing automaton is in a state of F_dl.

(iii) A reachable state (s,t) such that $t \in F_\mathsf{fin}$.

Büchi automata that represent the negations of properties can be either directly built by a system designer or obtained automatically from formulas of Linear Time Temporal Logic (LTL) [1, Section 9.4]. There is a challenge to define an algorithm building automata (for LTL formulas) that are as small as possible, see [7,14,4,6].

The methods aimed at finding counterexamples consist of checking the non-emptiness of the product $System \parallel \mathcal{B}$. Counterexamples of types (ii) or (iii) can be found by checking each state that is encountered during a state space search. There are essentially two methods for finding a counterexample of type (i). One way to accomplish this is to construct the strongly connected components of the state space [2] and then to check whether one of the components contains an infinite acceptance state. This method is not well-suited for on-the-fly verification, because strongly connected components often contain much more states than are needed by the counterexample - it is not unusual that a strongly connected component contains all reachable states. The other method consists of two depth-first-searches [3]; the first one determines and orders the reachable infinite acceptance states, while the second one finds out

whether any of the reachable infinite acceptance states is an element of a cycle.

2.3 Heuristics for Büchi automata reduction

Reduction of a Büchi automaton means the construction of a smaller Büchi automaton that accepts the same language as the original one. Reduction can save effort by making the state space smaller. Unfortunately, also the opposite may happen. Consider, for instance, a system and a Büchi automaton, each of which consists of just a cycle of three states with $val(s) = \emptyset$, one of which is an initial state. The state space has three states. However, if the Büchi automaton is reduced to a cycle of two states, the state space grows to six states.

Many reductions for Büchi automata can be obtained from reductions in finite automata and process algebras. Such reductions are heuristics, and therefore they usually do not guarantee minimal results. Some of the heuristics work for all Büchi automata (such as the strong bisimulation minimisation), but for stuttering-insensitive Büchi automata there exist more efficient heuristics. A detailed discussion of the reductions would be beyond the page limit of this research, but we introduce some superficially. They are used in the examples of this research, although details of the computations are not necessarily shown.

Let $(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a Büchi automaton. The following heuristics can be used to reduce its size:

- All states and transitions that are not reachable can be discarded. They can be found in linear time with any elementary graph search algorithm such as depth first search [2].
- All states and transitions from which no acceptance state is reachable can be removed. This can be done in linear time by conducting a backwards search starting from acceptance states.
- All transitions starting from a finite acceptance state can be removed.
- If an infinite acceptance state is not in any cycle, it can be removed from F_{inf}.

The following heuristics can be used only if the automaton is stuttering-insensitive:

- F_{fin} can be replaced by $\{\, s \in S \mid \exists s_0, \ldots, s_n \in S : s_0 = s \wedge \forall i < n : val(s_i) = val(s_{i+1}) \wedge (s_i, s_{i+1}) \in \Delta \wedge s_n \in F_{\mathsf{fin}} \,\}$, that is, stuttering immediately before entering a finite acceptance state can be ignored. The same applies to F_{dl}.

3 Testing automata

A *testing automaton* is a variant of an extended Büchi automaton that "reads" a sequence in a different way. The important feature of a testing automaton

is that it does not detect valuations, but changes of them. Consequently, a testing automaton can accept only stuttering-insensitive languages.

3.1 Definitions

A *testing automaton* is a 9-tuple

$$(S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$$

where

- S is a finite set. Its elements are called *states*.
- Π is a finite set. Its elements are called *propositions*.
- $val : \hat{S} \to 2^{\Pi}$. That is, only initial states are given valuations.
- $\Delta \subseteq S \times (2^{\Pi} - \{\emptyset\}) \times S$. Its elements are called *transitions*.
- $\hat{S} \subseteq S$. Its elements are called *initial states*.
- $F_{\mathsf{inf}} \subseteq S$. Its elements are called *infinite acceptance states*.
- $F_{\mathsf{fin}} \subseteq S$. Its elements are called *finite acceptance states*.
- $F_{\mathsf{dl}} \subseteq S$. Its elements are called *deadlock acceptance states*.
- $F_{\mathsf{ll}} \subseteq S$. Its elements are called *livelock acceptance states*.

A variant of testing automata was defined in [17]. There synchronous communication via transition labels was used instead of Π an val. Another notion related to stutter-invariant automata was defined in [5]. The main difference between these two definitions is in the acceptance criteria; stutter-invariant automata use infinite acceptance states only.

From now on, let $\mathcal{T} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ be a testing automaton. Define $A \oplus B$ as follows: $A \oplus B = (A - B) \cup (B - A)$.

The testing automaton does not make a move for every symbol that it reads. Instead, it moves only when the valuation changes. To discuss this, we define $\sim s_0 P_0 s_1 P_1 \cdots s_n P_n \rightsquigarrow$ iff $s_0 \in \hat{S} \wedge val(s_0) = P_0 \wedge \forall i < n : ((s_i, P_i \oplus P_{i+1}, s_{i+1}) \in \Delta \wedge P_i \neq P_{i+1}) \vee (s_i = s_{i+1} \wedge P_i = P_{i+1})$. For infinite sequences, $\sim s_0 P_0 s_1 P_1 s_2 P_2 \cdots \rightsquigarrow$ is defined analogously.

An infinite sequence $P_0 P_1 P_2 \cdots$ is *accepted* if at least one of the three conditions below holds:

(i) There are $s_0, s_1, s_2, \ldots \in S$ such that
- $s_i \in F_{\mathsf{inf}}$ for infinitely many i,
- $\forall i : \exists k > i : P_i \neq P_k$, and
- $\sim s_0 P_0 s_1 P_1 s_2 P_2 \cdots \rightsquigarrow$.

(ii) There are $s_0, \ldots, s_k \in S$ such that
- $s_k \in F_{\mathsf{ll}}$,
- $\forall i \geq k : P_i = P_k$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

(iii) There are $s_0, \ldots, s_k \in S$ such that

- $s_k \in F_{\mathsf{fin}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

A finite sequence $P_0 P_1 \cdots P_n$ is *accepted* if at least one of the two conditions below holds:

(iv) There are $s_0, s_1, s_2, \ldots, s_n \in S$ such that
- $s_n \in F_{\mathsf{dl}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_n P_n \rightsquigarrow$.

(v) There are $s_0, s_1, s_2, \ldots, s_k \in S$, for some $k \leq n$ such that
- $s_k \in F_{\mathsf{fin}}$, and
- $\sim s_0 P_0 s_1 P_1 \cdots s_k P_k \rightsquigarrow$.

We say that a testing automaton is *deterministic* iff the following holds: $\forall s, s_1, s_2 \in S : \forall P \subseteq \Pi : ((s, P, s_1) \in \Delta \wedge (s, P, s_2) \in \Delta \Rightarrow s_1 = s_2)$.

3.2 Verification with testing automata

We define a *system* as in Section 2.2. A testing automaton $\mathcal{T} = (S_{\mathsf{T}}, \Pi, val_{\mathsf{T}}, \Delta_{\mathsf{T}}, \hat{S}_{\mathsf{T}}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ is used in combination with a system, and we consider the product $System \parallel \mathcal{T} = (S, \text{“}\rightarrow\text{”}, \hat{S})$, where $S = S_{\mathsf{S}} \times S_{\mathsf{T}}$, $\hat{S} = \{\, (s,t) \in \hat{S}_{\mathsf{S}} \times \hat{S}_{\mathsf{T}} \mid val_{\mathsf{S}}(s) = val_{\mathsf{T}}(t) \,\}$, and $(s,t) \rightarrow (s',t')$ iff one of the following holds:

(i) $(s,s') \in \Delta_{\mathsf{S}} \wedge (t, val_{\mathsf{S}}(s) \oplus val_{\mathsf{S}}(s'), t') \in \Delta_{\mathsf{T}}$, or

(ii) $(s,s') \in \Delta_{\mathsf{S}} \wedge t = t' \wedge val_{\mathsf{S}}(s) = val_{\mathsf{S}}(s')$.

In verification with testing automata, a counterexample can be one of the following:

(i) An infinite sequence of states $(s_0,t_0)(s_1,t_1)\cdots$ such that $(s_0,t_0) \in \hat{S}$ and $\forall i : (s_i,t_i) \rightarrow (s_{i+1},t_{i+1})$ and for infinitely many i: $t_i \in F_{\mathsf{inf}} \wedge val(s_i) \neq val(s_{i+1})$. If S is finite, this means in practice that a cycle is found such that it is reachable from an initial state and there is at least one change in proposition values and at least one state (s',t') in the cycle where $t' \in F_{\mathsf{inf}}$.

(ii) An infinite sequence of states $(s_0,t)(s_1,t)\cdots$ such that $\forall i : (s_i,t) \rightarrow (s_{i+1},t)$ and (s_0,t) is reachable, and $t \in F_{\mathsf{ll}}$. That is, a cycle consisting of transitions with no propositions change is found where the testing automaton remains in a state $t \in F_{\mathsf{ll}}$.

(iii) A reachable state (s,t) such that $t \in F_{\mathsf{dl}}$ and there is no s' such that $(s,s') \in \Delta_{\mathsf{S}}$. That is, a deadlock of the system is reachable where the testing automaton is in a state of F_{dl}.

(iv) A reachable state (s,t) such that $t \in F_{\mathsf{fin}}$.

3.3 An algorithm for livelock detection

The counterexamples of type (ii) in the previous subsection can be detected with the algorithm originally published in [17]. We present it here only slightly

modified for our purpose and call it one-pass algorithm. Notation is the same as in the previous section.

```
procedure ONE-PASS((S,"→",Ŝ))
 Work := Ŝ; Found := Ŝ; ∀(s_S, s_T) ∈ Ŝ : colour((s_S, s_T)) := white
 while Work ≠ ∅
      choose (s_S, s_T) ∈ Work; Work := Work − {(s_S, s_T)}
      if s_T ∈ F_|| then LLDET((s_S, s_T))
      else
           for each (s'_S, s'_T) such that (s_S, s_T) → (s'_S, s'_T) do
                if (s'_S, s'_T) ∉ Found then
                     Work := Work ∪ {(s'_S, s'_T)}
                     Found := Found ∪ {(s'_S, s'_T)}
                     colour((s'_S, s'_T)) := white
end procedure

procedure LLDET((s_S, s_T))
 if colour((s_S, s_T)) = black then return
 colour((s_S, s_T)) := gray
 for each (s'_S, s'_T) such that (s_S, s_T) → (s'_S, s'_T) do
      if val(s_S) = val(s'_S) then
           if (s'_S, s'_T) ∉ Found then
                Found := Found ∪ {(s'_S, s'_T)}; colour((s'_S, s'_T)) := white
                LLDET((s'_S, s'_T))
           else if colour((s'_S, s'_T)) = gray then ERROR FOUND!
           else LLDET((s'_S, s'_T))
      else if(s'_S, s'_T) ∉ Found then
           Work := Work ∪ {(s'_S, s'_T)}; Found := Found ∪ {(s'_S, s'_T)}
           colour((s'_S, s'_T)) := white
 colour((s_S, s_T)) := black
 return
end procedure
```

If Work is a stack, the outer search (ONE-PASS) is effectively a DFS and if Work is a queue, it is a BFS. We do not commit to any particular way of "choosing" the transitions and the states to be explored. It does tend to have a significant effect on the way the algorithm behaves and this effect is explored in Section 5.

3.4 Heuristics for testing automata reduction

All the algorithms in Section 2.3 that do not assume stuttering-insensitivity apply. The remaining are superfluous. In addition:

- When we are only interested in infinite sequences, if there is a state that is

both an infinite and livelock acceptance state and has a local loop for each $P \in (2^{\Pi} - \emptyset)$, such a state can be converted into a finite acceptance state. If we are interested also in finite sequences, then the state must also be a deadlock acceptance state to begin with.

4 Transformation between Büchi automata and testing automata

4.1 Construction of a testing automaton from a Büchi automaton

Theorem 4.1 *If a Büchi automaton is stuttering-insensitive, then there is a testing automaton that accepts precisely the same sequences, and has the same number of states. If the Büchi automaton is deterministic, then the testing automaton is also deterministic.*

Proof. Let $\mathcal{B} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}})$ be a stuttering-insensitive Büchi automaton. We construct a testing automaton $\mathcal{T} = (S, \Pi, val^T, \Delta^T, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$, where

- $val^T(s) = val(s)$, whenever $s \in \hat{S}$.
- $\Delta^T = \{\, (s, P, s') \mid (s, s') \in \Delta \wedge (P = val(s) \oplus val(s')) \wedge P \neq \emptyset \,\}$.
- $F_{\mathsf{ll}} = \{\, s \in S \mid \exists s_0, s_1, \ldots \in S : s_0 = s \wedge \forall i : val(s_i) = val(s_{i+1}) \wedge (s_i, s_{i+1}) \in \Delta \wedge |\{\, i \mid s_i \in F_{\mathsf{inf}} \,\}| = \infty \,\}$. These states can be detected by only taking into account transitions $(s, s') \in \Delta$ such that $val(s) = val(s')$ while looking for cycles that contain an F_{inf}-state, and then taking all states from which such an F_{inf}-state is reachable via such transitions.

Note first that nondeterminism is not introduced in the construction. To prove that $\mathcal{L}(\mathcal{T}) = \mathcal{L}(\mathcal{B})$, we take any infinite sequence $P_0P_1P_2\cdots \in \mathcal{L}(\mathcal{B})$. Because the language is stuttering-insensitive, we can assume either that $\forall i : P_i \neq P_{i+1}$ or that $\exists k : \forall i < k : P_i \neq P_{i+1} \wedge \forall i \geq k : P_i = P_{i+1}$. We see that by construction, the testing automaton accepts this sequence. In the case of infinite stuttering, the F_{ll} accepts all the appropriate sequences. Finite sequences $P_0P_1\cdots P_n \in \mathcal{L}(\mathcal{B})$ are handled in a similar way. Since the testing automaton ignores stuttering, the inclusion in the other direction should be obvious. □

A similar result, formulated for stutter-invariant automata, can be found in [5].

4.2 Construction of a Büchi automaton from a testing automaton

Theorem 4.2 *Any testing automaton has a corresponding Büchi automaton that accepts precisely the same language.*

Proof. Let $\mathcal{T} = (S, \Pi, val, \Delta, \hat{S}, F_{\mathsf{inf}}, F_{\mathsf{fin}}, F_{\mathsf{dl}}, F_{\mathsf{ll}})$ be a testing automaton. We will construct a stuttering-insensitive Büchi automaton $(S^B, \Pi, val^B, \Delta^B, \hat{S}^B,$

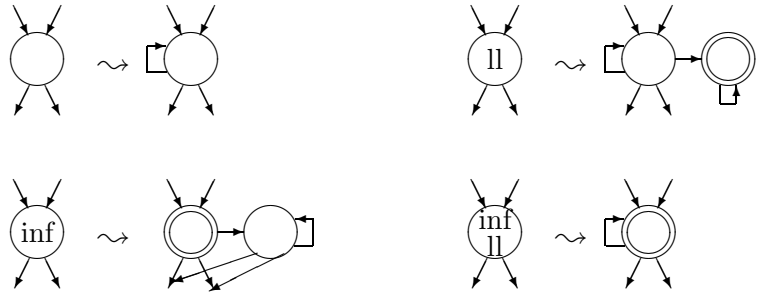


Fig. 1. The construction of a Büchi automaton from a testing automaton

F^B_{inf}, F^B_{fin}, F^B_{dl}). For clarity, we first construct an intermediate testing automaton $(S', \Pi, val', \Delta', \hat{S}', F'_{\mathsf{inf}}, F'_{\mathsf{fin}}, F'_{\mathsf{dl}}, F'_{\mathsf{ll}})$ such that the values of the propositions in the states are unique.

- Let $S' = S \times 2^\Pi$.
- For each $s \in \hat{S}$, put $(s, val(s))$ in $\hat{S}'$.
- $val'((s, P)) = P$ for $s \in \hat{S}'$.
- Δ' is constructed so that whenever $(s, P, s') \in \Delta$, we add $((s, Q), P, (s', Q \oplus P))$ into Δ' for each $Q \in 2^\Pi$.
- For each set of acceptance states F_{x}, $F'_{\mathsf{x}} = F_{\mathsf{x}} \times 2^\Pi$.

Only the reachable part of this intermediate testing automaton needs to be considered.

The second stage of the construction consists of transforming each state depending on whether it is a member of F_{inf} and/or F_{ll}. These transformations are shown in Figure 1. The value of function *val* of a state (s, P) is just P. States retain their status as an initial, finite or deadlock acceptance state. The "secondary" states introduced in Figure 1 inherit their *val* values and finite or deadlock acceptance status from their primary states, and are not initial states. □

When an automaton is obtained according to the construction in this proof or the one in Section 4.1, it can often be reduced using the heuristics of Section 2.3 or 3.4.

Theorem 4.3 *There is a deterministic testing automaton such that no deterministic Büchi automaton accepts precisely the same language.*

Proof. Consider the language $\mathcal{L} = (\{P\}|\emptyset)^*\{P\}^\omega$. It is known that it is not accepted by a deterministic Büchi automaton [15]. A deterministic testing automaton accepting this language is shown in Figure 2. □

However, this result turns out coincidental rather than fundamental: there is also a nondeterministic testing automaton such that no deterministic testing automaton accepts precisely the same sequences.

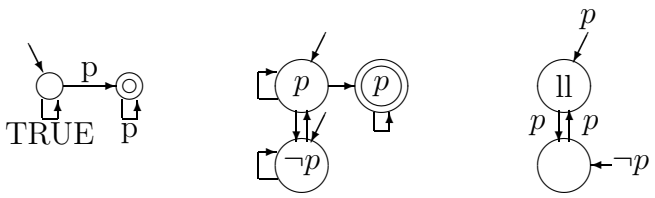


Fig. 2. A transition-labelled and state-labelled Büchi automaton and a testing automaton for the property $\neg\Diamond\Box p$.

5 On-the-fly verification experiments

We compared the algorithm in [3] (from now on just the CVWY-algorithm) to the algorithm in [17], which we call just one-pass algorithm. For our experiments, we have verified the property $\neg\Diamond\Box p$. This property yields a testing automaton that has an ll-state but no inf-states, so the one-pass algorithm applies. Two corresponding Büchi automata and a testing automaton are given in Figure 2. The testing automaton is obtained from the state-labelled Büchi automaton by first using the construction in the proof of Theorem 4.1, and then dropping the unreachable inf-state.

It is easy to compare the performance of the two algorithms when the system works according to the specification, i.e., no counterexample is found. In that case both the algorithms construct the whole state space. In addition, the CVWY-algorithm duplicates some of the states.

A theoretical comparison in the case when an error is actually found is much harder, because the effects of the synchronisation of the system and the automaton are complicated, as it was discussed in Section 2.3. The goal is in any case to produce as few states as possible before finding the error.

Various aspects must be considered when implementing these algorithms. If the incorrect part of the state space is investigated after all other parts, then, of course, the error is found late. Thus the order in which the transitions of the system are investigated may have a significant effect on the behaviour of the algorithms. This order may be affected by the way in which the system is modelled and represented. The implementation details of the algorithm may also turn out to have a formidable impact on the behavior. For example, the depth-first search has a non-recursive implementation where pointers to all the successor states of a state are put on the stack in one batch, but these states are not marked as found at this stage. The stack may contain several pointers to the same state, and the state is marked as found when a pointer to it is popped. This implementation scans the transitions in the opposite direction from the usual recursive implementation of the depth-first search.

In these experiments a total of eight implementations were studied, labelled here with letters from C to J. The meaning of the letters is shown in Table 1. "Error First" means that in the search, the acceptance state is searched first. "Forward" and "Backward" refer to the order in which the transitions of the

	one-pass		CVWY	
	BFS	DFS	Error first	Error last
Forward	C	E	G	H
Reverse	D	F	I	J

Table 1
Implementations of algorithms

Alg.	≥ 10000	≥ 1000	≥ 100	total
J	0	5	22	30
I	1	11	28	30
H	0	8	15	30
G	1	13	23	30
F	2	11	16	30
E	0	4	10	30
D	0	5	19	30
C	0	5	20	30

Table 2
Measurement results for the token ring (53856 states)

system are explored.

Ten experiments were made with the famous ten dining philosophers' system. The property was "philosopher i cannot starve in the state where she has one chop stick and is waiting for the other", where i ranged from 1 to 10.

Thirty experiments were made with an artificially 'broken' token-ring system of six servers. A comprehensive description of the token-ring system can be found in [16]. It consists of servers and clients, where the servers are organised in a ring. There is exactly one token, and a server serves a client only when it has the token. A request for the token is passed to the left in the ring and the token is passed to the right. The original token ring guarantees eventual access. The system we study here has such a flaw that the token may sometimes be passed in the wrong direction, introducing the possibility of starvation due to a livelock. Only the servers were included in the model and they had been minimised first. Five possible starvation states were tried for each of the six stations.

Twelve experiments were made with Fischer's mutual exclusion system [9, p. 2] with 12 servers. Each server was monitored for starvation while waiting for access to the critical section.

Alg.	≥ 10000	≥ 1000	≥ 100	total
J	0	4	7	10
I	1	4	10	10
H	0	4	7	10
G	1	4	10	10
F	2	4	8	10
E	2	4	7	10
D	0	0	1	10
C	0	0	1	10

Table 3
Measurement results for dining philosophers (59048 states)

Alg.	≥ 10000	≥ 1000	≥ 100	total
J	2	12	12	12
I	2	12	12	12
H	2	12	12	12
G	2	12	12	12
F	3	6	9	12
E	3	6	9	12
D	0	0	4	12
C	0	0	4	12

Table 4
Measurement results for Fischer's mutex (49153 states)

The number of states generated before detecting the illegal property was recorded. Tables 2, 3 and 4 show how many of these test runs resulted in at least 10 000 states, at least 1000 states, and at least 100 states to be generated. It is easy to notice that for all the three systems, the implementations C, D, E, and F are the most effective with C & D outperforming the others. The main advantage in the experimental results is shown for Fischer's mutual exclusion, where the test runs of C & D never generated more than 1000 states, whereas the test runs for G, H, I, and K resulted always in more than 1000 and two times in more than 10 000 states. For dining philosophers C & D generate more than 100 states (but less than 1000) only once, whereas the

other implementations generate four times more than 1000 states and at least seven times more than 100 states.

6 Conclusions

In this research we demonstrated with measurements that on-the-fly livelock detection with the algorithm of [17] often outperforms the algorithm of [3], and, to benefit from this observation, we developed the notion of a testing automaton. Due to the way a testing automaton observes the system, it is insensitive to stuttering. We gave constructions for transforming a stuttering-insensitive Büchi automaton to a testing automaton that accepts the same language and vice versa, and showed that a testing automaton can be deterministic more often than the Büchi automaton.

Of course, a testing automaton can benefit from the one-pass algorithm only if it contains livelock acceptance states. Even when it does not, Theorem 4.1 guarantees that a minimal testing automaton for a property can have fewer but cannot have more states than a minimal state-labelled Büchi automaton for the same property. However, one must take into account that reducing the number of states of a Büchi or testing automaton does not necessarily reduce the size of the state space – actually the opposite may happen. Because the size of the state space is more important, one should concentrate on it, and not worry too much about the size of the Büchi or testing automaton.

The algorithms in [17] and [3] disagree on the order in which the state space should be investigated, and thus cannot be immediately integrated. This causes a problem for the on-the-fly verification with testing automata that contain both livelock and infinite acceptance states. One, albeit not ideal, possibility is to use a triple state space search, where the main and secondary searches would be as in the CVWY-algorithm, and the main search would invoke the third level similarly to the "magic bits" in [8, pp. 235–237].

Testing automata are at their best when Π, the set of propositions, is small. In some cases a testing automaton must remember truth values of propositions in its states, making the number of states grow exponentially in the size of Π. This does not, however, directly make the size of the state space grow, because each state of the system specifies unique values for the propositions, and thus picks only one of the alternative testing automaton states as its pair.

Acknowledgements

The authors would like to thank Maciej Szreter for his help in obtaining the experimental results for Fischer's mutual exclusion algorithm.

References

[1] Clarke, E., Grumberg, O., and Peled, D.: *Model checking*, MIT Press, 1999.

[2] Cormen, T.H., Leiserson, C.E., and Rivest, R.L.: *Introduction to algorithms*, MIT Press,1990.

[3] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M.: "Memory-efficient algorithms for the verification of temporal properties", *Formal Methods in System Design*, vol. 1., pp. 275–288, 1992.

[4] Etessami, K. and Holzmann, G.: "Optimizing Büchi Automata", in *Proc. of CONCUR'00*, LNCS 1877, 2000.

[5] Etessami, K.: "Stutter-invariant Languages, ω-Automata, and Temporal Logic", in *Proc. of CAV'99*, pp. 236–248, LNCS 1633, 1999.

[6] Gastin, P. and Oddoux, D.: "Fast LTL to Büchi Automata Translation", in *Proc. of CAV'01*, pp. 53–65, 2001.

[7] Gerth, R., Peled, D., Vardi, M., and Wolper, P.: "Simple on-the-fly automatic verification of linear temporal logic", in *Proc. of PSTV'95*, pp. 3–18, 1995.

[8] Holzmann, G: *Design and Validation of Computer Protocols*, Prentice-Hall 1991.

[9] Lamport, L.: "A fast mutual exclusion algorithm", *ACM Transactions on Computer Systems*, 5(1): pp. 1–11, 1987.

[10] Lamport, L.: "What good is temporal logic.". *Proc. IFIP 9th World Congress*, R.E.A. Mason (editor), North-Holland, pp 657 – 668, 1983.

[11] Manna, Z. and Pnueli, A.: *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag 1991.

[12] Peled, D.: "Combining partial order reductions with on-the-fly model-checking", *Formal Methods in System Design 8*, pp. 39–64, 1996.

[13] Peled, D., Valmari, A. and Kokkarinen, I.: "Relaxed Visibility Enhances Partial Order Reduction", *Formal Methods in System Design*, 19, pp. 275–289, 2001.

[14] Somenzi, F. and Bloem, R.: "Efficient Buchi Automata form LTL formulae", in *Proc. of CAV'00*, LNCS 1855, pp. 247-263, 2000.

[15] Thomas, W.: "Languages, Automata, and Logic", in *Handbook of Formal Languages*, eds. G. Rozenberg and A. Salomaa, Springer-Verlag, pp. 389–455, 1997.

[16] Valmari, A.: "Composition and Abstraction", Cassez, F., Jard, C., Rozoy, B. & Ryan, M. (eds.): *Modelling and Verification of Parallel Processes*, LNCS Tutorials, Lecture Notes in Computer Science 2067, pp. 58–99, Springer-Verlag 2001.

[17] Valmari, A.: "On-the-fly Verification with Stubborn Sets". *Proc. Computer-Aided Verification (CAV) '93*, Lecture Notes in Computer Science 697, pp. 397–408, Springer-Verlag 1993.

Context-Sensitive Visibility

Antti Valmari [1], Heikki Virtanen [2] and Antti Puhakka [3]

Institute of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

An improvement to the so-called *visual verification* approach is presented. Visual verification is a method for checking the correctness of the behaviour of a reactive or concurrent system. It shares a great deal of common ground with ordinary formal state space verification, but is more user-friendly. This is because the user does not need to specify in detail the properties that the system must satisfy to be correct. Instead, the user only lists the atomic actions that are relevant for the property. Computer tools are used to obtain a graphical representation which is a summary of all possible alternative behaviours of the system, and the user then analyses the result. The improvement presented in this article allows the user to pick a region of the graphical representation and investigate it in more detail, without being overwhelmed by the details outside the region. The improvement is illustrated by analysing the livelocks in a model of the alternating bit protocol.

1 Introduction

In order to improve the quality of concurrent and reactive systems, in particular for safety-critical applications, several *formal verification methods* have been developed for ensuring the correctness of the behaviour of the system. Formal verification consists of checking that a formal model of the system *satisfies* a formal *requirement specification* according to some mathematically defined notion of "to satisfy".

Because checking satisfaction is mathematically challenging and therefore a significant burden for the system designer, verification researchers have tried to automate it as much as possible. Unfortunately, verification is demanding also computationally. Fortunately, with a number of ingenious techniques the researchers have been able to develop verification algorithms and tools that

[1] Email: ava@cs.tut.fi
[2] Email: hvi@cs.tut.fi
[3] Email: anpu@cs.tut.fi

are capable of handling many verification tasks of practical significance. (An extensive survey of formal verification, its fundamental performance problem and enhanced verification algorithms is presented in [12].)

With an automated verification method, it suffices that the system developer submits a formal model of the system and the requirement specification. However, in many cases it is very difficult to present a comprehensive requirement specification. A great difficulty here is that one should be able to think a priori of all possible things that the system might do wrong. This means that it is difficult to determine beforehand all the requirements that should be made. On the other hand, if an important requirement is accidentally forgotten, then a badly incorrect system may pass formal verification.

These problems with requirement specifications led to the development of an alternative approach called *visual verification* [16]. Visual verification is based on certain theories and algorithms originally developed for ordinary verification, namely the *Communicating Sequential Processes* (*CSP*) [5,11] and its descendant *Chaos-Free Failures Divergences* (*CFFD*) *Semantics* [17], but these are applied in a slightly different way.

In visual verification, to check a behavioural property of the system, the property needs not be specified in detail — it suffices that the actions (that is, operations, or execution steps) of the system that are relevant for the property are pointed out. The chosen actions are called *visible actions*. Then computer tools produce a graphical representation of the behaviour of the system abstracted such that only the visible actions, their relations to each other, and some information for detecting deadlocks and livelocks are shown. It is important to notice that the representation does not describe just one execution of the system, but *all alternative executions simultaneously*, although with a great deal of detail left out. The user analyses this representation against the expectations that the user has regarding the behaviour of the system.

In our experience, behavioural properties are often easy to check in this way without the burden of specifying the property formally beforehand. What is more, an attempt to fully understand the graphical representation sometimes reveals an error against a necessary correctness property that the user did not even think of, and would thus not have included in the requirement specification. Section 4.2 contains an example of this.

Therefore, although visual verification is perhaps not verification in the strictest sense of the word, in practice it very often produces results that are at least as reliable and sometimes more reliable than with ordinary verification. Examples of the use of visual verification in the development of communication protocols have been given in [7,13,8].

An example of a different kind of visualisation in verification is given in [3]. There, telephone services are specified with graphical diagrams. A model checker is used to find violations of constraints in these diagrams, and the relevant parts of the violating paths are shown to the user.

Nothing in this world is perfect. The main drawback of visual verifica-

tion — in addition to the performance problems that hamper all automatic verification — is that unless the number of the visible actions is kept small, the graphical representation of the behaviour becomes too big for the user to comprehend. As was pointed out in [16], the size of the graphical representation depends crucially on the chosen semantics. The CFFD semantics is optimal (in a certain well-defined sense) for analysing livelocks, deadlocks and illegal sequences of visible actions. This has contributed to the fact that we have been able to apply visual verification to interesting tasks, as discussed above. However, the need for methods of obtaining more useful information with smaller graphical representations is still obvious.

In this article we develop one such method: *context-sensitively visible actions*. Our new method is applicable in a situation where the user has detected something strange in the behaviour, and wants to investigate the peculiar part in more detail. More details can be obtained by declaring more actions visible, but then the graphical representation easily grows too big. Context-sensitive visibility makes it possible to declare that an action is visible *in the peculiar part* and nowhere else. ("Nowhere else" is not precisely true here, but this issue can be clarified only after presenting the theory.) In this way the user can investigate the peculiar part in great detail without being overwhelmed by the details of uninteresting parts of the behaviour of the system.

In Section 2 we recall the background theory underlying this article. Section 3 introduces visual verification and illustrates it with the aid of an example. Use of our new method is illustrated in Section 4, and its theory and implementation are discussed in Section 5.

2 Background Theory

2.1 Labelled transition system

A *labelled transition system* (*LTS*) is a state-machine-like representation of the behaviour of a system or its component process. The system interacts with its environment by executing *visible actions*. The system may also execute *invisible actions* that the environment cannot directly observe. The symbol "τ" has been reserved to denote all invisible actions.

Definition 2.1 A labelled transition system is a quadruple $(S, \Sigma, \Delta, \hat{s})$, where

- S is the set of *states*,
- Σ is the *alphabet*, that is, the set of the *visible actions*; it is assumed that $\tau \notin \Sigma$,
- $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ is the set of *transitions*, and
- $\hat{s} \in S$ is the *initial state*.

If L is an LTS, then its components are denoted with S_L, Σ_L, Δ_L and $\hat{s}_L$.

Example 2.2 Figure 1 shows four LTSs, Sender, Receiver, Data_channel, and

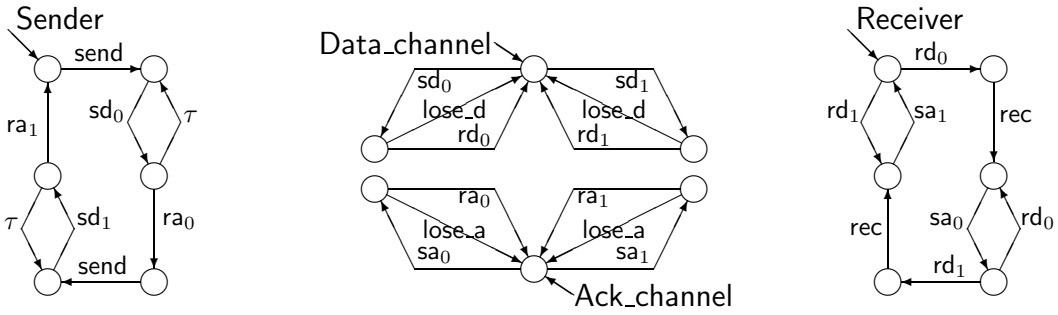


Fig. 1. The LTSs of the alternating bit protocol.

Ack_channel. Together these LTSs comprise a model of the well-known *alternating bit protocol* of [4]. The purpose of the protocol is to implement a reliable data transmission link given unreliable channels. Our model covers only the logic of the protocol, and omits the payload data that is transported. The LTSs Sender and Receiver model the actual protocol, and the other two LTSs model the channels.

Sender first receives a sending request from the customer by executing a send-transition. Then it sends a data message augmented with the bit "0" (sd_0) to Receiver through Data_channel, and starts to wait for an acknowledgement (ra_0). After receiving the acknowledgement, Sender is ready for the transmission of the next data message, this time using "1" as the value of the alternating bit. If the acknowledgement does not arrive or arrives too slowly, Sender makes a timeout with the invisible τ-transition, and sends sd_0 another time. Sender may send sd_0 even a third time and, indeed, any number of times.

Receiver declares new messages with rec and sends an acknowledgement for all messages. The channel processes Data_channel and Ack_channel take a message and then either deliver it to the other side, or dispose of the message via the action lose_d or lose_a. □

2.2 LTS operators

LTSs may be composed together to construct subsystems and systems. The most important operators for this are *parallel composition* " $\|$ " and *hiding*. (For more operators, see e.g. [2,5,11,17].)

Definition 2.3 [Parallel composition] Let $L_1 = (S_1, \Sigma_1, \Delta_1, \hat{s}_1)$, ..., $L_n = (S_n, \Sigma_n, \Delta_n, \hat{s}_n)$ be LTSs. The *parallel composition* of $L_1, \ldots, L_n$ is the LTS $L_1 \| \cdots \| L_n = (S, \Sigma, \Delta, \hat{s})$ such that $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$ and $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_n)$, and S and Δ are defined as the smallest sets such that the following hold.

- Each $s \in S$ is an n-tuple $s = (s_1, \ldots, s_n)$ such that $s_1 \in S_1 \wedge \cdots \wedge s_n \in S_n$.
- $\hat{s} \in S$.
- Let $(s_1, \ldots, s_n) \in S$. If and only if either
  - $a = \tau$ and $\exists i, 1 \le i \le n : (s_i, \tau, s'_i) \in \Delta_i \ \wedge$
    $\forall j, 1 \le j \le n : (j \ne i \Rightarrow s'_j = s_j)$, or
  - $a \in \Sigma$ and $\forall i, 1 \le i \le n : (a \in \Sigma_i \wedge (s_i, a, s'_i) \in \Delta_i) \vee (a \notin \Sigma_i \wedge s'_i = s_i)$

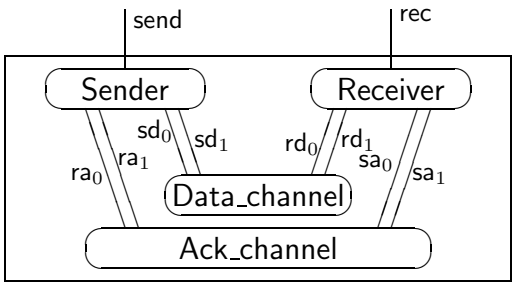


Fig. 2. Subprocesses of the alternating bit protocol and their common actions.

then $(s'_1, \ldots, s'_n) \in S$ and $((s_1, \ldots, s_n), a, (s'_1, \ldots, s'_n)) \in \Delta$.

In less formal terms, visible transitions synchronise as determined by the alphabets of the components, and invisible transitions are always executed by one component at a time.

Definition 2.4 [Hiding] Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS and $A \subseteq \Sigma$. Then **hide** A **in** L is the LTS $(S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \Sigma - A$ and $\Delta' = \{\, (s, a, s') \mid (s, a, s') \in \Delta \wedge a \notin A \,\} \cup \{\, (s, \tau, s') \mid \exists a \in A : (s, a, s') \in \Delta \,\}$.

In other words, **hide** just changes to τ the labels of any transitions labelled with an element of A. For simplicity, if $A = \{a_1, \ldots, a_n\}$, we allow writing **hide** $a_1, \ldots, a_n$ **in** L instead of **hide** $\{a_1, \ldots, a_n\}$ **in** L.

Example 2.5 The structure of the alternating bit protocol is shown in Figure 2. The protocol can now be defined as

hide H **in** (Sender ‖ Receiver ‖ Data_channel ‖ Ack_channel),

where

$H = \{$ sd_0, sd_1, rd_0, rd_1, sa_0, sa_1, ra_0, ra_1, lose_d, lose_a $\}$.

□

We will need later in this article also the less common *multiple renaming* operator $L[A/a]$. Here, L is an LTS, $a \in \Sigma_L$, and A is just any nonempty set of visible action names. The operator replaces each a-transition with $|A|$ alternative transitions, one for each member of A. The alphabet is changed accordingly. It is a special case of a more general multiple renaming operator $L[A_1/a_1, \ldots, A_k/a_k]$ that has been discussed at least in [15].

Definition 2.6 [Multiple renaming] Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS, $a \in \Sigma$, and $\tau \notin A \neq \emptyset$. Then $L[A/a]$ is the LTS $(S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = (\Sigma - \{a\}) \cup A$ and $\Delta' = \{\, (s, b, s') \mid (s, b, s') \in \Delta \wedge b \neq a \,\} \cup \{\, (s, b, s') \mid (s, a, s') \in \Delta \wedge b \in A \,\}$.

2.3 Strong Bisimilarity

For technical reasons the well-known notion of *strong bisimilarity* [10] will be needed.

Definition 2.7 The LTSs $L_1 = (S_1, \Sigma, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, \Sigma, \Delta_2, \hat{s}_2)$ that have the same alphabet are *(strongly) bisimilar*, denoted in this article by $L_1 \simeq_{\mathsf{sb}} L_2$, if and only if there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that the following hold for every $s_1, s_1' \in S_1$, $s_2, s_2' \in S_2$, and $a \in \Sigma \cup \{\tau\}$:

(i) $\hat{s}_1 \sim \hat{s}_2$.

(ii) If $s_1 \sim s_2$ and $(s_1, a, s_1') \in \Delta_1$,
then there is s such that $s_1' \sim s$ and $(s_2, a, s) \in \Delta_2$.

(iii) If $s_1 \sim s_2$ and $(s_2, a, s_2') \in \Delta_2$,
then there is s such that $s \sim s_2'$ and $(s_1, a, s) \in \Delta_1$.

The relation "$\sim$" is called *strong bisimulation*.

2.4 CFFD-Semantics

The notation $s -a_1a_2\cdots a_n\rightarrow s'$ means that the system has a finite execution (that is, a path in the LTS) that starts at s and leads to s' such that the sequence of the labels of the transitions along the path is precisely $a_1, a_2, \ldots, a_n$. If we want to say that there is some s' such that $s -a_1a_2\cdots a_n\rightarrow s'$ but we do not want to specify any such s', we write $s -a_1a_2\cdots a_n\rightarrow$. The existence of an infinite execution from s with the infinite sequence $a_1, a_2, a_3, \ldots$ of transition labels is denoted with $s -a_1a_2a_3\cdots\rightarrow$. For instance, Data_channel has the infinite execution $\hat{s}_{\mathsf{Data\_channel}} -\mathsf{sd}_0\ \mathsf{rd}_0\ \mathsf{sd}_1\ \mathsf{lose\_d}\ \mathsf{sd}_0\ \cdots\rightarrow$.

If $a_1, a_2, \ldots, a_n$ are visible actions, the notation $s =a_1a_2\cdots a_n\Rightarrow s'$ means that there are $m \geq n$ and $b_1, b_2, \ldots, b_m$ such that $s -b_1b_2\cdots b_m\rightarrow s'$ and the result of removing all τs from $b_1b_2\cdots b_m$ is $a_1a_2\cdots a_n$. The notations $s =a_1a_2\cdots a_n\Rightarrow$ and $s =a_1a_2a_3\cdots\Rightarrow$ are defined in an analogous way. We say that $a_1a_2\cdots a_n$ is a *trace* of the system if and only if $\hat{s} =a_1a_2\cdots a_n\Rightarrow$, and $a_1a_2a_3\cdots$ is an *infinite trace* if and only if $\hat{s} =a_1a_2a_3\cdots\Rightarrow$.

We define a *deadlock state* as any state without outgoing transitions. Livelocks may be modelled with the concept of *divergence*. A state s is *divergent*, if and only if an infinite sequence of τ-transitions can be executed from it. The trace $a_1a_2\cdots a_n$ is a *divergence trace*, if and only if there is a divergent state s such that $\hat{s} =a_1a_2\cdots a_n\Rightarrow s$.

Analogously, we could define *deadlock traces* as those traces that can lead to a deadlock state. However, we want our equivalences to be *congruences*, which means that a system is guaranteed to remain equivalent when any of its components is replaced with an equivalent component. The deadlock traces do not induce a congruence with respect to the parallel composition operator, and therefore we need the more general notion of *stable failures*. A stable failure is a pair $(a_1a_2\cdots a_n, \{b_1, b_2, \ldots, b_m\})$ such that there is a state s such that $\hat{s} =a_1a_2\cdots a_n\Rightarrow s$, and $s -b\rightarrow$ is not true for any $b \in \{b_1, b_2, \ldots, b_m, \tau\}$.

The sets of traces, infinite traces, divergence traces and stable failures of an LTS L are denoted with $Tr(L)$, $Inftr(L)$, $Divtr(L)$ and $Sfail(L)$.

Definition 2.8 The *Chaos-free failures divergences (CFFD) semantics* [17] of

L is the triple $(\mathit{Sfail}(L), \mathit{Divtr}(L), \mathit{Inftr}(L))$.[4] Two LTSs are *CFFD-equivalent* if and only if they have the same CFFD-semantics and the same set of visible actions.

The set $\mathit{Tr}(L)$ is not included in the triple, because it can be uniquely determined from the other components due to the formula $\mathit{Tr}(L) = \mathit{Divtr}(L) \cup \{\, \sigma \mid (\sigma, \emptyset) \in \mathit{Sfail}(L) \,\}$ [17]. The set of deadlock traces is $\{\, \sigma \mid (\sigma, \Sigma) \in \mathit{Sfail}(L) \,\}$

CFFD-equivalence is a congruence with respect to parallel composition, hiding and multiple renaming. Furthermore, strong bisimilarity implies CFFD-equivalence; that is, $L_1 \simeq_{\mathsf{sb}} L_2 \Rightarrow L_1 \simeq_{\mathsf{CFFD}} L_2$.

CFFD-semantics contains enough information about the behaviour of the system for the detection of deadlocks, livelocks and illegal actions or sequences of actions, and for listing the traces after which the deadlock etc. may occur. What is more, it was shown in [6] that as long as the LTSs are finite, any semantic model that (1) contains enough information for these tasks and (2) induces a congruence with respect to the parallel composition and hiding operators, must contain at least the same information as CFFD-semantics. This means that CFFD-semantics does not contain more information than is needed. This is very important for visual verification, because it helps to keep the graphical representations small.

In the absence of livelocks, CFFD-semantics coincides with the well-known CSP-semantics of Brookes, Hoare and Roscoe [5,11]. In the presence of livelocks, CFFD-semantics contains more information than CSP-semantics. In CSP-semantics, livelocks are considered as catastrophic modes of behaviour (and called "chaos"). Absolutely no information is preserved about the behaviour of a system that has passed through a potentially livelocking trace. This feature makes CSP-semantics less useful for the verification of a number of systems, including the one used as an example in this article.

3 Visual Verification

We illustrate visual verification with the aid of the alternating bit protocol that was shown in Figures 1 and 2. As we mentioned in Section 2.2, the system as a whole is defined by the formula

hide H **in** (Sender || Receiver || Data_channel || Ack_channel)

where H is the set of actions that we want to consider as internal to the protocol.

ARA [14] is a tool that can be used, among other things, for computing parallel composition and hiding of LTSs, and for reducing LTSs such that

[4] It was originally also required that if one of the systems has a τ-transition that starts in the initial state, then also the other should have. This requirement is needed to ensure the congruence property with respect to the so-called *choice* operator that is common in process algebras, but not used in this article.

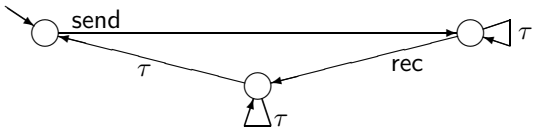


Fig. 3. The externally observable behaviour of the alternating bit protocol.

CFFD-semantics is preserved. ARA contains also a visualisation tool that can show small LTSs on a computer screen graphically in a fairly readable (albeit not always elegant) manner. When ARA was told to construct an LTS of the above model of the alternating bit protocol and then reduce and visualise it, the result was — in essence — Figure 3. We have redrawn the figures for this article, because the output of ARA is unsuitable for printing: it relies on colours and consumes space uneconomically.

We can make a number of observations from Figure 3. First, send and rec alternate, meaning that a message cannot be delivered before a message is sent, and the protocol does not accept a new message for transmission before the previous one has been delivered. Because each state has at least one output transition, we also see that the protocol cannot deadlock. [5]

On the other hand, the two τ-loops in the figure imply that the protocol can livelock. Since the channels are unreliable, and Sender contains no upper limit to the number of times it tries to transmit a message if it receives no acknowledgement, it is natural to guess that the livelocks are due to systematic loss of messages in the channels. Regarding the correctness of the protocol, this explanation of the livelocks would be acceptable, because no protocol can deliver messages if the channels are totally broken. Unfortunately, we do not yet know if it is the *only* reason for the livelocks (or even a reason at all); perhaps there is also a genuine error that causes livelocks even when the channels work well?

We can, fortunately, check this. Each "acceptable" livelock contains infinitely many losses of messages, so livelocks should go away if we make lose_d and lose_a visible, that is, remove them from H. The result of doing this is shown in Figure 4. This LTS contains no τ-loops, so the protocol does not have illegal livelocks. Figure 4 is, however, quite complicated. The reason is that now that lose_d and lose_a are visible, all their possible orderings relative to each other and to send and rec are shown, although we need them only where the livelocks were in the previous picture. The next section presents a new method that solves this problem.

[5] The presence of a τ-transition from the end state of the rec-transition to the initial state, and the absence of a similar transition at the start state of the rec-transition, are explained by the fact that Sender only stops sending data messages after it receives an acknowledgement. On the other hand, it cannot receive an acknowledgement before the rec-transition, because Receiver sends it only after the rec-transition.

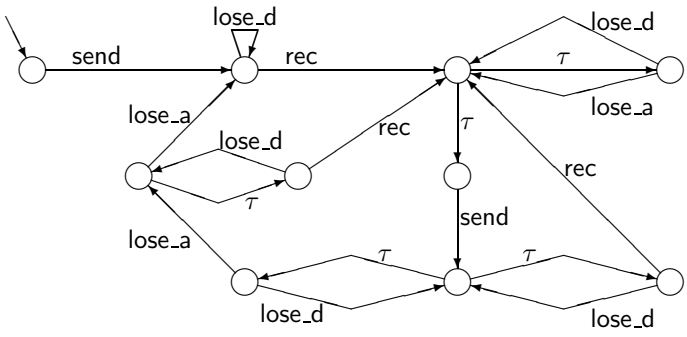


Fig. 4. The alternating bit protocol with lose_d and lose_a visible.

4 Use of Context-Sensitive Visibility

The basic idea of our new method is to make chosen actions visible for *only a part* of the behaviour of the system. The user specifies this part by choosing a set of states from the original visualised LTS, that is, the LTS where the chosen actions are hidden everywhere. An automatic tool then produces an LTS that is semantically otherwise like the original LTS, but occurrences of the chosen actions are now visible (at least) in the part that the user selected.

Before we explain the theory and implementation of context-sensitive visibility in the next section, we illustrate in this section how the user sees it.

4.1 Livelocks in the Protocol

Figure 5 shows the result of making lose_d visible in the ordinary sense (that is, visible everywhere), and lose_a visible in the end state of the rec-transition in Figure 3. To emphasize that not all lose_a-transitions are shown, those that are shown are labelled with $\mathsf{lose\_a}_{\mathsf{vis}}$.

The τ-loop at the start state of the rec-transition has been replaced with a lose_d-loop, so the reason for its existence indeed was an infinite number of losses of messages in Data_channel. If only a finite number of messages is lost, then the system can execute lose_d only a finite number of times, and it must thus eventually execute rec. The τ-loop at the end state of the rec-transition has been replaced with a loop starting with τ and ending with either lose_d or $\mathsf{lose\_a}_{\mathsf{vis}}$. This means that the original τ-loop had two different (although not mutually exclusive) causes: an infinite number of losses of messages in Data_channel, and an infinite number of losses of messages in Ack_channel.

It was thus possible — and actually quite easy — to check from Figure 5 that there are no illegal livelocks. The simplicity of Figure 5 compared to Figure 4 demonstrates the benefit of context-sensitive visibility.

That Figure 5 has essentially the same structure as Figure 3 is good luck; the method does not try to preserve the structure. The method tries to make the resulting LTS small, and it succeeded very well in this example.

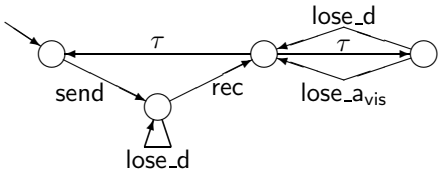


Fig. 5. The alternating bit protocol with lose_d visible everywhere and lose_a context-sensitively visible.

The user would no doubt try other combinations before deciding to make lose_d visible everywhere and lose_a only in the chosen state. This is not a problem since, thanks to computer tools, an unsuccessful attempt does not take much time or effort, and often gives hints for the next attempt.

4.2 Further Analysis

We note from Figure 5 that the state where the $\mathsf{lose\_a}_{\mathsf{vis}}$-transition starts is actually rather curious. When the protocol is in this state, it can continue only by losing either a data message or an acknowledgement. This suggests that if the channels were made reliable by removing the lose_d-transitions from Data_channel and lose_a-transitions from Ack_channel, then the protocol could deadlock.

A more detailed analysis (performed with tools and techniques that are not a topic of the present article) shows that after executing, for example, send sd_0 rd_0 rec τ sd_0 τ sa_0 rd_0 sd_0 τ, the protocol is in a situation where each channel contains a message, and both Sender and Receiver are ready to send but not ready to receive a message. Sending is not possible, however, because the channels are already full. Thus the protocol cannot continue before a data or acknowledgement message is lost by a channel. With reliable channels it would be in a deadlock.

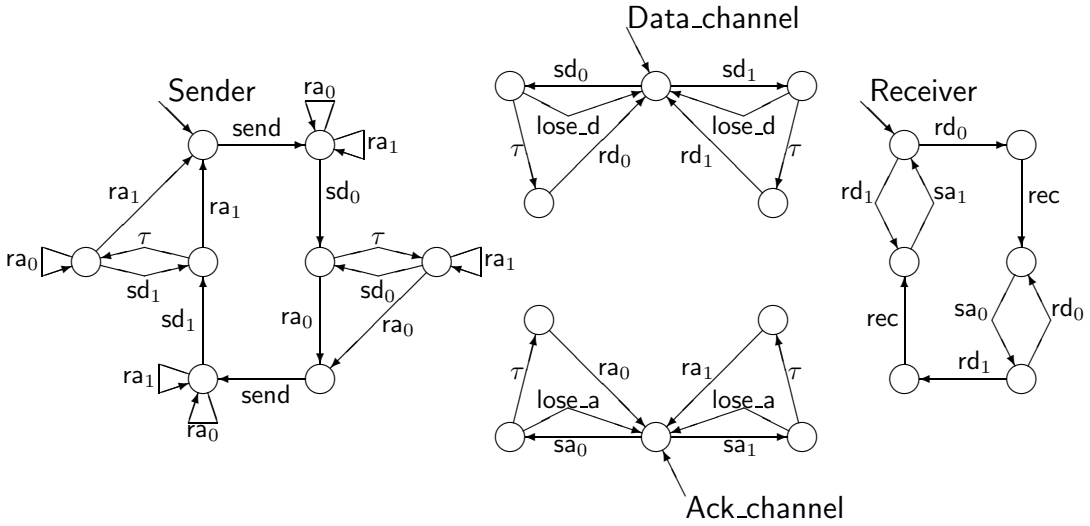


Fig. 6. The correct alternating bit protocol.

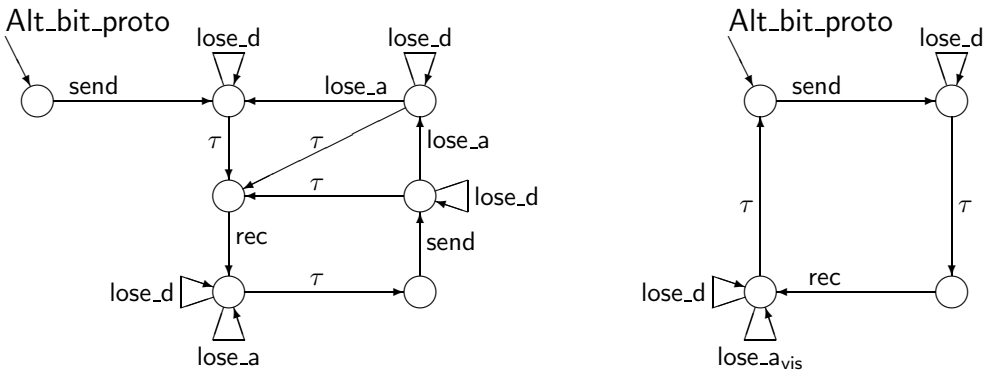


Fig. 7. The behaviour of the correct alternating bit protocol (a) with loss of messages visible all the time, and (b) with lose_a context-sensitively visible.

The above is an example of a subtle error that is easily ignored in ordinary verification. It is not apparent in Figure 3, because the model of channels we have used until now is such that if nothing else can happen, then the channel is guaranteed to lose the message in it, and therefore no deadlock arises. On the other hand, a reliable channel does not have this nice property of losing messages when the protocol would otherwise deadlock.

In brief, our model of channels is incorrect in a way that hides an error in the protocol. Ordinary verification of properties such as "messages are not duplicated" and "if only a finite number of messages is lost, then each send is eventually followed by rec" cannot reveal the error, because the system as a whole *has* these properties. We found the error because Figure 5 gave us some information we did not ask for, namely that the protocol has a state where it may only execute lose_d or lose_a. This is an example of the ability of visual verification to point out errors whose possibility is easily ignored when writing a requirement specification.

To fix the protocol, we add transitions to Sender that consume all unexpected messages. We also change the channels such that they can commit to not lose the message. The fixed protocol is shown in Figure 6, and its behaviour in Figure 7. The behaviour seems correct. The τ-transitions immediately before the rec-transition might seem surprising: does not Sender keep on sending data messages until it receives an acknowledgement from Receiver, which cannot happen before the rec-transition? The answer is that yes, it tries to do that. However, if the channel decides enough many times (twice, to be precise) to deliver a message, then eventually a situation is reached where Sender cannot send any more messages because the data channel is full, while Receiver is ready for rec.

When we made the same analyses by using data and acknowledgement

channels of capacity 2 in the protocol, we found that the pictures where lose_a is visible everywhere, Figures 4 and 7 (a), became more complicated, but the pictures where lose_a is context-sensitively visible, Figures 5 and 7 (b), remained the same. [6]

5 Theory of Context-Sensitive Visibility

5.1 Correctness

In this section we will describe how an LTS is made where some action a is context-sensitively visible, and then formulate and prove two of its properties. The properties state that the LTS is, in a certain precise sense, "between" the LTS where a is visible everywhere, and the original LTS where a is hidden everywhere. It would suffice for our purposes to state and prove these properties in terms of CFFD-semantics. In this case, however, it is natural and easy to prove a much stronger result, namely that the properties hold also when strong bisimilarity is used in their definition. This implies immediately the corresponding results for CFFD, because strong bisimilarity implies CFFD-equivalence, as was mentioned in Section 2.4.

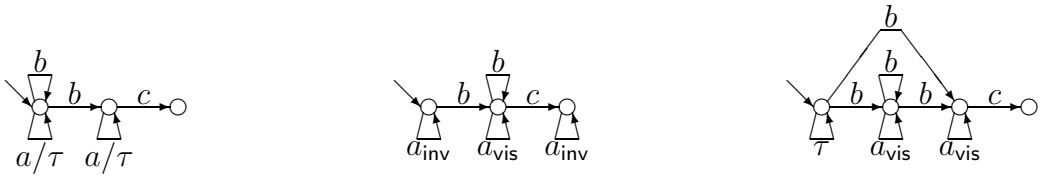
Let us assume that we are analysing the system $Sys = \mathbf{hide}\ a\ \mathbf{in}\ L$, where L can be any (finite) LTS. In a typical case, as in the protocol example above, L has been constructed through parallel composition from subprocesses and, after hiding actions (other than a), reduced according to our equivalence. We would now like to make the action a visible in some states of the system. Context-sensitive visibility is based on (1) introducing two new action names a_{vis} and a_{inv} that are not in Σ_L; (2) constructing a special *switch process* W from Sys, a_{vis}, a_{inv} and a list of states where a should be visible; and (3) then producing the following LTS:

$$Csv = \mathbf{hide}\ a_{\mathsf{inv}}\ \mathbf{in}\ (W \parallel L[\{a_{\mathsf{vis}}, a_{\mathsf{inv}}\}/a])$$

We will describe the actual construction of W in the next section. In this section we will formulate and prove the correctness of Csv. To do that we need to know that W has certain special properties. These properties are listed in the next definition, which says that the alphabet of W is obtained by adding a_{vis} and a_{inv} to the alphabet of Sys; each trace of the original LTS leads to precisely one state of W; W does not have τ-transitions; each state has either an a_{vis}- or an a_{inv}-transition to itself, but not both; and there are no other a_{vis}- or a_{inv}-transitions.

Definition 5.1 Let L be an LTS with Σ_L as its alphabet, and let $a \in \Sigma_L$, but $a_{\mathsf{vis}} \notin \Sigma_L$ and $a_{\mathsf{inv}} \notin \Sigma_L$. Let $Sys = \mathbf{hide}\ a\ \mathbf{in}\ L$. An LTS $(S_W, \Sigma_W, \Delta_W, \hat{s}_W)$ is a *switch* for L and a if and only if

[6] We thank the anonymous referees for the idea of trying the case study with increased channel capacities.



(a) (b) (c)

Fig. 8. (a) An example L/Sys (a-transitions/τ-transitions), (b) a corresponding W, and (c) the corresponding Csv

- $\Sigma_W = \Sigma_{Sys} \cup \{a_{\mathsf{vis}}, a_{\mathsf{inv}}\}$
- $\forall \sigma \in Tr(Sys) : \exists s_\sigma \in S_W : \hat{s}_W =\sigma\Rightarrow s_\sigma$.
  Furthermore, if $\hat{s}_W =\sigma\Rightarrow s_1$ and $\hat{s}_W =\sigma\Rightarrow s_2$, then $s_1 = s_2$.
- $\forall s \in S_W : \neg(s -\tau\rightarrow)$
- $\forall s \in S_W : (s -a_{\mathsf{vis}}\rightarrow s \vee s -a_{\mathsf{inv}}\rightarrow s) \wedge \neg(s -a_{\mathsf{vis}}\rightarrow \wedge\, s -a_{\mathsf{inv}}\rightarrow)$
- $\forall s, s' \in S_W : s -a_{\mathsf{vis}}\rightarrow s' \vee s -a_{\mathsf{inv}}\rightarrow s' \Rightarrow s' = s$

The first correctness criterion says that if we hide again what has been made context-sensitively visible, then what we get is CFFD-equivalent to the original Sys. In fact, it is strongly bisimilar.

Theorem 5.2 **hide** a_{vis} **in** $Csv \simeq_{\mathsf{sb}} Sys$

Proof. The claim is **hide** a_{vis} **in hide** a_{inv} **in** $(W \parallel L[\{a_{\mathsf{vis}}, a_{\mathsf{inv}}\}/a]) \simeq_{\mathsf{sb}}$ **hide** a **in** L. Let $Sys' =$ **hide** a_{vis} **in** Csv. By Definitions 2.3, 2.4, 2.6 and 5.1 we see that $\Sigma_{Sys'} = \Sigma_W \cup \Sigma_{L[\{a_{\mathsf{vis}},a_{\mathsf{inv}}\}/a]} - \{a_{\mathsf{inv}}\} - \{a_{\mathsf{vis}}\} = \Sigma_{Sys} \cup \{a_{\mathsf{vis}}, a_{\mathsf{inv}}\} \cup (\Sigma_L - \{a\}) \cup \{a_{\mathsf{vis}}, a_{\mathsf{inv}}\} - \{a_{\mathsf{vis}}, a_{\mathsf{inv}}\} = \Sigma_{Sys} \cup (\Sigma_L - \{a\}) = \Sigma_{Sys}$. Furthermore, Sys has the same states as L, and the states of Sys' are of the form (s_W, s_L), where s_W is a state of W and s_L is a state of L. Let "$\sim$" $\subseteq S_{Sys'} \times S_{Sys}$ be the relation such that $(s_W, s_{L1}) \sim s_{L2}$ holds if and only if $s_{L1} = s_{L2}$. Clearly, $\hat{s}_{Sys'} \sim \hat{s}_{Sys}$. The τ-transitions of L on either side simulate each other, and whatever else L can do, W can participate in it, either because of its a_{vis}- and a_{inv}-transitions or because of the existence of the unique states s_σ for each trace σ of $Sys =$ **hide** a **in** L. Furthermore, all transitions of W are participated by L. Thus, both processes can simulate every transition of the other process, which proves that the relation is a strong bisimulation. □

A natural next claim could be that Csv — or a process strongly bisimilar to it — can be obtained by converting some a-transitions of L to a_{vis}-transitions, and the remaining a-transitions to τ-transitions. This is not true, however, as the example in Figure 8 demonstrates. We want to make a visible at the middle state of Sys in Figure 8 (a), and invisible elsewhere. However, if the first a-transition of L is converted to τ, then no state of the result can execute both b and a_{vis}, so no state can simulate the second state of Csv in Figure 8 (c). On the other hand, if the first a-transition is converted to a_{vis}, then the result cannot simulate the initial τ-transition of Csv.

What is true, however, is that there is an LTS that is strongly bisimilar to L, and from which *Csv* can be obtained by renaming and hiding a-transitions.

Theorem 5.3 *Let* $L' = (W \parallel L[\{a_{\mathsf{vis}}, a_{\mathsf{inv}}\}/a])[\{a\}/a_{\mathsf{vis}}][\{a\}/a_{\mathsf{inv}}]$.

- *Csv can be obtained from* L' *by hiding some* a*-transitions, and renaming the remaining* a*-transitions to* a_{vis}.
- $L' \simeq_{\mathsf{sb}} L$

Proof. The first claim follows directly from the definitions, if we hide those a-transitions that were created from a_{inv}-transitions with the "$[\{a\}/a_{\mathsf{inv}}]$"-operator, and rename the remaining a-transitions to a_{vis}. A proof of the second claim is obtained from the proof of Theorem 5.2 by replacing L' and L for Sys' and Sys, respectively, and making trivial changes to the formulae giving the alphabets and to the labels of transitions that originate from a-transitions of L. □

5.2 Constructing the Switch

Let $S_{\mathsf{vis}} \subseteq S_{Sys}$ be the set of the states of Sys where the user wanted a to be visible. For every $\sigma \in \Sigma_{Sys}^*$ let $s_\sigma = \{s \in S_{Sys} \mid \hat{s}_{Sys} =\sigma\Rightarrow s\}$. Intuitively, the switch W is obtained from Sys by converting it to a deterministic LTS, and adding to each state an a_{vis}- or a_{inv}-loop depending on whether the user wanted a to be visible in any original state contained in the deterministic state. The idea of the loops is that while the switch process remains in this state, it allows the target process to execute freely the chosen action but blocks it from executing the alternative action. Formally, the switch W is the following LTS:

- $S_W = \{\, s_\sigma \mid \sigma \in Tr(Sys) \,\}$
- $\Sigma_W = \Sigma_{Sys} \cup \{a_{\mathsf{vis}}, a_{\mathsf{inv}}\}$
- $\Delta_W = \{\, (s_\sigma, b, s_{\sigma b}) \mid \sigma b \in Tr(Sys) \,\} \cup \{\, (s_\sigma, a_{\mathsf{vis}}, s_\sigma) \mid s_\sigma \cap S_{\mathsf{vis}} \neq \emptyset \,\} \cup \{\, (s_\sigma, a_{\mathsf{inv}}, s_\sigma) \mid s_\sigma \cap S_{\mathsf{vis}} = \emptyset \,\}$
- $\hat{s}_W = s_\varepsilon$

Sys is made deterministic with the well-known subset construction that can be found in any textbook on finite automata or compilers, for example [9,1]. It is important to notice that despite the subset construction this operation is not costly, because we are not using the complete state space of the system but the abstracted, reduced version that in any case has to be small enough for visual verification.

It is possible that s_σ contains a state from S_{vis} and another state from outside S_{vis}. In that case, W may make a visible even if Sys is in a state where the user did not request a to be visible. The visibility of a depends only on the trace executed by Sys so far. This is the sense mentioned in the introduction in which our new method may show more than the user asked

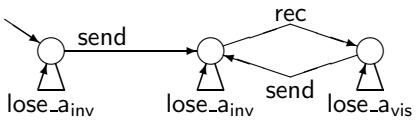


Fig. 9. The switch process.

for. The user can never lose any information, but the final LTS can become larger than would be absolutely necessary.

The switch used in the protocol example is shown in Figure 9.

6 Conclusions

We presented a method for making an action visible in some parts of an LTS and invisible in other parts. The method is based on duplicating the action in question into a visible and invisible version, and constructing a special switch process that chooses which version of the action may occur. The invisible version is then hidden, and the visible version is shown.

As the example in this paper demonstrates, context-sensitive visibility can improve visual verification. It reduces the size of the LTS that is shown, while still providing the information that the user wanted.

The implementation of context-sensitive visibility is not at all difficult. Multiple renaming and the adding of self-loop transitions are straightforward operations, and hiding, parallel composition and determinisation already exist in many LTS manipulation tools, including ARA. The method is also computationally cheap, because it is performed on an abstracted, reduced version of the system instead of the complete state space.

The method can perhaps be improved by minimising the switch before use. This issue is not trivial, however, because the last two conditions of Definition 5.1 are not preserved by strong bisimilarity. Another hypothesis is that the method can be used to make several actions context-sensitively visible simultaneously. We plan to investigate these hypotheses in the future.

Acknowledgements

The work of A. Puhakka was funded in part by the Academy of Finland.

References

[1] Aho, A. V., Sethi, R. & Ullman, J. D.: *Compilers — Principles, Techniques, and Tools*. Addison-Wesley 1986, 796 p.

[2] Bolognesi, T. & Brinksma, E.: "Introduction to the ISO Specification Language LOTOS". Computer Networks and ISDN Systems 14, 1987, pp. 25-59.

[3] Braun, V., Margaria, T., Steffen, B. & Yoo, H.: "Automatic Error Location for IN Service Definition". *Services and Visualization: Towards User-Friendly Design*, Lecture Notes in Computer Science 1385, Springer-Verlag 1998, pp. 190–207.

[4] Bartlett, K. A., Scantlebury, R. A. & Wilkinson, P. T.: "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links". *Communications of the ACM* 12 (5) 1969, pp. 260–261.

[5] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall 1985, 256 p.

[6] Kaivola, R. & Valmari, A.: "The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic". *Proceedings of CONCUR '92, Third International Conference on Concurrency Theory*, Lecture Notes in Computer Science 630, Springer-Verlag 1992, pp. 207–221.

[7] Karsisto, K. & Valmari, A.: "Verification-Driven Development of a Collision-Avoidance Protocol for the Ethernet". *Proceedings of Formal Techniques in Real-Time and Fault Tolerant Systems*, Lecture Notes in Computer Science 1135, Springer-Verlag 1996, pp. 228–245.

[8] Kervinen, A., Valmari, A. & Järnström, R.: "Debugging a Real-life Protocol with CFFD-Based Verification Tools". *Proceedings of FMICS 2001, 6th International Workshop on Formal Methods for Industrial Critical Systems*, pp. 13–27.

[9] Lewis, H. R. & Papadimitriou, C. H.: *Elements of the Theory of Computation.* Prentice-Hall 1998, 361 p.

[10] Park, D.: "Concurrency and Automata on Infinite Sequences". *Theoretical Computer Science: 5th GI-Conference*, Lecture Notes in Computer Science 104, Springer-Verlag 1981, pp. 167–183.

[11] Roscoe, A. W.: *The Theory and Practice of Concurrency.* Prentice-Hall 1998, 565 p.

[12] Valmari, A.: "The State Explosion Problem". *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science 1491, Springer-Verlag 1998, pp. 429–528.

[13] Valmari, A., Karsisto, K. & Setälä, M.: "Visualisation of Reduced Abstracted Behaviour as a Design Tool". *Proceedings of PDP'96, the Fourth Euromicro Workshop on Parallel and Distributed Processing*, IEEE Computer Society Press, pp. 187–194.

[14] Valmari, A., Kemppainen, J., Clegg, M. & Levanto, M.: "Putting Advanced Reachability Analysis Techniques Together: the ARA Tool". *Proceedings of Formal Methods Europe '93: Industrial-Strength Formal Methods*, Lecture Notes in Computer Science 670, Springer-Verlag 1993, pp. 597–616.

[15] Valmari, A. & Kokkarinen, I.: "Unbounded Verification Results by Finite-State Compositional Techniques: 10^{any} States and Beyond". *Proceeding of the 1998 International Conference on Application of Concurrency to System Design*, Aizu-Wakamatsu, Fukushima, Japan, March 1998, IEEE Computer Society, pp. 75–85.

[16] Valmari, A. & Setälä, M.: "Visual Verification of Safety and Liveness". *Proceedings of Formal Methods Europe '96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science 1051, Springer-Verlag 1996, pp. 228–247.

[17] Valmari, A. & Tienari, M.: "Compositional Failure-Based Semantic Models for Basic LOTOS". *Formal Aspects of Computing* (1995) 7: 440–468.