NASA/CP–2005–212788

IEEE/NASA ISoLA 2005

IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation

Tiziana Margaria, Bernhard Steffen, and Michael G. Hinchey, Editors

Proceedings of a workshop held at the
Loyola College Graduate Center
Columbia, Maryland, USA
23–24 September 2005

September 2005

NASA/CP–2005–212788

IEEE/NASA ISoLA 2005

IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation

Editors:

Tiziana Margaria
University of Göttingen, Göttingen, Germany

Bernhard Steffen
University of Dortmund, Dortmund, Germany

Michael G. Hinchey
NASA Goddard Space Flight Center, Greenbelt, Maryland

Proceedings of a workshop held at the
Loyola College Graduate Center
Columbia, Maryland, USA
23–24 September 2005

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

September 2005

Preface

This volume contains the Preliminary Proceedings of the 2005 IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation, with a special track on the theme of Formal Methods in Human and Robotic Space Exploration. The workshop was held on 23–24 September 2005 at the Loyola College Graduate Center, Columbia, Maryland, USA.

The idea behind the Workshop arose from the experience and feedback of ISoLA 2004—the 1st International Symposium on Leveraging Applications of Formal Methods—held in Paphos (Cyprus) held during October–November. ISoLA 2004 served the need of providing a forum for developers, users, and researchers to discuss issues related to the adoption and use of rigorous tools and methods for the specification, analysis, verification, certification, construction, test, and maintenance of systems from the point of view of their different application domains.

The ISoLA series of events serves the purpose of bridging the gap between designers and developers of rigorous tools, and users in engineering and in other disciplines, and to foster and exploit synergetic relationships among scientists, engineers, software developers, decision makers, and other critical thinkers in companies and organizations. In particular, by providing a venue for the discussion of common problems, requirements, algorithms, methodologies, and practices, ISoLA aims at supporting researchers in their quest to improve the utility, reliability, flexibility, and efficiency of tools for building systems, and users in their search for adequate solutions to their problems.

This Workshop has a particular focus on Formal Methods in Human and Robotic Space Exploration. It concentrates on application domains relevant to the use of rigorous and/or provable techniques in several aspects of space exploration, as well as on the enabling technologies and infrastructure.

The workshop program consisted of one keynote lecture by John Knight (University of Virginia, USA); two invited talks by Ramesh Bharadwaj (Naval Research Laboratory, Washington, DC) and by César A. Muñoz (National Institute of Aerospace, Hampton, VA); and three thematic sessions composed of nine regular papers, and a panel. Additionally, two doctoral symposium sessions offered students the possibility to orally present and demonstrate their ongoing research in the course of their graduate and advanced undergraduate studies.

We thank the members of the Program committee and their sub-referees for selecting the papers to be presented. Our thanks also goes to Ben Benokraitis and Binod Rai of Loyola College in Maryland for the excellent local organization. Special thanks are due to the following organizations for their sponsorship: the IEEE-CS TC on Complexity in Computing, NASA Goddard Space Flight Center, Loyola College in Maryland (who provided extremely comfortable premises at their Graduate Center in Columbia), the University of Dortmund, and the University of Göttingen. We are also very grateful to the IFIP TC10 on Computer Systems Technology for the support granted to the doctoral students. Finally, we thank EASST for its endorsement of the workshop.

— *Tiziana Margaria, Bernhard Steffen,* and *Michael Hinchey*

ISoLA 2005 Chairs and Committee Members

General Chair

Tiziana Margaria (University of Göttingen, Germany)

Program Chairs

Bernhard Steffen (University of Dortmund, Germany)
Michael Hinchey (NASA GSFC, USA)

Organization Chairs

V.J. "Ben" Benokraitis (Loyola College in Maryland, USA)
Binod Rai (Loyola College in Maryland, USA)

Publicity Chairs

Lenore Zuck (University of Illinois at Chicago, USA)

Program Committee

Yamine Ait Ameur (ENSMA, France)
V.J. "Ben" Benokraitis (Loyola College in Maryland, USA)
Ramesh Bharadwaj (Naval Research Laboratory, USA)
Shawn Bohner (Virginia Tech, USA)
Jonathan Bowen (London South Bank University, UK)
Martin Featjer (JPL, USA)
Stefania Gnesi (IEI Pisa, Italy)
John Hatcliff (Kansas State University, USA)
Joost Kok (University of Leiden, The Netherlands)
Mike Lowry (NASA Ames, USA)
Markus Müller-Olm (University of Dortmund, Germany)
Jim Rash (NASA GSFC, USA)
Chris Rouff (SAIC, USA)
Walt Truszkowski (NASA GSFC, USA)

Table of Contents

HYBRID VERIFICATION OF AN AIR TRAFFIC OPERATIONAL CONCEPT*

César A. Muñoz†
National Institute of Aerospace, USA
Gilles Dowek‡
École polytechnique, France

ABSTRACT

A concept of operations for air traffic management consists of a set of flight rules and procedures aimed to keep aircraft safely separated. This paper reports on the formal verification of separation properties of the NASA's Small Aircraft Transportation System, Higher Volume Operations (SATS HVO) concept for non-towered, non-radar airports. Based on a geometric description of the SATS HVO air space, we derive analytical formulas to compute spacing requirements on nominal approaches. Then, we model the operational concept by a hybrid non-deterministic asynchronous state transition system. Using an explicit state exploration technique, we show that the spacing requirements are always satisfied on nominal approaches. All the mathematical development presented in this paper has been formally verified in the Prototype Verification System (PVS).

Keywords. Formal verification, hybrid systems, air traffic management, theorem proving

INTRODUCTION

The safety objective of air traffic management is to provide aircraft separation. This objective is achieved trough air/ground equipment and a set of flight rules and procedures, usually called *concept of operations*. Emerging and more reliable surveillance and communication technologies have enabled new concepts where pilots and air traffic controllers share the responsibility for traffic separation. One of such concepts is NASA's *Small Aircraft Transportation System (SATS), Higher Volume Operation (SATS HVO)* [Ref. 1].

The SATS program [Ref. 6] aims to increase access to small airports in the US during instrument approach operations. Currently, under poor weather conditions, small airports are restricted to *one-in*/*one-out* operations. The SATS HVO concept enables up to four simultaneous arrival approaches and multiple departures. A key aspect of the concept is that, under nominal operations, aircraft are *self-separated*, i.e., pilots are responsible for separation without assistance of an air traffic controller. To this end, the SATS HVO concept designs the airspace surrounding the airport as a *Self-Controlled Area (SCA)*. A centralized, automated system, called the *Airport Management Module* (AMM), serves as an arbiter to aircraft entering the SCA. In this concept, aircraft constantly broadcast their locations and, therefore, they have an updated view of the SCA.

The SATS HVO operational concept is a collection of rules and procedures to be followed by aircraft operating or transitioning in/out the SCA. For instance the concept of operations states when and how an aircraft is allowed to enter (or leave) the SCA, when an aircraft is allowed to initiate the approach, and how to perform a missed approach. In order to alleviate pilot workload and increase situation awareness, on board navigation tools provide advisories that assist pilots in following these procedures.

Because the operational concept is a safety critical element of the SATS program, the task of showing that it satisfies safety requirements is acomplished using formal mathematical analysis. A discrete mathematical model of the SATS HVO operational concept is described in [Ref. 5]. That model was mechanically checked for safety and liveness properties. As result of this research, several modification were incorporated to the concept [Ref. 2].

*This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043.
†`munoz@nianet.org`
‡`Gilles.Dowek@polytechnique.fr`

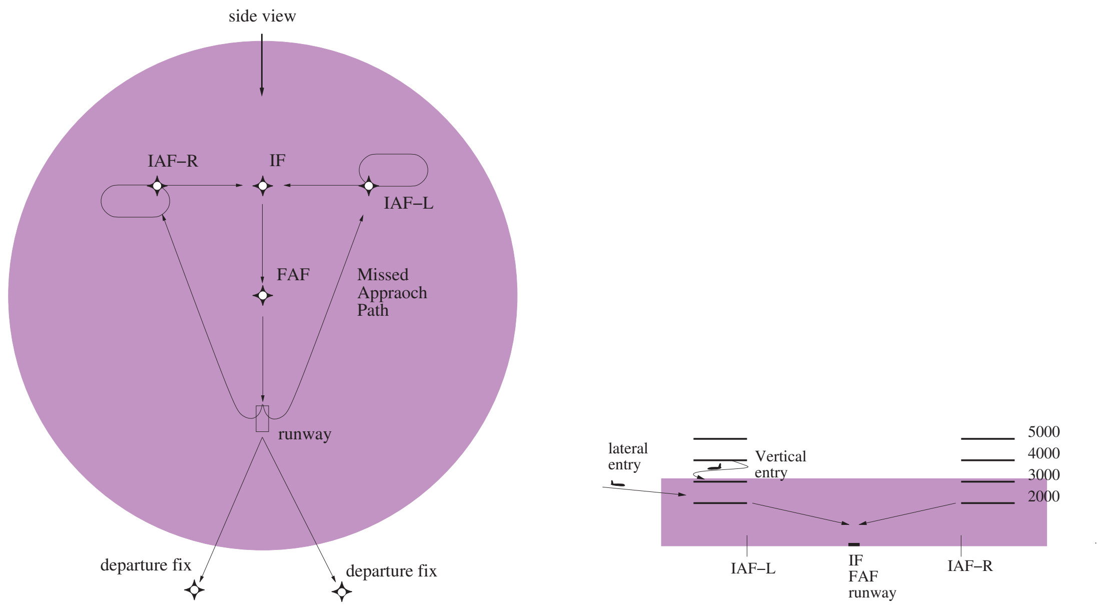


Figure 1: Top and side view of SCA

The discrete model in [Ref. 2, 5] is not precise enough to enable verification of spacing properties. In this paper, we described a *hybrid* model that extends the discrete model to take into account the geometry of the SCA and the aircraft speed performances. Using this new model, we formally verified that the SATS HVO operational concept *effectively* achieves self-separation, i.e., aircraft performing nominal approaches are safely separated according to minimum spacing criteria.

HIGHER VOLUME OPERATIONS

In the SATS HVO concept, pilots operating within the Self-Controlled Area (SCA) are required to fly by latitude/longitude points in the space, called *fixes*. Similar to a GPS-T approach [Ref. 3], fixes are arranged as a T (see Figure 1).[1] The fixes at the extremes of the T are called *initial approach fixes (IAF's)* and they are the entry points to the SCA. The IAF's also serve as *missed approach holding fixes (MAHF's)*, i.e., fixes where aircraft will proceed in case they have to perform a missed approach. The holding areas are located at 2000 feet and 3000 feet at the IAF's.

There are two types of entry procedures: *vertical entry* and *lateral entry*. In a vertical entry, an aircraft holds at 3000 feet until it is enabled to descend to 2000 feet. In a lateral entry an aircraft flies directly to its IAF at 2000 feet. When the aircraft is enabled to initiate the approach, it flies to the *intermediate fix (IF)*, from there to the *final approach fix* (FAF), and finally to the runway threshold. In case of a missed approach, the aircraft flies to its assigned missed approach holding fix at the lowest available altitude (2000 or 3000 feet). Then, it re-initiates the approach and either follows a normal landing procedure or leaves the SCA. The linear segments between the IAFs and the IF are called *base segments* and the segment between the IF and the runway threshold is called *final segment*. Henceforth, we say that an aircraft is *on final approach* if it is in the base of final segments.

The Airport Management Module (AMM) is an automated centralized system that resides at the airport grounds. It receives state information from aircraft in the vicinity of the airport and communicates with aircraft via data link. The AMM provides entry clearances (vertical or lateral) and assigns missed approach holding fixes. When an entry is granted by the AMM, the aircraft receives a *follow notification* and a *missed approach holding fix assignment*. The follow notification is either *none*, if it is the first aircraft in the landing sequence, or the identification of a *lead* aircraft. Missed approach holding fixes are assigned by the AMM on an alternating basis. This technique ensures that consecutive aircraft on missed approach are not flying to the same MAHF.

[1] As it is usually depicted, right and left are relative to the pilot facing the runway, i.e., opposite from the reader's point of view.

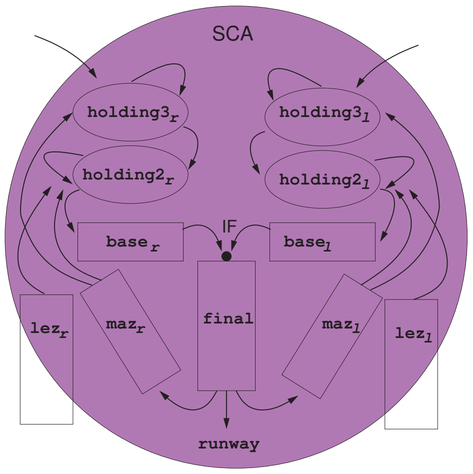


Figure 2: Discrete view of SCA

For nominal arrival operations, self-separation is achieved by requiring an aircraft to hold at its IAF until it meets a spacing safety threshold with respect to its lead aircraft. The threshold shall guarantee a minimum separation during the approach and during a missed approach, in case of this eventuality.

The concept of operations also describes nominal departure operations. However, for simplicity, the analysis presented in this paper only considers arrival operations. This simplification does not affect the result of the formal verification as arriving aircraft are geographically separated from departing aircraft and an aircraft cannot depart if there is an aircraft on final approach. The fact that departing aircraft are also separated can be verified using the techniques presented in this paper.

DISCRETE MODEL AND ITS LIMITATIONS

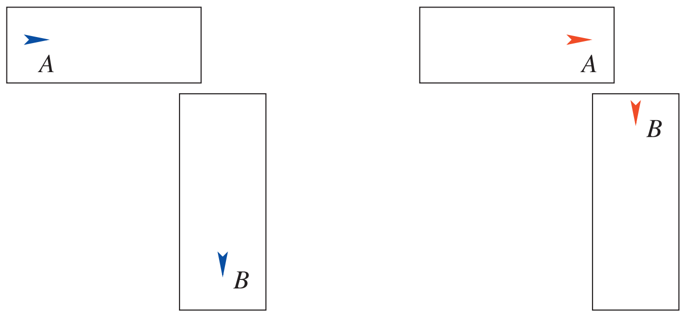
The discrete model described in [Ref. 2, 5] is a mathematical abstraction of the SATS HVO concept. A simple way to visualize that model is via an analogy with a board game where the board is a discretized SCA, the pieces that move across the board are the aircraft, and the rules of the game are given by the concept of operations. This analogy is illustrated in Figure 2. The places where an aircraft can be during an arrival operation are called *zones*. There are 12 zones:

- `holding3` (left, right): Holding patterns at 3000 feet.
- `holding2` (left, right): Holding patterns at 2000 feet.
- `lez` (left, right): Lateral entry zones.[2]
- `base` (left, right): Base segments.
- `maz` (left, right): Missed approach zones.
- `final` and `runway`: Final segment and runway.

An aircraft is always in one and only one zone, but several aircraft may be in the same zone. Aircraft leave the zones in the same order as they arrive. The arrows in Figure 2 are the valid moves and they represent 15 flight rules and procedures:

- Vertical entry (left, right): Initial move to `holding3`.
- Lateral entry (left, right): Initial move to `lez`.
- Descend (left, right): Move from `holding3` to `holding2`.

[2]Lateral entry zones start outside the SCA.



(a) Aircraft A and B are separated (b) Aircraft A and B are not separated

Figure 3: Indistinguishable discrete states

- Approach initiation (left, right): Move from `holding2` to `base`.
- Final approach (left, right): Move from `base` to `final`.
- Landing: Move from `final` to `runway`.
- Missed approach initiation (left, right): Move from `final` to `maz`.
- Transition to lowest available altitude (left, right). Move from `maz` to either `holding3` or `holding2`.

In this model, each aircraft is represented by its initial approach fix (left or right), landing sequence (natural number), and missed approach holding fix assignment (left or right). Aircraft identifications are implicit as aircraft can be distinguished from each other by their landing sequence. The AMM is modeled by the next available landing sequence (natural number) and the next alternating missed approach holding fix (left or right).

The discrete model is conservative in the sense that it abstracts away the SCA geometry and physical performance parameters of the aircraft. Hence, it includes scenarios that may no physically occur in the real world. We argue that the model is complete, i.e., it includes all nominal operations. Of course, this cannot be proved formally. However, the model has been extensively reviewed by the developers of the SATS HVO concept as it was used as a designing tool of the final concept [Ref. 2].

From a mathematical point of view, the discrete model is a state transition system where the states are snapshots of the zones at discrete times and the transitions describe how the states evolve when the flight procedures are applied. A priori, there are no bounds on the number of aircraft in each zone; therefore, the transition system is potentially infinite. However, it turns out that the transition system is finite. Indeed, it was exhaustively explored [Ref. 5] using the verification system PVS [Ref. 7]. Among several other properties, it was formally verified that the model of the SATS HVO concept allows up to four simultaneous arrival approaches, which is better than the current one-in/one-out mode of operation, and that eventually all aircraft land or depart, i.e., there are no deadlocks.

The discrete model does not support verification of spacing properties. In particular, the two states depicted in Figure 3 are indistinguishable by the discrete model, although they do not satisfy the same separation requirements. This behavior is due to the way the approach initiation procedure was written in the discrete model. Indeed, the concept of operations states that an aircraft may initiate the approach if (a) it is the first aircraft in the landing sequence or (b) it meets a safety threshold with respect to the lead aircraft, which is already on approach [Ref. 1]. There are several ways a pilot can check whether the safety threshold is satisfied or not. In the most conservative case, the pilot has to delay the approach initiation until the lead aircraft is within 6 nautical miles from the runway. The value 6 is for a nominal SCA where the base segments are 5 nautical miles and the final segment is 10 nautical miles. In the general case, this value is configurable according to the geometry of the SCA. Since the geometry of the aircraft is not considered in the discrete model, the approach initiation procedure has to be modified. The condition (a) rests the same. However, the discrete model uses a weaker condition (b) where an aircraft can initiate the approach as soon as the lead aircraft is already on the final approach (base or final segments). As the safety threshold is not checked, spacing properties cannot be verified using the discrete model.

In order to verify spacing properties, we need a more accurate modeling of the approach initiation procedure. To this end, we extend the discrete model of the SATS HVO concept with continuous variables that encode the geometry of the SCA and the aircraft speed performances. Before that, we formally specify the spacing requirements.

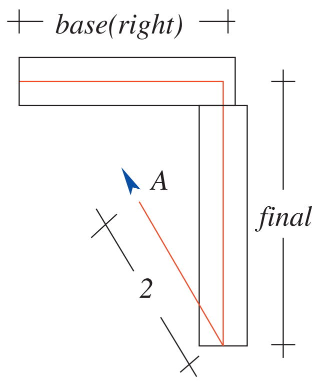


Figure 4: Linear distance from IAF

SPACING REQUIREMENTS

The term *spacing* refers to linear separation of an aircraft with respect to a lead aircraft. If both aircraft are not flying the same approach, spacing is usually computed relative the merging point of their linear trajectories. For instance, in a symmetric SCA, if the trail and lead aircraft are on opposite initial approach fixes their spacing is 0, although their Euclidean distance is twice the length of the of the base segments. Note that, independently of the initial Euclidean distance, if both aircraft start the approach at roughly the same time and speed, they will have a conflict at the merging point.

Assume that the geometry of the SCA is described by $base(left)$, $base(right)$, $final$, $maz(left)$, and $maz(right)$, which are the lengths of the left and right base segments, final segment, and left and right missed approach zones, respectively. We define $D_A(t)$ as the linear distance at time t of an aircraft A from its initial approach fix. For instance, in Figure 4,

$$D_A(t) \quad = \quad base(right) + final + 2. \tag{1}$$

In a symmetric SCA, i.e., $base(left) = base(right)$ and $maz(left) = maz(right)$, the spacing at time t between an aircraft A and its lead aircraft B is simply defined as $D_B(t) - D_A(t)$. However, in the general case, we must consider the difference in length of the base segments. Hence, if B is before A in the landing sequence, the spacing between A and B is defined as

$$S_{A \to B}(t) \quad \equiv \quad D_B(t) - D_A(t) + base(iaf_A) - base(iaf_B). \tag{2}$$

Now, we specify the spacing requirements to be formally verified.

Proposition 1. *Under nominal operations, aircraft A and B on final approach at time t, such that B is the lead aircraft of A, satisfy the following spacing requirement:*

$$S_T \quad \leq \quad S_{A \to B}(t). \tag{3}$$

Proposition 2. *Under nominal operations, A and B on final approach, on missed approach at the same fix at time t, such that B is before A in the landing sequence, satisfy the following spacing requirement:*

$$S_{MAZ} \quad \leq \quad S_{A \to B}(t). \tag{4}$$

The constants S_T and S_{MAZ} are the theoretical spacing that the concept guarantees on final approach and missed approach, respectively. These constants are determined by the geometry of the SCA, the minimum and maximum speed of the aircraft, i.e., $v_{\min}$ and $v_{\max}$, and the initial spacing between the aircraft, i.e., S_0, as follows:

$$S_T \quad \equiv \quad S_0 - (L_{max} + final - S_0)\Delta_v, \tag{5}$$

$$S_{MAZ} \quad \equiv \quad \min(L_{min} + final - L_{maz}\Delta_v, 2S_0 - (L_{max} + final + L_{maz} - S_0)\Delta_v), \tag{6}$$

where

$$\begin{aligned} L_{min} &\equiv \min(base(left), base(right)), && (7)\\ L_{max} &\equiv \max(base(left), base(right)), && (8)\\ L_{maz} &\equiv \max(maz(left), base(right)), && (9)\\ \Delta_v &\equiv \frac{v_{\max} - v_{\min}}{v_{\min}}. && (10) \end{aligned}$$

HYBRID MODEL

The hybrid model of the SATS HVO concept extends the discrete state of the original model with the following continuous variables:

- A current time t that evolves in a continuous way.
- For each aircraft A on final approach or missed approach, the linear distance from its IAF, i.e., $D_A(t)$. We assume that the speed of an aircraft may vary with time in the interval $[v_{\min}, v_{\max}]$. Therefore, the value of $D_A(t)$ is constrained by

$$(t_1 - t_0)v_{\min} \leq D_A(t_1) - D_A(t_0) \leq (t_1 - t_0)v_{\max}, \tag{11}$$

  if $t_0 \leq t_1$ (t_0 and t_1 are measured in the same approach operation).

These continuous variables allow us to state the approach initiation rule in a more precise way:

- *Approach initiation for vertical and lateral entry (left and right)*: An aircraft A may initiate the approach when (a) it is the first aircraft in the landing sequence or (b) its lead aircraft B is already on the final approach (base or final segments) and

$$S_0 \leq S_{A \to B}(t). \tag{12}$$

Other transitions have to be modified as well to handle the new variables:

- *Merging*: An aircraft A in the base segment turns to the final segment when

$$D_A(t) = base(iaf_A). \tag{13}$$

- *Missed approach initiation*: An aircraft A in the final segment may go to the missed approach zone when it is the first aircraft in the landing sequence and

$$D_A(t) = base(iaf_A) + final. \tag{14}$$

- *Landing*: An aircraft A in the final segment may land if it is the first aircraft in the landing sequence, there is no other aircraft in the runway, and

$$D_A(t) = base(iaf_A) + final. \tag{15}$$

- *Determination of lowest available altitude (left and right)*: An aircraft A on missed approach may go to the holding fix at the lowest available altitude when

$$D_A(t) = base(iaf_A) + final + maz(mahf_A). \tag{16}$$

In the next section, we show how Propositions 1 and 2 can be mechanically verified on this hybrid transition system.

MECHANICAL VERIFICATION

The discrete model of the SATS HVO concept was written in PVS and verified using a state exploration PVS tool called Besc [Ref. 5]. Roughly speaking, Besc is a basic explicit model checker, written and formally verified in PVS.[3] Early attempts to analyze the hybrid transition system described in this paper, using a hybrid model checker, e.g., HyTech [Ref. 4], were unsuccessful due to the complexity of the SATS HVO model. We tried a different approach: we encoded the hybrid transition system as a discrete one and explored it using Besc.

We first note that the discrete system is a valid abstraction of the SATS HVO concept. From a high level, all the reachable states in the hybrid system are reachable in the discrete system (modulo the common discrete variables). Of course, the converse is not true: not all the reachable states of the discrete system are reachable in the hybrid system; in particular, those states violating the spacing requirements should not be reachable in the hybrid system. Therefore, if we take all the reachable states in the discrete system and eliminate those that do not satisfy the continuous behavior expressed by Formulas (12)–(16), we should still have a valid abstraction of the SATS HVO concept.

Instead of eliminating states, we simply add the continuous behavior as constraints to the reachable states in the discrete system at the same time as the transitions take place. For instance, after a *Merging* rule, according to Formula (13), it should hold that

$$\mathit{base}(\mathit{iaf}_A) \;\leq\; D_A(t) \;\leq\; \mathit{base}(\mathit{iaf}_A) + \mathit{final}. \tag{17}$$

The semantics of a constrained state is that it is a valid reachable state if it is reachable in the discrete system and, moreover, all its constraints hold. The verification objective is to show that for each one of these hybrid reachable states, Propositions 1 and 2 hold.

Hybrid System as a Constrained Discrete System

In order to write the hybrid system as a discrete transition system, the continuous behaviors is encoded using *symbolic* constraints. A PVS data type, called `Constraint`, is inductively defined according to the following grammar:

$$\begin{aligned}
A, B &::= 1, 2, \ldots && (18)\\
s &::= \mathit{left} \mid \mathit{right} \mid \mathit{iaf}_A \mid \mathit{mahf}_A && (19)\\
T &::= t \mid T_A && (20)\\
e, f &::= T \mid D_A(T) \mid \mathit{base}(s) \mid \mathit{final} \mid \mathit{maz}(s) \mid S_0 \mid L_{min} \mid L_{max} \mid L_{maz} \mid S_{A\to B}(T) \mid e + f && (21)\\
\texttt{Constraint} &::= e \leq f && (22)
\end{aligned}$$

We use the variable T_A to denote the time when aircraft A initiates the approach.

The global state of the SCA is extended with a new field `constraints`, which is a list of `Constraints` that hold at a particular state. The hybrid transition system described before is encoded as follows:

- *Approach initiation for vertical and lateral entry (left and right)*: Let A be the aircraft that initiates the approach. The following symbolic constraints are added to `constraints`:
  - The fact that A is in the base segment, i.e,

$$\begin{aligned} T_A &\leq t, && (23)\\ D_A(t) &\leq \mathit{base}(\mathit{iaf}_A). && (24)\end{aligned}$$

  - If B is the lead aircraft of A, the fact that the aircraft are spaced at time T_A, i.e.,

$$\begin{aligned} T_B &\leq T_A, && (25)\\ S_0 &\leq S_{A\to B}(T_A). && (26)\end{aligned}$$

  - For all aircraft C on missed approach, the fact that C was ahead of A:

$$\mathit{base}(\mathit{iaf}_A) + \mathit{final} \;\leq\; D_C(T_A). \tag{27}$$

[3]Besc is available from `http://research.nianet.org/~munoz/Besc`.

- *Merging*: Let A be that aircraft that goes into the final segment. Constraint (24) is removed from `constraints`. Moreover, the fact that A is in the final segment is added to `constraints`:

$$D_A(t) \quad \leq \quad \mathit{base}(\mathit{iaf}_A) + \mathit{final}. \tag{28}$$

- *Missed approach initiation*: Let A be the aircraft that initiates the missed approach. Constraint (28) is removed from `constraints`. Moreover, the fact that A is on missed approach is added to `constraints`:

$$D_A(t) \quad \leq \quad \mathit{base}(\mathit{iaf}_A) + \mathit{final} + \mathit{maz}(\mathit{mahf}_A). \tag{29}$$

- *Landing*: Let A be the aircraft that is landing. All constraints related to A are removed from `constraints` except instances of Constraints (25) and(26) when B, the previous lead aircraft of A, is on missed approach.

- *Determination of lowest available altitude (left and right)*: Let A be the aircraft that goes to the lowest available altitude. All constraints related to A are removed from `constraints`.

State Exploration

To verify Propositions 1 and 2, we have to prove the following invariant properties for every reachable state s.

Invariant 1. *For each pair of aircraft A and B in s such that A and B are on final approach at time t, and B is the lead of aircraft A,*

$$\texttt{constraints}(s) \quad \Longrightarrow \quad S_T \ \leq \ S_{A \to B}(t). \tag{30}$$

Invariant 2. *For each pair of aircraft A and B in s such that they are on missed approach to the same fix at time t, and B is before A in the landing sequence,*

$$\texttt{constraints}(s) \quad \Longrightarrow \quad S_{MAZ} \ \leq \ S_{A \to B}(t). \tag{31}$$

We remark that the constraints are just data without any logical meaning. Thus, the invariant properties cannot be checked on the fly during the state exploration process. The mechanical verification proceeds in three different stages. In the first stage, the transition system is fully explored in PVS using the explicit model checker Besc. In order to get a finite system, the constraints are implemented as a set rather than a list to avoid repetitions. Besc reports a total of 2768 reachable states and a diameter, maximum length of a path, of 27 states.

In the second stage, we process the set of reachable states using an external tool called PVSio[4] and generate a PVS file where there is a lemma for each possible instance of Invariant 1 or Invariant 2. Without counting repetitions, 117 spacing lemmas were generated. From those, 73 lemmas are instances of the first invariant and the remaining 44 lemmas are instances of the second one.

In addition to the spacing lemmas, proof scripts, which automatically discharge these lemmas, are also generated. In the final stage of the mechanical verification task, the proof scripts are checked in batch mode via the utilities provided by ProofLite.[5] After a couple of minutes, ProofLite reports that all 117 lemmas are proved in PVS.

The proof scripts that are automatically generated are based on three lemmas. One lemma, called *T*, takes care of instances of Invariant 1. The other two lemmas, called *Maz1* and *Maz2*, handle particular cases of Invariant 2. The rest of this section sketches the proof of these lemmas.

Three Lemmas

The lemmas described here were mechanically checked in PVS. Afterward, they were integrated into a PVS strategy that mechanically discharges the automatically generated spacing lemmas.

First, we present some auxiliary properties. The time when an aircraft A initiates the final approach, i.e., when it enters the base segment, is denoted T_A. Hence, by definition,

$$D_A(T_A) \quad = \quad 0. \tag{32}$$

[4]PVSio enhances the PVS ground evaluator with input/output operations. It is available from `http://research.nianet.org/~munoz/PVSio`.

[5]ProofLite is a PVS tool for non-interactive proof checking. It is available from `http://research.nianet.org/~munoz/ProofLite`.

Therefore, Constraint (26) is equivalent to

$$S_0 + \mathit{base}(\mathit{iaf}_B) - \mathit{base}(\mathit{iaf}_A) \quad \leq \quad D_B(T_A). \tag{33}$$

Furthermore, if A is on final approach at time t, Constraint (24) and Constraint (28) yield

$$D_A(t) \quad \leq \quad \mathit{base}(\mathit{iaf}_A) + \mathit{final}. \tag{34}$$

Lemma 1 (T). *Let A and B be aircraft on final approach at time t such that B is the lead of aircraft A. It holds*

$$S_0 - (L_{max} + \mathit{final} - S_0)\Delta_v \quad \leq \quad S_{A\to B}(t), \tag{35}$$

under the hypotheses

$$T_A \quad \leq \quad t \tag{36}$$

$$S_0 + \mathit{base}(\mathit{iaf}_B) - \mathit{base}(\mathit{iaf}_A) \quad \leq \quad D_B(T_A), \tag{37}$$

$$D_B(t) \quad \leq \quad \mathit{base}(\mathit{iaf}_B) + \mathit{final}. \tag{38}$$

(Formula (36) *is the Constraint* (23)*, Formula* (37) *is the spacing constraint from Formula* (33)*, and Formula* (38) *is the instantiation of Formula* (34) *on aircraft B, which is on final approach.)*

Proof. Subtracting Formula (37) from Formula (38), we get

$$D_B(t) - D_B(T_A) \quad \leq \quad \mathit{base}(\mathit{iaf}_A) + \mathit{final} - S_0. \tag{39}$$

Using Formula (11) on A and B,

$$(t - T_A)v_{\min} \quad \leq \quad D_B(t) - D_B(T_A), \tag{40}$$

$$D_A(t) - D_A(T_A) \quad \leq \quad (t - T_A)v_{\max}. \tag{41}$$

Formula 41 yields

$$D_A(t) \quad \leq \quad (t - T_A)v_{\max}. \tag{42}$$

From Formulas (39) and (40),

$$t - T_A \quad \leq \quad \frac{\mathit{base}(\mathit{iaf}_A) + \mathit{final} - S_0}{v_{\min}}. \tag{43}$$

Hence,

$$\begin{aligned}
S_{A\to B}(t) &= D_B(t) - D_A(t) + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_B) \\
&= D_B(T_A) + (D_B(t) - D_B(T_A)) - D_A(t) + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_B) \\
&\geq S_0 + (D_B(t) - D_B(T_A)) - D_A(t), \quad \text{by Formula (37),} \\
&\geq S_0 + (t - T_A)v_{\min} - (t - T_A)v_{\max}, \quad \text{by Formulas (40) and (42),} \\
&\geq S_0 - (\mathit{base}(\mathit{iaf}_A) + \mathit{final} - S_0)\frac{v_{\max} - v_{\min}}{v_{\min}}, \quad \text{by Formula (43),} \\
&\geq S_0 - (L_{max} + \mathit{final} - S_0)\Delta_v, \quad \text{by Formulas (8) and (10).}
\end{aligned}$$

□

Lemma 2 (Maz1). *Let A and B be aircraft on missed approach at time t such that B is before A in the landing sequence. Furthermore, assume that when A initiated the approach, B was on missed approach. It holds*

$$L_{min} + \mathit{final} - L_{maz}\Delta_v \quad \leq \quad S_{A\to B}(t), \tag{44}$$

under the hypotheses

$$T_A \quad \leq \quad t \tag{45}$$

$$D_B(t) \quad \leq \quad \mathit{base}(\mathit{iaf}_B) + \mathit{final} + \mathit{maz}(\mathit{mahf}_B), \tag{46}$$

$$\mathit{base}(\mathit{iaf}_B) + \mathit{final} \quad \leq \quad D_B(T_A). \tag{47}$$

(Formula (45) *is the Constraint* (23)*, Formula* (46) *is the instantiation of Constraint* (29) *on aircraft B, and Formula* (47) *is the additional assumption about aircraft A and B.)*

Proof. Subtracting Formula (47) from Formula (46), we get

$$D_B(t) - D_B(T_A) \quad \leq \quad maz(mahf_B). \tag{48}$$

Formulas (40)–(42) are derived as in Lemma 1. From Formulas (40) and (48),

$$t - T_A \quad \leq \quad \frac{maz(mahf_B)}{v_{\min}}. \tag{49}$$

Hence,

$$\begin{aligned} S_{A \to B}(t) &= D_B(t) - D_A(t) + base(iaf_A) - base(iaf_B) \\ &= D_B(T_A) + (D_B(t) - D_B(T_A)) - D_A(t) + base(iaf_A) - base(iaf_B) \\ &\geq base(iaf_A) + final + (D_B(t) - D_B(T_A)) - D_A(t), \quad \text{by Formula (47)}, \\ &\geq base(iaf_A) + final + (t - T_A)v_{\min} - (t - T_A)v_{\max}, \quad \text{by Formulas (40) and (42)}, \\ &\geq base(iaf_A) + final - maz(mahf_B)\frac{v_{\max} - v_{\min}}{v_{\min}}, \quad \text{by Formula (49)}, \\ &\geq L_{min} + final - L_{maz}\Delta_v, \quad \text{by Formulas (7), (9), and (10).} \end{aligned}$$

□

Lemma 3 (Maz2). *Let A and B be aircraft on missed approach at time t such that B is before A in the landing sequence. Furthermore, assume that when A initiated the approach, aircraft B and X where on final approach, B was the lead of aircraft X, and X was the lead aircraft of A. It holds*

$$2S_0 - (L_{max} + final + L_{maz} - S_0)\Delta_v \quad \leq \quad S_{A \to B}(t), \tag{50}$$

under the hypotheses

$$\begin{aligned} T_A &\leq t & (51) \\ T_X &\leq T_A & (52) \\ D_B(t) &\leq base(iaf_B) + final + maz(mahf_B), & (53) \\ S_0 + base(iaf_B) - base(iaf_X) &\leq D_B(T_X), & (54) \\ S_0 + base(iaf_X) - base(iaf_A) &\leq D_X(T_A). & (55) \end{aligned}$$

(Formula (51) *is the Constraint* (23)*, Formula* (52) *is the instantiation of Constraint* (25) *on aircraft X and A, Formula* (53) *is the instantiation of Constraint* (29) *on aircraft B, and Formulas* (54) *and* (55) *are the additional assumptions about aircraft A, B, and X.)*

Proof. Subtracting Formula (54) from Formulas (53), we get

$$D_B(t) - D_B(T_X) \quad \leq \quad base(iaf_X) + final + maz(mahf_B) - S_0. \tag{56}$$

Formula (42) is derived as in Lemma 1. From Formula (32), $D_X(T_X) = 0$. Therefore, using Formula (11) on X,

$$D_X(T_A) \quad \leq \quad (T_A - T_X)v_{\max}. \tag{57}$$

From Formulas (51) and (52), $T_X \leq t$. Using Formula (11) on B,

$$(t - T_X)v_{\min} \quad \leq \quad D_B(t) - D_B(T_X). \tag{58}$$

From Formulas (56) and (58),

$$t - T_X \quad \leq \quad \frac{base(iaf_X) + final + maz(mahf_B) - S_0}{v_{\min}}. \tag{59}$$

Hence,

$$\begin{aligned}
S_{A\to B}(t) &= D_B(t) - D_A(t) + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_B)\\
&= D_B(T_X) + (D_B(t) - D_B(T_X)) - D_A(t) + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_B)\\
&\geq S_0 + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_X) + (D_B(t) - D_B(T_X)) - D_A(t),\\
&\quad \text{by Formula (54)},\\
&\geq S_0 + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_X) + (t - T_X)v_{\min} - (t - T_A)v_{\max},\\
&\quad \text{by Formulas (42) and (58)},\\
&= S_0 + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_X) - (t - T_x)(v_{\max} - v_{\min}) + (T_A - T_X)v_{\max}\\
&\geq S_0 + \mathit{base}(\mathit{iaf}_A) - \mathit{base}(\mathit{iaf}_X) - (t - T_x)(v_{\max} - v_{\min}) + D_X(T_A),\\
&\quad \text{by Formula (57)},\\
&\geq 2S_0 - (t - T_x)(v_{\max} - v_{\min}), \quad \text{by Formula (55)},\\
&\geq 2S_0 - (\mathit{base}(\mathit{iaf}_X) + \mathit{final} + \mathit{maz}(\mathit{mahf}_B) - S_0)\frac{v_{\max} - v_{\min}}{v_{\min}},\\
&\quad \text{by Formula (59)},\\
&\geq 2S_0 - (L_{max} + \mathit{final} + L_{maz} - S_0)\Delta_v, \quad \text{by Formulas (8), (9), and (10).}
\end{aligned}$$

□

Note that the conclusions of Lemmas 2 and 3 could be replaced by

$$\min(L_{min} + \mathit{final} - L_{maz}\Delta_v, 2S_0 - (L_{max} + \mathit{final} + L_{maz} - S_0)\Delta_v) \quad \leq \quad S_{A\to B}(t). \tag{60}$$

Furthermore,

$$S_{MAZ} \quad = \quad 2S_0 - (L_{max} + \mathit{final} + L_{maz} - S_0)\Delta_v, \tag{61}$$

when

$$1 + \frac{v_{\min}}{v_{\max}} \quad \leq \quad \frac{L_{min} + \mathit{final}}{S_0}, \tag{62}$$

and

$$S_t \quad \leq \quad S_{MAZ}, \tag{63}$$

when

$$L_{maz}\Delta_v \quad \leq \quad S_0. \tag{64}$$

CONCLUSION

This papers proposes a hybrid model that extends the discrete model presented in [Ref. 2]. In contrast to the original model, the proposed model enables the verification of safety spacing requirements of SATS HVO operations. To this end, aircraft performances, such as ground speed ranges, and information about the SCA geometry, such as length of the approach segments, were integrated into the original model. Thus, in the hybrid model, the concept of operations is described by the continuous dynamics of aircraft and the discrete events within the SCA. Using theorem proving and model checking techniques, we have exhaustively explored the hybrid model and mechanically verified spacing requirements over all nominal operations.

The SATS HVO development, excluding the PVS tools Besc, PVSio and ProofLite, is about 2800 lines of PVS specification and lemmas and 6500 lines of proofs. From these, 1600 lines of lemmas and 5900 lines of proofs were automatically generated using the PVS tools.

From a practical point of view, the analytical formulas presented in this paper, e.g., Formulas (5) and (6), can be used to configure a nominal SCA and the parameters of the baseline procedure for self-separation. For instance, consider a

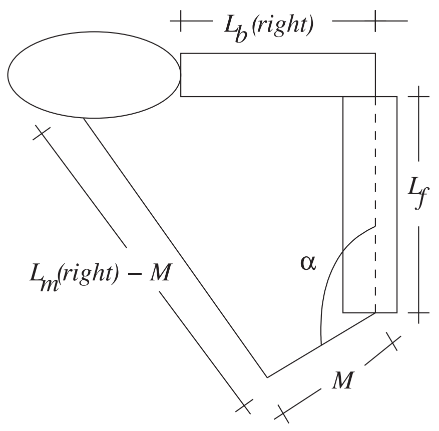


Figure 5: Nominal SCA

symmetric nominal SCA where $base(\mathit{left}) = base(\mathit{right}) = 5$ nm, $\mathit{final} = 10$ nm, and $maz(\mathit{left}) = maz(\mathit{right}) = 13$ nm. If the initial separation S_0 is 6 nm and $v_{\min} = 90$ kt, $v_{\max} = 120$ kt, then

$$L_{min} = L_{max} = 5 \text{ nm}, \tag{65}$$

$$L_{maz} = 13 \text{ nm}, \text{ and} \tag{66}$$

$$\Delta_v = \frac{120-90}{90} = \frac{1}{3}. \tag{67}$$

The value of S_T is computed using Formula (5):

$$S_T = 6 - \frac{5+10-6}{3} = 3 \text{ nm}. \tag{68}$$

This configuration of the SCA satisfies Formula (62). Therefore, the value of S_{MAZ} can computed using Formula (61):

$$S_{MAZ} = 12 - \frac{5+10+13-6}{3} = 4.66 \text{ nm}. \tag{69}$$

Hence, if the initial spacing of the trail aircraft with respect to the lead aircraft is 6 nm, the SATS HVO concept of operations guarantees a minimum spacing of 3 nm on final approach and 4.66 nm on missed approach.

The analysis used in this paper can be extended to study Euclidean separation of aircraft on final approach and missed approach. Figure 5 illustrates a nominal SCA where aircraft on missed approach turn toward their missed approach zone α degrees with respect to the runway, fly a straight trajectory of M nautical miles, and then turn to their MAHF. A geometric analysis reveals that

$$M = \frac{\min(S_T, S_{MAZ})}{2} \tag{70}$$

achieves maximum separation for an arbitrary α. In this case, the minimum Euclidean distance D_α that the concept guarantees for an aircraft on final approach and an aircraft on missed approach is given by

$$D_\alpha = M\sqrt{2(1-\cos\alpha)}. \tag{71}$$

In the example above, the optimal value of M, given by Formula (70), is 1.5 nm. The minimum Euclidean distance between an aircraft on final approach and an aircraft on missed approach, for different values of α, is computed using Formula (71):

- $D_{60^o} = 1.5$ nm.
- $D_{90^o} = 2.12$ nm.
- $D_{120^o} = 2.59$ nm.

Increasing the initial spacing S_0 to 7 nm yields the following values: $S_T = 4.33$ nm, $S_{MAZ} = 7$ nm, $M = 2.16$ nm, $D_{60^o} = 2.16$ nm, $D_{90^o} = 3.06$ nm, and $D_{120^o} = 3.75$ nm.

The mechanical verification is necessary to make sure that no cases were forgotten. For instance, the fact that Lemmas 1, 2, and 3 are sufficient to prove the spacing requirements for all nominal scenarios is shown by enumerating all the possibilities (in this case 117) and mechanically proving all of them using these 3 lemmas. Formal proofs are the ultimate guarantee that the mathematical development presented here is correct.

REFERENCES

1. T. Abbott, K. Jones, M. Consiglio, D. Williams, and C. Adams. Small Aircraft Transportation System, High Volume Operation concept: Normal operations. Technical Report NASA/TM-2004-213022, NASA Langley Research Center, NASA LaRC Hampton VA 23681-2199, USA, 2004.

2. G. Dowek, C. Muñoz, and V. Carreño. Abstract model of the SATS concept of operations: Initial results and recommendations. Technical Report NASA/TM-2004-213006, NASA Langley Research Center, NASA LaRC,Hampton VA 23681-2199, USA, 2004.

3. *Federal Aviation Regulations/Aeronautical Information Manual*, 1999.

4. T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

5. C. Muñoz, G. Dowek, and V. Carreño. Modeling and verification of an air traffic concept of operations. *Software Engineering Notes*, 29(4):175–182, 2004.

6. SATS Program Office. Small aircraft transportation system program plan. http://sats.larc.nasa.gov/documents.html, 2001.

7. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992.

MODELLING “SYSTEMC” SCHEDULER BY REFINEMENT

Dominique Cansell
LORIA, Université de Metz

Dominique Méry and Cyril Proch
LORIA,Université Henri Poincaré Nancy 1

ABSTRACT

Systems on Chip, or shortly SoCs, and SoC architectures denote a challenging set of problems of specification, modelling techniques, security issues and structuring questions. Our methodology, for designing models of (SoC) system from requirements, leads to formally justify hints on the future architectural choices of that system; it is based on the B event-based method, which integrates the incremental development of models using a theorem prover to validate each step of development called refinement. The target system is generally expressed using a programming language notation like SystemC; the SystemC language is used by electronic designers to describe different parts of the system (hardware and software); SystemC constitutes a general framework for simulating and validating the design of the system under construction. The semantics of SystemC is based on its scheduling algorithm described in the language reference manual and we develop a B model of the scheduling. The B *scheduling* model left unspecified parameters depending on the simulated SystemC program and those parameters are instantiated from the operational semantics of the developed SystemC program. By instantiation, we obtain a B abstract model of the simulated program and we can study properties of the SystemC program by simulation. B models are completely validated by the proof assistant of the event-B method. Finally, our models provide a sound framework for understanding the scheduling process.

Keywords. Event B method, refinement, scheduler, operational semantics, systemC

Pour supprimer le numéro de page de cette page Supprimer complètement le numéro de page.

INTRODUCTION

Modelling the SystemC Scheduler

The refinement of events-based models provides a general framework for developing systems from requirements and for expressing the semantical relationship between views of a system; the main idea is to begin the development by a very abstract view or model and to state the fundamental properties required by the system. The goal of the refinement-based development is to produce a formal validated model of the system in an incremental way. Benefits of refinement are numerous and first we underline the control of proof complexity by diffusion through the refined models. Second, the refinement process should start from a very abstract view of the system that leads to the possibility to tackle non trivial systems. The main objective is to write a B event-based model [3] of the SystemC [19] scheduler; the modelling is the part of a general refinement-based methodology for developing systems on chip from requirements to SystemC-like systems. First, we develop a B event-based model of the scheduler defined in the reference manual of SystemC and we let informations on the simulated program in parameters; the refinement makes possible the production of a precise model for the scheduler. Second, since the scheduler's model has parameters left unspecified, we can instantiate the scheduler for a specific SystemC program. Subsequently, the resulting B event-based model is a formal model of the global system made up of the scheduler and the particular program; the resulting model can be used in further developments and can be compared to another instantiated model. It means that the generic model provides a framework for defining the operational semantics for the simulation process, as defined in the reference manual. Since the scheduler is modelled as a generic model, it is defined and developed only once and the user should only define the parameters specific to the give SystemC program. Moreover, the resulting model is completely validated by the proof process. Objectives of the paper can be summarized as follows:

- To provide an (formal) operational semantics for the SystemC scheduler and hence for the simulation of each SystemC program.
- To use the refinement for capturing the semantics of the scheduler.
- To validate the correctness of the translation of B models into SystemC modules.

Proof-based incremental modelling

Proof-based development methods [4, 2] integrate formal proof techniques in the development of software systems. The main idea is to start with a very abstract model of the system under development. Details are gradually added to this first model by building a sequence of more concrete ones. The relationship between two successive models in this sequence is that of *refinement* [4, 2]. The essence of the refinement relationship is that it preserves already proved *system properties* including safety properties and termination.

A development gives rise to a number of, so-called, *proof obligations*, which guarantee its correctness. Such proof obligations are discharged by the proof tool using automatic and interactive proof procedures supported by a proof engine [8].

At the most abstract level it is obligatory to describe the static properties of a model's data by means of an "invariant" predicate. This gives rise to proof obligations relating to the consistency of the model. They are required to ensure that data properties which are claimed to be invariant are preserved by the events or operations of the model. Each refinement step is associated with a further invariant which relates the data of the more concrete model to that of the abstract model and states any additional invariant properties of the (possibly richer) concrete data model. These invariants, so-called *gluing invariants* are used in the formulation of proof obligations related to the refinement.

The goal of a B development is to obtain a *proved model*. Since the development process leads to a large number of proof obligations, the mastering of proof complexity is a crucial issue. Even if a proof tool is available, its effective power is limited by classical results over logical theories and we must distribute the complexity of proofs over the components of the current development, e.g. by refinement. Refinement has the potential to decrease the complexity of the proof process whilst allowing for traceability of requirements.

B models rarely need to make assumptions about the *size* of a system being modelled, e.g. the number of nodes in a network. This is in contrast to model checking approaches [7]. The price to pay is to face possibly complex mathematical theories and difficult proofs. The re-use of developed models and the structuring mechanisms available in B help in decreasing the complexity. Where B has been exercised on known difficult problems, the result has often been a simpler proof development than has been achieved by users of other more monolithic techniques.

A short introduction to B event-based notations

The B event language is based on substitutions; a substitution states the transformation of state variables from a possible pre-state to a possible post-state. In our B models, we use specific substitutions; the substitution $x := E(x)$ denotes the transformation leading to the updating of the state variable x according to the value of $E(x)$ and the substitution $x :\in A(x)$ denotes the updating of the state variable x according to a value of $A(x)$ (a set depending on the pre-value of x).The Before-After predicate of a substitution $P(x, x')$ defines the relation between values of variables before substitution (x) and values of variables after substitution (x'). For the substitution $x := E(x)$, the predicate $P(x, x')$ is $x' = E(x)$ whereas for the substitution $x :\in A(x)$, the predicate is $x' \in A(x)$. Each event has a guard controlling the substitution and the occurrence of the event. A Before-After predicate of event is defined from Before-After predicate of substitution and guard of event. We denote by $S(x)$ any substitution form and an event is built with respect to three schemata recalled in the figure 1.

Finally, the B model language provides the way to define a B event-based model. A (abstract) model is made up of a part defining mathematical structures related to the problem to solve and a part containing elements on state variables, transitions and (safety and invariance) properties of the model. Proof obligations

Event : E	Guard	Before-After Predicate
begin $S(x)$ **end**	$true$	$P(x, x')$
select $G(x)$ **then** $S(x)$ **end**	$G(x)$	$G(x) \;\; \wedge \;\; P(x, x')$
any t **where** $G(t, x)$ **then** $S(x)$ **end**	$\exists\, t \cdot (\, G(t, x))$	$\exists\, t \cdot (\, G(t, x) \;\; \wedge \;\; P(x, x', t)\,)$

Figure 1: Definition of events and before-after predicates of events

Name	Syntax	Definition
Binary Relation	$s \leftrightarrow t$	$\mathcal{P}(s \times t)$
Domain	$\mathsf{dom}(r)$	$\{a \mid a \in s \wedge \exists b.(b \in t \wedge a \mapsto b \in r)\}$
Codomain	$\mathbf{ran}(r)$	$\mathsf{dom}(r^{-1})$
Co-restriction	$r \rhd t$	$r;\ \mathbf{id}(s)$
Anti-co-restriction	$r \;\, ⩥ t$	$r \rhd (ran(r) - t)$
Image	$r[w]$	$\mathbf{ran}(w \lhd r)$
Overwrite	$q \mathbin{<\!\!\!+} r$	$(\mathsf{dom}(r) ⩤ q) \cup r$
Partial Function	$s \mathrel{+\!\!\!\!\to} t$	$\{r \mid r \in s \leftrightarrow t \;\wedge\; (r^{-1}; r) \subseteq \mathbf{id}(t)\}$
Total Function	$s \longrightarrow t$	$\{f \mid f \in s \mathrel{+\!\!\!\!\to} t \;\wedge\; \mathsf{dom}(f) = s\}$

Figure 2: B set notations

are generated from the model to ensure that properties are effectively holding: it is called *internal consistency* of the model. A model is assumed to be closed and it means that every possible change over state variables is defined by transitions; transitions correspond to events observed by the specifier. A model m is defined as follows. A model has a name m; the clause **sets** contains definitions of sets of the problem; the clause **constants** allows one to introduce information related to the mathematical structure of the problem to solve and the clause **properties** contains the effective definitions of constants: it is very important to list carefully properties of constants in a way that can be easily used by the tool. Another point is the fact that sets and constants can be considered like parameters and extensions of the B method exploit this aspect to introduce parameterization techniques in the development process of B models. The second part of the model defines dynamic aspects of state variables and properties over variables using the invariant called generally inductive invariant and using assertions called generally safety properties. The invariant $I(x)$ types the variable x, which is assumed to be initialized with respect to the initial conditions and which is preserved by events (or transitions) of the list of events. Conditions of verification called proof obligations are generated from the text of the model using the first part for defining the mathematical theory and the second part is used to generate proof obligations for the preservation of the invariant and proof obligations stating the correctness of safety properties with respect to the invariant.

The B event-based method includes the B data modelling language, the B events language and the B models language. The figure 2 gives set-theoretical notations of the B data modelling language and it borrows notations and concepts of Bourbaki's group. If f is a function then the substitution $f(x) := E$ is equivalent to $f := f \mathbin{<\!\!\!+} \{x \mapsto E\}$.

Due to the lack of space, we do not introduce formally the refinement models and they will be effectively used later. A complete introduction of B can be found in [6].

Applications to SoC development

The scheduler model was developed for validating a system on chip produced for measuring the service performance (TS level) in a DVB environment. Formal modelling techniques [12, 1, 5] provide hints on the architecture of the future system and the formal model has been developed using the B event-based method; the resulting B model provides an invariant, which is incrementally built and validated through the refinement process and the details are extracted from the documentation [9]. However, the resulting B model should be translated into an equivalent code and the question of the adequacy of the resulting code with the B model should be solved by defining a semantical framework for asserting the semantical relationship. It is why we have developed the B model for the SystemC scheduler. Our case study is a monitoring tool for measurement in Digital Video Broadcasting Television (DVB-T) and problems are related to the number of computations and real-time constraints. The implementation of this tool is driven by the hierarchy derived from invariant of models. The refinement allows us to classify parameters into a consistent hierarchy; the hierarchy has properties for deriving a so-called abstract architecture for the system. The hierarchy of the abstract model is not falsified by the hierarchy of the concrete one, thanks to the refinement. Obviously, events of the model can be used to derive algorithmic methods for computing the value of each parameters. Explanations to non specialists of refinement are given through graphs, which capture the relation between parameters. The project includes colleagues of the electrical engineering department and three industrial partners; the project leads to the effective design of a tool correct with respect to the hierarchy among parameters and the B event-based method helps in validating the final choice of implementation. However, it is out of the scope of the current paper which focuses on the model of the scheduler.

Related works

The definition of an operational semantics is not new [15, 17]; the main fact is that we use the B event-based methodology for writing the abstract scheduler; for instance, the ASM language is used to define the simulation semantics of SystemC [15, 10] as the semantics of SpecC [13], an equivalent language of SystemC, or semantics of VHDL [11]. Unfortunately, these works [15, 10] consider the scheduler of SystemC V1.0 which is really different of the actual version (V2.0). The major goals of these works are the definition of precise specifications for future implementation of a scheduler or to investigate SystemC interoperability with Verilog, SpecC and VHDL. Our goal is to provide a formal semantics to the SystemC scheduler and we use the B framework for expressing the semantics. A second difference is that we write incrementally the operational semantics and the incremental process improve the understanding of the scheduler. Finally, the resulting B event-model for the simulation semantics can be used as a parametric framework for analysing a specific SystemC program and this point is not addressed elsewhere in the literature. Others works [18, 16] aim to develop abstract models of SystemC programs and use model checking techniques; those approaches are verification-oriented and we are dealing mainly with design-oriented ones.

Summary

Section 2 describes the SystemC programming language and its concepts; the principles of simulation are sketched by the simulation algorithm. Section 3 reports the incremental development of the SystemC scheduler using the refinement process; the section is the main technical aspect of the paper. A simple example illustrates the technique of model instantiation in the section 4. Section 5 concludes the work.

SYSTEMATIC B MODELS FOR SYSTEMC SIMULATION

Requirements for the SystemC Simulation

SystemC [19, 14] is a set of C++ class definitions with hints for using these classes. The SystemC library of classes and simulation kernel extend C++ to enable the modelling of systems. Extensions include handling for concurrent behavior, time sequenced operations, data types for describing hardware, structure hierarchy and simulation support. The core language consists of an event-driven simulator as the base (scheduler). The scheduler uses events and processes.

SystemC: Quick Overview

A SystemC system consists of a set of modules. A *module* is a container class and provides the ability to describe structure. Module is a hierarchical entity that can have other modules or processes inside it. Modules typically contain processes, ports, internal data channels and possibly instances of other modules. Ports are used to describe structure, while channels are used to represent communication. Processes are concurrent and are used to model the functionality of the module. Processes are contained inside modules and are particular methods of modules. SystemC provides different process abstractions for hardware and software designers. Channels or signals handle communications between processes but communications between processes inside different modules is supported by ports, interfaces and channels. The port of a module is the object through which the process accesses a channel. Events are the basic synchronization objects for processes. Processes are triggered with respect to sensitivity on events. Concretely, an event is used to represent a condition that may occur during the simulation and to control the triggering of processes. Static sensitivity is defined before simulation starts but dynamic sensitivity is defined after simulation starts and can be altered during simulation.

SystemC: Execution Semantics

The function `sc_main()` is the entry point from the library to the user's code (as the function `main()` in C++ programs). Elaboration is defined as the execution of the `sc_main()` function from the start of `sc_main` to the first invocation of scheduler. During elaboration, the structural elements of the system are created and connected throughout the system hierarchy. The structure of the system is created during elaboration time and does not change during simulation.

Before first invocation of scheduler, *initialization* is the first step of simulation. Each process is executed (you can turn off initialization for particular processes with calls of method `dont_initialize()`) during initialization. The order of execution of processes is unspecified but two simulations run using the same version by the same simulator must yield identical results. The next figure presents an example of SystemC modules with concurrent processes, channels and events.

```
#include ''systemc.h''

SC_MODULE(my_module) {
  sc_in<bool> port1;
  sc_out<bool> port2;
  event e2,e3; // events declaration
  sc_signal<int> count; // intern channel

  void proc1() {
     if (count.read() < 10) {
       count.write(count.read()+1);
       e2.notify(); // immediate notification
     } else {
       e3.notify(5,SC_NS); // timed notification
     }
  }

  void proc2() {
     if (count.read() < 11) {
       count.write(count.read()+2);
     } else {
       e3.notify(4,SC_NS); // timed notification
     }
  }

  void proc3() {count.write(0);}
}

  SC_CTOR(my_module) {
    count.write(0);
    SC_METHOD(proc1); sensitive << count;
    SC_METHOD(proc2); sensitive << e2;
    dont_initialize();
    SC_METHOD(proc3); sensitive << e3;
    dont_initialize();
 }
};
```

The SystemC scheduler controls the timing and order of process execution, handles event notifications and manages updates to channels. It supports δ-cycles. A δ-cycle consists of the execution of *evaluate* and *update* phases. There may be a variable number of δ-cycles for every simulation time. SystemC processes are non-preemptive. It means that for *thread* processes, code delimited by two `wait` statements will execute without any other process interrupt and a *method* process completes its execution without interrupt by another process. The scheduler may be invoked such that it will run indefinitely. Once started the scheduler continues until either there are no more events, or a process explicitly stops it, or an exception condition occurs.

Event Notification

Events can be notified in three ways: immediate, δ-cycle delayed and timed. Immediate notification means that the event is triggered in the current evaluation phase of the current δ-cycle. A δ-cycle delayed notification means that the event will be triggered during the *evaluate* phase of the next δ-cycle, the event is scheduled for the next δ-cycle. Timed notification means that the event will be triggered at the specified time in the future.

Events can have only one pending notification, and retain no "memory" of past notifications. Multiple notifications to the same event, without an intermediate trigger are resolved according to the following rule:

$$\boxed{timed \prec \delta \prec immediate}$$

An earlier notification will always override a scheduled one to occur later, and an immediate notification is always earlier than any δ-cycle delayed or timed notification, rules imply non determinism.

Complete Algorithm of Scheduler

The semantics of the SystemC simulation scheduler is defined by the following eight steps in [14]. A δ-cycle consists of steps 2 through 4.

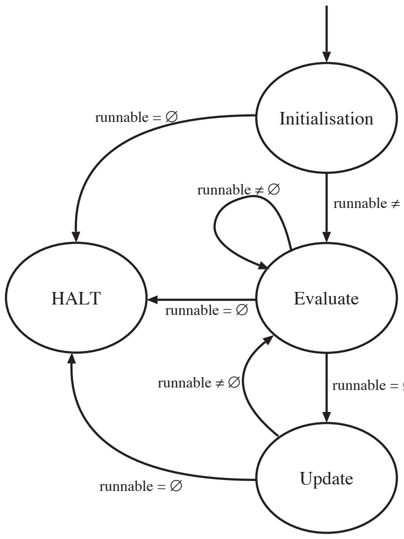
1. *Initialization Phase*:
2. *Evaluate Phase*: From the set of processes that are ready to run, select a process and resume its execution. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified.

   The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same *evaluate* phase.
3. Repeat step 2 for any other processes that are ready to run.
4. *Update Phase*: Execute any pending calls to `update()` from calls to the `request_update()` function executed in the *evaluate* phase.
5. If there are pending delta-delay notifications, determine which processes are ready to run and go to step 2.
6. If there are no more timed event notifications, the simulation is finished.
7. Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.
8. Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to the step 2.

We propose to develop the algorithm by refinement from the description of the language reference manual. By this way, we provide an abstract simulation framework which can be instantiated later for a given SystemC program. By instantiation of abstract scheduling model, we define operational semantics of SystemC programs.

INCREMENTAL CONSTRUCTION OF THE SYSTEMC SCHEDULEr

Our B models models the SystemC scheduling and different parts of algorithm are introduced by refinement. Dynamic sensitivity is not considered in our models for simplifications reasons.



model
$Abstract\_Scheduler$
sets
$PROCESSES$;
$STATE = \{init, run, stop\}$
variables
$runnable$
invariant
$runnable \subseteq PROCESSES \wedge$
$phase \in STATE \wedge$
$(phase = stop \Rightarrow runnable = \emptyset)$
initialisation
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$phase := init$

Figure 3: Automaton and header of abstract model

Abstract Model

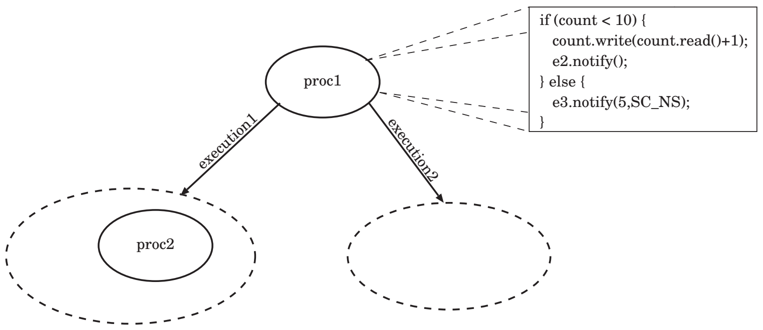
The first abstract model describes, in a very abstract way, SystemC scheduler during simulation of program. As shown in previous algorithm, scheduler has two important phases: during the *evaluate* phase, runnable processes are executed and are removed from list of runnable processes. During the *update* phase, a new list of runnable processes is built. In particular cases, processes are adding to the list in *evaluate* phase. The abstract model captures the essence of scheduler and an automaton presented in figure 3 shows the different states of our model. Only processes are considered and there are no clocks, signals and events. The abstraction plays with processes of abstract program. Our abstract model contains three distinct events to animate variables and represent SystemC scheduler reactions. The three events are represented by the three states of figure 3. Because of the abstraction level, the automaton is not deterministic, from particular state (Update for instance), many transitions are possible with the same conditions. As shown figure 3, initialization, δ-cycle and, possibly stop are modelled in the system. Remember that, when refining models, the main idea is to reduce non-determinism but we should start by a very abstract model.

More precisely, abstraction is built very simply: $PROCESSES$ is the set of processes defined in an abstract SystemC program. The abstract model uses a variable $runnable$ which is a sub-set of $PROCESSES$, runnable processes at the current time. Last, a variable $phase$ is introduced. This variable is used to separate different states of the system. Header of model with constants, properties of constants, variables, invariant and initialization of system are presented in figure 3. First, set $STATE$ and value of variable $phase$ model three different states of the system:

- $phase = init$, means than system is in *initialization* phase.
- $phase = run$, means than system is in execution phase i.e. in *evaluate* phase or *update* phase.
- $phase = stop$, means than system is halting and simulation finished.

A first interesting safety property about $runnable$ is $phase = stop \Rightarrow runnable = \emptyset$. It means than simulation is finished only, when there is no more runnable process. This is an important property of simulation presented in the language reference manual. The invariant property is preserved by events of abstract model. The $runnable$ variable is updated during *evaluate* phase, after executions of processes:

- when a process p is executed, it is suppressed from set $runnable$.
- execution of p could add new processes in the same current *evaluate* phase.



```
if (count < 10) {
   count.write(count.read()+1);
   e2.notify();
} else {
   e3.notify(5,SC_NS);
}
```

Figure 4: Possible executions of same process

In general case, a same process can have different executions between context of its current execution. For example, figure 4 shows a process with an `if then else` instruction. The process `proc1` is considered as runnable and different executions are produced by different activations of the process. Figure 4 presents the two subsets of processes generated by executions of `proc1`. These two subsets are very simple: only one process for the first and the second is empty.

The two next events model the dynamic of system and scheduling of SystemC design during simulation. Event Evaluate represents a non-deterministic choice of process p in $runnable$ (see guard of event: $runnable \neq \emptyset$) and resulting consequences of its execution. Event Evaluate suppresses processes in $runnable$ and builds a new set of runnable processes. In this abstract level, details of the list construction are not presented but the main information is stated: after each process execution a new list of processes is built.

After one or more activations of event Evaluate, value of variable $runnable$ can be the empty set ($\emptyset$). In the SystemC point of view, it means than all runnable processes have been executed and scheduler must begin its *update* phase. Event Update models the *update* phase of scheduler. Its abstracts level of modelling can not express how the new list is built but our model shows that a new list of runnable processes is built in *update* phase. Details of new list built will be presented in the first refinement.

```
Evaluate =
  select
    $phase \neq stop$
  then
    $runnable :\in \mathbb{P}(PROCESSES) \parallel$
    $phase := run$
  end
```

```
Update =
  select
    $phase = run \wedge$
    $runnable = \emptyset$
  then
    $runnable :\in \mathbb{P}(PROCESSES)$
  end
```

At last, event HALT models the ending of simulation. In the abstract model, without SystemC event notion, simulation can halt, when variable $runnable$ is empty. The invariant properties are preserved and event is consistent with invariant and requirement of SystemC scheduler. After ending of simulation (modelled by B event HALT), system is deadlocked and no event can be activated.

```
HALT =
  select
    $phase \neq stop \wedge$
    $runnable = \emptyset$
  then
    $phase := stop$
  end
```

Finally, our first model sketches the core of SystemC scheduler and simulation principles. Our abstraction presents evolution of processes (runnable thereafter not) during simulation but does not explain scheduling algorithm. Next refinement add details of SystemC simulation principles and the role of scheduler.

First Refinement: SystemC Events

The first refinement introduces SystemC events and notifications of SystemC events. Addition of SystemC events notion implies to split B event Update to specify algorithm of scheduling. Splitting concrete

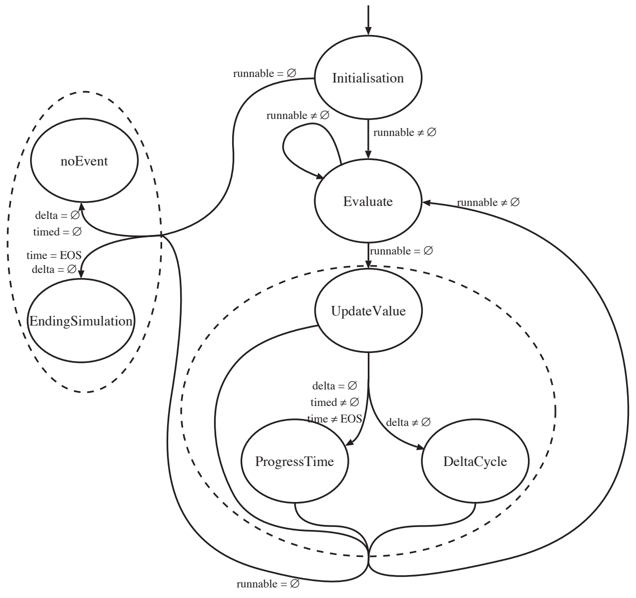


Figure 5: Concrete automaton of the scheduler

events refines abstract event Update. In the same way, abstract B event is refined by two concrete events to model different terminations. The figure 5 shows the new concrete automaton produced from the refined model. The non-deterministic transitions of abstract model (see figure 3) are now deterministic because the new refined model introduces more details. New set and constants are introduced:

- SC_EVENTS is the set of events used during execution of abstract SystemC program.
- $sensitivity$ is a relation from $PROCESSES$ to SC_EVENTS which models the static sensitivity list of each processes defined in program. Because a process can be sensitive on many events, $sensitivity$ is a relation. The relation $sensitivity$ is constant because our models do not consider dynamic sensitivity.
- EOS is a number which represents the ending-of-simulation time. Scheduler can be invoked with a integer parameter which represents the total time of simulation.
- $trigger$ is a relation from $PROCESSES$ to $\mathbb{P}(SC\_EVENTS)$. The relation represents events produce by execution of processes. Because of the code structure, $trigger$ is a relation; a conditional instruction can produce two different executions as presented in figure 4.

SystemC event and sensitivity of processes notions are introduced, we must model different kinds of event notification. To represent notifications, time must be considered in the refined model. Header of refined model is presented below:

refinement
$Event\_Scheduler$
refines
$Abstract\_Scheduler$
sets
SC_EVENTS
constants
$sensitivity, trigger, EOS$
properties
$sensitivity \in$
$PROCESSES \leftrightarrow SC\_EVENTS \wedge$
$trigger \in$
$PROCESSES \leftrightarrow \mathbb{P}(SC\_EVENTS) \wedge$
$EOS \in \mathbb{N}$
variables
$runnable, time, phase,$
$timed, \delta$

invariant
$time \in \mathbb{N} \wedge$
$EOS \geq time \wedge$
$timed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$\forall t.(t \in ran(timed) \Rightarrow t > time) \wedge$
$\delta \subseteq SC\_EVENTS \wedge$
$dom(timed) \cap \delta = \emptyset \wedge$
$(phase = stop \Rightarrow \delta = \emptyset) \wedge$
$(phase = stop \Rightarrow time = EOS \vee$
$timed = \emptyset)$
initialisation
$time := 0 \parallel$
$phase := init \parallel$
$runnable :\in \mathbb{P}(PROCESSES) \parallel$
$timed := \emptyset \parallel$
$\delta := \emptyset$

New concrete variables help to model scheduling algorithm. The two variables $timed$ and δ correspond to different kinds of event notifications. δ is the subset of SystemC events which have a pending delta-delay notification whereas $timed$ is a function from SC_EVENTS to $\mathbb{N}$ which models the subset of pending time notification events. An important invariant predicate is the disjunction of the two subsets: SystemC events can have only one pending notification and multiple notifications of the same event are resolved by priority rule. New natural variable $time$ models the current time of the system. The variable is very important and a new invariant predicate $\forall t.(t \in ran(timed) \Rightarrow t > time)$ translates that timed notifications indicate future occurrences of events.

The new concrete version of B event Evaluate must now precise behaviors of SystemC events triggered by execution of process p selected in $runnable$. First, the set E models the set of events triggered by an execution of process p ($E \in trigger[\{p\}]$). We must partition the set E:

- let i, a subset of E, the set of SystemC events related to immediate notification.
- let d, a subset of E, the set of SystemC events related to δ-delay notification.
- let t, a subset of E, the set of SystemC events related to timed notification.

These subsets are only composed of explicit invoked events. At this abstract level, the model uses only explicit SystemC events and not events produced by channels updates. These three subsets partition E as defined in the guard of B event Evaluate. The next box presents a part of B event Evaluate guard:

$E \in trigger[\{p\}] \wedge$
$t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(t) \subseteq E \ \wedge \ d \subseteq E \ \wedge i \subseteq E \wedge$
$dom(t) \cap d = \emptyset \ \wedge dom(t) \cap i = \emptyset \wedge$
$d \cap i = \emptyset \ \wedge \ dom(t) \cup d \cup i = E \wedge$

Rules of priority about multiple event notifications imply important properties on function t and on the new subset of events with pending timed notifications represented by domain of the function $newTimed$. The function is built with the function $timed$ (old set of timed notification events) and the function t which represents events with timed notifications triggered by execution of process p. Because of priority rules presented in section* , we establish these properties:

$newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
$dom(newTimed) = dom(timed \ \Leftarrow t) - (d \cup i) \wedge$
$\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(t) \Rightarrow newTimed(e) = timed(e)) \wedge$
$\forall e.(e \in dom(newTimed) \wedge e \notin dom(timed) \Rightarrow newTimed(e) = t(e))$

In the same way, variable δ is updated by rules of scheduler; immediate notifications are more prioritary than δ-notifications ($\delta := \delta \cup d - i$). In another hand, the set $runnable$ is updated by suppression of executed process p and by addition of the set of processes sensitive to immediate notifications of events of i subset. The new concrete version of B event Evaluate is finally:

```
Evaluate =
  any
    $p, E, t, d, i, newTimed$
  where
    $p \in runnable \wedge$
    $E \in trigger[\{p\}] \wedge$
    $t \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
    $dom(t) \subseteq E \ \wedge \ d \subseteq E \ \wedge i \subseteq E \wedge$
    $dom(t) \cap d = \emptyset \ \wedge dom(t) \cap i = \emptyset \wedge$
    $d \cap i = \emptyset \ \wedge \ dom(t) \cup d \cup i = E \wedge$
    $newTimed \in SC\_EVENTS \nrightarrow \mathbb{N} \wedge$
    $dom(newTimed) = dom(timed \Leftarrow t) - (d \cup i) \wedge$
    $\forall e.(e \in (dom(timed) \cap dom(t)) \Rightarrow newTimed(e) = min(\{timed(e), t(e)\})) \wedge$
    $\forall e.(e \in dom(newTimed) \wedge e \notin dom(t) \Rightarrow newTimed(e) = timed(e)) \wedge$
    $\forall e.(e \in dom(newTimed) \wedge e \notin dom(timed) \Rightarrow newTimed(e) = t(e)) \wedge$
    $dom(t) \cap \delta = \emptyset \ \wedge \ \forall x.(x \in ran(newTimed) \Rightarrow time < x)$
  then
    $runnable := (runnable - \{p\}) \cup sensitivity^{-1}[i] \ ||$
    $timed := newTimed \ ||$
    $\delta := \delta \cup d - i \ ||$
    $phase := run$
  end
```

Now, we detail the *update* phase of SystemC simulation. Because of different kind of notifications, *update* phase is more complex. The *update* phase begins when the set $runnable$ is empty. It means that all runnable processes have been executed in the previous *evaluate* phase. This important precondition was present in the guard of first abstract B event Update. The splitting of abstract B event Update produces three concrete events, updateValue, DeltaCycle, ProgressTime.

The concrete event updateValue is a non-deterministic event which adds a subset S to the set δ of events with δ-notifications. It means that sometimes, in *update* phase, SystemC scheduler produces new events notifications. Details will be added in the second refinement. Invariant properties are preserved by activation of this event. New concrete event DeltaCycle models the *update* phase due to pending δ-notifications (see guard of event:$\delta \neq \emptyset$). At this abstract level, the model explains the construction of new list of runnable processes: this is the set of processes sensitive to events with pending δ notification.

```
updateValue =
  any
    $S$
  where
    $S \subseteq SC\_EVENTS \wedge$
    $runnable = \emptyset \wedge$
    $S \cap dom(timed) = \emptyset \wedge$
    $phase = run$
  then
    $\delta := \delta \cup S$
  end
```

In the other hand, B event ProgressTime represents the *update* phase due to pending timed event notification. The event is activated only when there is no more δ-delay notifications. B event ProgressTime advances the current simulation time to the time of the earliest (next) pending time event notification. New list of runnable processes is built with $sensitivity$ relation and events which occur at new current simulation time.

```
DeltaCycle =
  select
    $runnable = \emptyset \wedge$
    $\delta \neq \emptyset \wedge$
    $phase = run$
  then
    $runnable := sensitivity^{-1}[\delta] \parallel$
    $\delta := \emptyset$
  end
```

```
ProgressTime =
  select
    $runnable = \emptyset \wedge$
    $\delta = \emptyset \wedge$
    $timed \neq \emptyset \wedge$
    $EOS \geq min(ran(timed)) \wedge$
    $phase = run$
  then
    $runnable :=$
      $sensitivity^{-1}[timed^{-1}[\{min(ran(timed))\}]] \parallel$
    $time := min(ran(timed)) \parallel$
    $timed := timed \rhd \{min(ran(timed))\}$
  end
```

Finally, the abstract event HALT is refined into two more concrete events. First, the event noEvent models the ending of simulation, because of lack of event notifications. The simulation stops because of lack of activities: no more event notifications implies no more runnable processes. Second, the event EndingSimulation models ending simulation, because of the simulation time is the ending-of-simulation time and no more events notifications can occur.

```
EndingSimulation =
  select
    $runnable = \emptyset \wedge$
    $\delta = \emptyset \wedge$
    $time = EOS \wedge$
    $phase \neq stop$
  then
    $phase := stop$
  end
```

```
noEvent =
  select
    $runnable = \emptyset \wedge$
    $\delta = \emptyset \wedge$
    $timed = \emptyset \wedge$
    $phase \neq stop$
  then
    $phase := stop$
  end
```

Second Refinement: Complete Model

Channels are introduced in this final refinement. New automaton is not presented because only some transitions (ie guards of events) are strengthened to consider adding of channels and values of them.

New constants are introduced in this final refinement. First, a new set $CHANNELS$ models the set of used channels in a SystemC program. Another set is $VALUE$ which represents the set of abstract values of considered channels. Our model introduces a subset C_EVENTS which represents the set of implicit events of the program. The implicit event is an event used by system to indicate a modification of channel's value. SystemC users can not access directly to this kind of events. Our model introduces the next properties which translate previous remarks:

$$\boxed{\begin{array}{l} C\_EVENTS \subseteq SC\_EVENTS \wedge \\ \forall S.(S \in ran(trigger) \Rightarrow S \cap C\_EVENTS = \emptyset) \end{array}}$$

Another constants are introduced in this model. The function $produce$ is a total function from $CHANNELS$ to C_EVENTS and represents implicit events triggered by modifications of channel. The current model deals with abstract channels and does not detail generation of implicit events for positive and negative sensitivity lists (keywords `sensitive_pos` and `sensitive_neg`). These kinds of sensitivity lists are only used with particular (boolean) channels and, in the same way, we could model these lists.

A new variable $value$ is introduced to represent the current values of channels. A second new variable $newValue$ is introduced to construct the new valuation of channels after *update* phase. The two variables $value$ and $newValue$ are total functions from $CHANNELS$ to $VALUE$.

New concrete version of event Evaluate considers channels. A partial function f is introduced to represent modifications of channels made by process p. The variable function $newValue$ is built with the partial function f during the *evaluate* phase. It is easy to prove that the new version of event Evaluate refined the oldest abstract version.

```
Evaluate =
  any
    $p, E, t, d, i, newTimed, f$
  where
    $p \in runnable \wedge$
    $t \in EVENTS \nrightarrow NATURAL \wedge$
    $E \in trigger[p] \wedge$
    $dom(t) \subseteq E \wedge$
    $newTimed \in EVENTS \nrightarrow NATURAL \wedge$
    $d \subseteq E \wedge i \subseteq E \wedge$
    ...
    $f \in CHANNELS \nrightarrow VALUE$
  then
    $runnable := (runnable - p) \cup sensitivity^{-1}[i] \parallel$
    $timed := newTimed \parallel$
    $\delta := \delta \cup d - i \parallel$
    $newValue := newValue \lhd\!\!- f \parallel$
    $phase := run$
  end
```

The new concrete version of event updateValue is deterministic and explain the adding of δ-delay notification during *update* phase. When values of channels are updated, δ-delay events notifications occur, when the new value is different from old value. The set δ is updated with these new δ-delay notifications. The event explains the behavior of scheduler when multiple channels modifications: only the last modification is considered.

```
updateValue =
select
  $runnable = \emptyset \wedge$
  $value \neq newValue \wedge$
  $phase = run$
then
  $value := newValue \parallel$
  $\delta := \delta \cup produce\left[\left\{\begin{array}{l} c \mid c \in CHANNELS \wedge \\ value(c) \neq newValue(c) \end{array}\right\}\right]$
end
```

The new concrete versions of other events are not too different from abstract versions and for limited size reasons we do not present the concrete versions of B events.

Producing a B model from a SystemC program

From a particular SystemC program or design, we can produce a B event model which represents the simulation of the program by the scheduler. The new B model must be a particular instantiation of abstracts models of scheduler. For each process of SystemC program we produce events (one at least) which represent execution of process. The set of events produced must refine event Evaluate from abstract models, which represent abstract executions of abstract processes.

Abstract sets and constants are concretized with particular values of program. Sets and constants of instantiated model must preserve properties of abstract sets and constants. For example, concrete set $CHANNELS$ is composed by the channels used in Systemc programs modelled. In the same way, abstract set $PROCESSES$ is the set of particular processes of current SystemC program.

EXAMPLE

Our abstract models can be instantiated to simulate execution of particular program by the SystemC scheduler. Instantiation of abstract scheduler for particular program is very easy: sets, constants and variables are specified for particular SystemC program studied. Abstract event Evaluate is split into particular events, which model execution processes of particular SystemC program. As previously announced, other abstract events are unchanged: algorithm of SystemC scheduler did not evolve with programs simulations.

Sets and constants

We use a toy example presented in section page 5. Instantiation is made with the concrete sets:

Abstract set $PROCESSES$ is instantiated by three processes (`proc1, proc2, proc3`) of the toy example. Abstract set SC_EVENTS is instantiated by SystemC events $e2$ and $e3$, defined by user, and by SystemC event e which is an implicit event. In the same way, abstract set $CHANNELS$ is instantiated by only one element, $count$ channel. Event e is gluing to channel $count$ in relation $produce$. Abstract set $VALUE$ is instantiated by the integer set.

```
$PROCESSES = \{proc1, proc2, proc3\}$
$SC\_EVENTS = \{e, e2, e3\}$
$CHANNELS = \{count\}$
$VALUE == \mathbb{N}$
```

Concrete constants are defined for the particular SystemC program considered. It is easy to show that concrete constants satisfy abstract properties of abstract constants. The major part of properties is type-checking and concrete constants trivially preserve these properties.

```
$C\_EVENTS = \{e\}$
$produce = \{count \mapsto e\}$
$trigger = \{proc1 \mapsto \{e2\},$
                $proc1 \mapsto \{e3\},$
                $proc2 \mapsto \emptyset,$
                $proc2 \mapsto \{e3\},$
                $proc3 \mapsto \emptyset\}$
$sensitivity = \{proc1 \mapsto e, proc2 \mapsto e2, proc3 \mapsto e3\}$
```

```
variables
  $runnable, time, phase,$
  $\delta, timed, value, newValue$
invariant
  $\mathbf{dom}(timed) \subseteq \{e3\} \wedge e3 \notin \delta \wedge e2 \notin \delta$
initialisation
  $time := 0 \parallel$
  $runnable := \{\text{proc1}\} \parallel$
  $phase := init \parallel$
  $timed := \emptyset \parallel \delta := \emptyset \parallel$
  $value := \{count \mapsto 0\} \parallel$
  $newValue := \{count \mapsto 0\}$
```

The invariant predicates of model precise relations between concrete instantiated variables. In the current SystemC program, only event $e3$ is concerned by time event notifications. This fact is translated into an invariant predicate $\mathbf{dom}(timed) \subseteq \{e3\}$.

Instantiation: concrete event EvaluateXXX

From structure of the three processes of listing page 5 we derive five B events:

- Process `proc1` contains conditional statement and it implies that two different executions can occur. Two different events model the process. Guards of events are disjunctive.
- As process `proc1`, process `proc2` uses a conditional statement. Equivalently, two B event simulate behaviors of process `proc2`.
- Process `proc3` does not contain conditional statement. Only one event is needed to represent its execution.

```
EvaluateProc1Then =
select
    $proc1 \in runnable \wedge$
    $value(count) < 10$
then
    $runnable := runnable - \{proc1\} \cup$
      $sensitivity^{-1}[\{e2\}] \parallel$
    $newValue := newValue \lhd\!\!\!- $
      $\{count \mapsto value(count) + 1\} \parallel$
    $phase := run$
end
```

```
EvaluateProc1Else =
select
    $proc1 \in runnable \wedge$
    $value(count) \geq 10$
then
    $runnable := runnable - \{proc1\} \parallel$
    $timed := \{e3 \mapsto$
      $\min(\mathbf{ran}(timed) \cup \{time + 5\})\} \parallel$
    $phase := run$
end
```

Finally, the two previous B events simulate executions of process `proc1` during SystemC simulation. All instructions of each block of conditional statement are translated/considered in the corresponding event. Guards of B events represent test of conditional instruction and context of simulation.

As previously, two next events are built and model behavior of process `proc2` during SystemC simulation. This two events give operationnal semantics of process `proc2`.

```
EvaluateProc2Then =
select
    $proc2 \in runnable \wedge$
    $value(count) < 15$
then
    $runnable := runnable - \{proc2\} \parallel$
    $newValue := newValue \lhd\!\!\!- $
      $\{count \mapsto value(count) + 2\} \parallel$
    $phase := run$
end
```

```
EvaluateProc2Else =
select
    $proc2 \in runnable \wedge$
    $value(count) \geq 15$
then
    $runnable := runnable - \{proc2\} \parallel$
    $timed := \{e3 \mapsto$
      $\min(\mathbf{ran}(timed) \cup \{time + 4\})\} \parallel$
    $phase := run$
end
```

The next event represents executions of process `proc3`.

```
EvaluateProc3 =
    select
      $proc3 \in runnable$
    then
      $runnable := runnable - \{proc3\} \parallel$
      $newValue := newValue \lhd\!\!\!- \{count \mapsto 0\} \parallel$
      $phase := run$
    end
```

CONCLUSION AND OPEN ISSUES

The refinement is the key concept for developing complex systems, since it starts by a very abstract model and incrementally adds new details of the set of requirements; the main result of our work is the production of a formal model for the SystemC scheduler with proved invariant properties correct with respect to the properties required by the scheduling process; the incremental proof-based construction of the formal model allows us to produce a understandable and well structured documentation for the SystemC simulation. The complexity of the proof process is indicated by the assessment:

B Models	Automatic Proofs	Interactive Proofs	%automatic/interactive P.
AbstractScheduler	4	0	100/0
Scheduler1	22	5	82/18
Scheduler2	10	2	84/16
Instanciation	20	10	66/34
Total	56	17	77/23

The SystemC scheduler allows us to instantiate parameters according to the current SystemC program to simulate and hence we obtain an instance of the scheduler that can be used for simulation and for further studies of the current instantiated SystemC program. Another result is directly related to the methodology for producing an operational semantics for a given algorithm and the refinement proves that the definition of an operational semantics can be incrementally written and can be proved by checking proof obligations. However, the result is applied to our case study which is an effective tool for measuring the quality of audio/video signals in the Digital Video Broadcasting (DVB) [9]; the tool is built from the B modelling and the SystemC code is certified by the use of a proof assistant. Further works should implement a tool for helping the manipulation of abstract scheduler and for checking conditions over the instantiation for a given SystemC program; the tool should integrate a function for defining the parameters to instantiate in the scheduler model. New case studies should be developed, as well as others properties over SoC should be taken into account like confidentiality, access control, ...

References

[1] Abraham, D., Cansell, D., Ditsch, P., Méry, D., and Proch, C. Synthesis of the QoS for digital TV services. In *IBC'05, The Netherlands* (2005).

[2] Abrial, J. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[3] Abrial, J.-R. B# : Toward a synthesis between Z and B. In *ZB'2003 - Formal Specification and Development in Z and B* (Turku, Finland, June 2003), D. Bert, J. P. Bowen, S. King, and M. Waldén, Eds., vol. 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, Springer, pp. 168 – 177.

[4] Back, R. J. R. On correct refinement of programs. *Journal of Computer and System Sciences 23*, 1 (1979), 49–68.

[5] Cansell, D., Culat, J.-F., Méry, D., and Proch, C. Derivation of SystemC code from abstract system models. In *Forum on specification & Design Languages - FDL'04, Lille, France* (Sep 2004).

[6] Cansell, D., and Méry, D. Logical foundations of the B method. *Computers and Informatics 22* (2003).

[7] Clarke, E. M., Grumberg, O., and Peled, D. A. *Model Checking*. The MIT Press, 2000.

[8] ClearSy. Web site b4free set of tools for development of b models. `http://www.b4free.com/index.php`, 2004.

[9] European Broadcasting Union. Digital video broadcasting (DVB)- measurement guidelines for DVB systems. Tech. Rep. TR 101 290 v1.2.1., ETSI, 05 2001.

[10] Gawanmeh, A., Habibi, A., and Tahar, S. An executable operational semantics for SystemC using Abstract State Machines. Tech. rep., Concordia University, Department of Electrical and Computer Engineering, mar 2005.

[11] Glässer, U., Börger, E., and Müller, W. Formal definition of an abstract VHDL'93 simulator by EA-machines. In *Formal Semantics for VHDL* (1995), C. Delgado Kloos and P. T. Breuer, Eds., Kluwer Academic Publishers.

[12] Méry, D., Cansell, D., Proch, C., Abraham, D., and Ditsch, P. The challenge of QoS for digital television services. *EBU Technical Review 302* (Apr 2005).

[13] Mueller, W., Dömer, R., and Gerstlauer, A. The formal execution semantics of SpecC. In *ISSS '02: Proceedings of the 15th international symposium on System Synthesis* (New York, NY, USA, 2002), ACM Press, pp. 150–155.

[14] Open SystemC Initiative. *SystemC 2.0.1 Language Reference Manual*, 2004.

[15] Ruf, J., Hoffmann, D., Gerlach, J., Kropf, T., Rosenstiehl, W., and Mueller, W. The simulation semantics of SystemC. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 64–70.

[16] Ruf, J., Hoffmann, D., Kropf, T., and Rosenstiel, W. Simulation-guided property checking based on a multi-valued AR-automata. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe* (Piscataway, NJ, USA, 2001), IEEE Press, pp. 742–748.

[17] Salem, A. Formal semantics of synchronous SystemC. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 376–381.

[18] SOCFV Project. System on chip formal verification home page. `http://www.ensta.fr/~hammami/resproj.SOCFV.html`, 2004.

[19] SystemC. Official web site of SystemC community. `http://www.systemc.org/`, 1999.

REAL-IT: MODEL-BASED INTERFACE DEVELOPMENT ENVIRONMENT[1]

Alexander N. Ivanov and Dmitrij V. Koznov
Saint-Petersburg State University

ABSTRACT

We present REAL-IT – model-based interface development environment with powerful support of source code generation from declarative interface models. REAL-IT provides some special modeling technique based on UML collaboration diagrams to specify data dependencies. This information is actively used for generation of advanced user interface features. REAL-IT supports iterative development process and provides guarantied consistency between different models and also between models and source code. REAL-IT development environment is integrated with UML CASE-tools (IBM Rational Rose and REAL) and development environment Microsoft Visual Studio/Visual Basic. It provides code generation for MS Visual Basic/Java and MS Access/MS SQL Server/Oracle. REAL-IT is intensively used in industry, and we analyze two industrial projects where it was successfully used.

INTRODUCTION

The model-based user interface development environment (MBUIDE) aims to provide a context in which developers can design and implement user interfaces (UIs) constructing declarative models (ref. 1). These models give the following advantages:

- They can provide more high-level description of the UI than UI descriptions provided by the other UI development tools (ref.2).
- They support the infrastructure for automating the processes of design and implementation of the UI (ref. 3).
- They facilitate the creation of methods to design and implement the UI in a systematic way (ref.1).

There are numerous MBUIDEs, for example, GENIUS (ref. 4), TEALLACH (ref. 5), MASTERMIND (ref.3)[2]. Model-based user interface development facilities are embedded into various CASE-tools, for example, ERwin[3]. Data Base Management Systems (DBMSs) – MS Access, Oracle, etc. – also have a partial support for model-based development of the UI. The model-based user interface development approach is adopted for Web-applications (ref. 7, 8).

However, MBUIDEs are not widely used since some essential problems that are related to this approach are not completely solved (ref. 1):

- There is no consensus about the set of models that is the most suitable for describing user interfaces.
- It is not easy to integrate generated UI code together with other software components.
- It is hard to estimate the quality of UI models without demonstrating the execution of UI code.

In this paper we present REAL-IT, which is a MBUIDE with powerful support of source code generation from declarative interface models. REAL-IT provides some special modeling technique, which is based on UML collaboration diagrams. The diagrams are used to specify data dependencies for future generation of advanced UI features. REAL-IT supports iterative development process and provides consistency between different models and between models and source code. REAL-IT toolset is integrated together with UML CASE-tools (IBM Rational Rose and REAL (ref. 9)) and Microsoft Visual Studio/Visual Basic. It provides code generation for MS Visual Basic/Java and MS Access/MS SQL Server/Oracle. We discuss also perspectives of applying REAL-IT for Web-applications development. Lastly, we analyze two industrial projects where REAL-IT was successfully used.

[1] This work is partially supported by RFBR grant 05-0951/07.
[2] Detailed surveys on MBUIDEs can be found in references 1 and 6.
[3] AllFusion® ERwin Data Modeler 4.1.1, http://www3.ca.com/

REAL LIFE CHALLENGES FOR MBUIDEs

In this section we highlight the technologic challenges for MBUIDEs. They are the following:

- Support of the UI iterative development process. The development of the UI is an iterative process. It means that the models should be intensively refined and MBUIDE must support their consistency through the whole refinement process. Moreover if MBUIDE provides code generation it should also provide the support of consistency for models and source code. This is because, first, models can be changed after generation has been performed and we have to regenerate the UI. Second, the generated code can be modified manually and since these modifications are not described in the models they are obviously not regenerated in the new version of the UI. This leads to the classical roundtrip problem as described in reference 10. If the model-based approach does not solve this problem it is hard to apply such approach in real practice (ref. 11).
- Balance between poor/rich models and simple/complex development process. There are two wide MBUIDEs groups. MBUIDEs of the first group implement simple process development but provide poor modeling facilities. The UI that has been modeled in these environments can be used only as a prototype. Examples of the first group of MBUIDEs are GENIUS (ref. 4), MS Access and ERwin. MBUIDEs of the second group provide rich modeling facilities but their development processes are extremely complicated. The cost of constructing model descriptions can be higher than the benefits provided by such environments. Examples of the second group of MBUIDEs are TEALLACH (ref. 5) and MASTERMIND (ref. 3).
- Integration between modeling toolset and development environment. It is not easy to split real software development process into design and implementation, as well as into designs of the UI and other software components (database, business logic, etc.). All process activities are tightly integrated together because of the iterative nature of software development process. So it is not effective isolating one of the process activity (in this case, the development of the UI) providing it with any standalone tool. For UI modeling tools it is necessary to be deeply integrated together with the development environment.

Many researches ignore the above mentioned aspects and concentrate their attention only on models (structure, notation, etc.) and modeling techniques.

REAL-IT OVERVIEW

Today there is no general software development process (ref. 12). Therefore, if we want to shift from general principles of the software development to the real approaches and toolsets we have to introduce some constraints and restrictions for our development process as well as the target software.

REAL-IT is oriented on the development of data intensive systems with database business logic – such operations as input, modify, browse of data, etc. These systems should not support complicated business functionality. The structure of the system's user actions is typical and very simple. Therefore, the corresponding part of the UI has also a typical structure.

There is a large number of industrial applications that are such systems or have corresponding subsystems. The size of code for such applications is significantly large and their development process is an unwanted routine.

We introduce the following types of windows for data intensive systems:

- Card. It is for editing information about one database object. Figure 1 (a) shows an example for this window type.

- Data browser. It is for browsing database objects. Figure 2 (c) is an example of this window type. Data browser can contain a number of different browse filters that can depend on each other.
- Relation. It is for linking database objects, which are instances of two classes connected by n:n association[4]. An example is shown in Figure 1 (b): Classes District and Street connected by n:n association, i.e. one district may be linked with many streets and one streets may be linked with many districts. If we select one district from the combo box named "District" we will see all the available streets in the left list (streets that can be linked with the district). In the right list we will see those streets that have been already linked to this district. We are able to link or unlink streets selected for this district.

Windows of these types can be combined together with a different manner: card can be invoked from the data browser and can contain other data browsers and relation windows, etc.

For these types of windows REAL-IT provides a mature UI development framework: model-based design of the user interface, generation of source code, supporting iterative development process, toolset integration with other development tools.

We can summarize three dimensions of REAL-IT following reference 1:

- Declarative models that are used for development of the UI;
- Development process of the UI;
- Environment that supports the development process.

DECLARATIVE MODELS

In terms of reference 6 REAL-IT supports domain and dialog models. We don't support user model and task model because we want to reduce the problem of model consistency. REAL-IT does not have separate presentation model because MS Visual Studio, Delphi, Power Builder and other software environments have it. We think MBUIDE has to integrate with such software environments but not duplicate them. REAL-IT is integrated with MS Developer Studio/Visual Basic.

Domain model

We use UML for creating domain model, i.e. class diagrams to model data base schema, and collaboration diagrams to capture data dependencies. We apply UML class diagrams for modeling rational database schema. Classes denote tables, attributes are columns, associations express foreign keys, and objects are table records. We will name classes and objects in the context of data modeling as "data class" and "data object".

There are a lot of dependencies in data that can not be specified in terms of class diagrams. But these dependencies are very impotent because they capture some essential properties of UI, for example, connected combo box filters in data browsers, connected combo boxes in card windows for restriction of user inputs and so on. Collaboration diagrams are very useful to specify the data dependencies.

Figure 2 (a) shows a fragment of a database schema of a business company. But this fragment does not answer to the following question: is it possible for employee to be connected with computer of another department or not? Figure 2 (b) shows a collaboration diagram, which is expressed the negative answer to this question. For Employee-card two related combo boxes will be generated to specify department and computer. Computers will be presented for selection depending on department.

REAL-IT provides the ability for more complicated modeling of data dependencies as it is presented in reference 13.

[4] Actually REAL-IT is able to work not only with binary n:n associations but also with k-ary ones where $k > 2$.

Dialog model

This model provides a conceptual description of the structure and behavior of the visual parts of the UI in terms of abstract objects while omitting the visual details (ref. 1).

The structure of each window designed in REAL-IT is typical. The developer is able to do one of the following:

- In case if window is Card, developer selects data class which properties (attributes and associates) she/he want to place on the window. Class attributes selected will be presented as input controls, its associates selected will be presented as list/combo box controls. The developer creates property pages and arranges window elements on them.
- In case if window is Data browser, developer selects data class or view[5], which objects will be presented on the window, creates and tunes browse filters.
- In case if window is Relations, developer selects two data classes, which are connected by n:n association; defines one of them to be presented as combo box; creates and tunes browse filters.

The developer can also connect designed windows with each other.

The dialog model is saved in terms of UML class diagrams, and developer can analyze these diagrams in CASE-tool. Actually, the developer very rear works with these UML diagrams. The main aim of UML representation of dialog model is to provide the storage for all REAL-IT models in one UML-repository. It simplifies the support of model consistency.

PROCESS

In this section we present the user interface development process that is supported by REAL-IT. We briefly describe the baseline of the process emphasizing on the code generation and iterative development additionally provided by REAL-IT.

Baseline

REAL-IT user interface development process starts from creating a domain model. The developer models a database schema by means of class diagrams and specifies data dependencies using UML collaboration diagrams. The domain model is developed using UML CASE-tool.

The next step is creating a dialog model. It should be stressed that the firstly created domain model serves as a foundation for building the dialog model. During the development of the dialog model the developer creates views, elaborates windows for separated data-classes and views, and links these windows with each other. He/she does this work using a special dialog wizard (not UML CASE-tool) provided by REAL-IT. From this model REAL-IT generates UML class diagrams and system source code. Generated code is later refined and integrated with other system components. This can be performed using any conventional and commonly used development environment such as MS Visual Studio, IntelliJ IDEA, etc. The next step is linking system code together with REAL-IT runtime library resulting in the target application. Figure 3 illustrates the user interface development process with REAL-IT.

Code generation

Beside the ability to generate the programming code for the user interface, REAL-IT can also produce a database schema (SQL/DDL), code for the connection of the database and the UI, and the main application window.

[5] Besides data classes we used SQL-views to as a skeleton for data browser windows.

REAL-IT bases its source code generation on a set of manually predefined templates. Such templates can be easily modified in order to meet the needs of a particular software project.

Supporting iterative development process

Because REAL-IT has only a few types of UML diagrams and stores all models in a one repository it can easily guarantee the model consistency. However, the synchronization between models and system code is more complicated.

It is impossible that a MBUIDE addresses all numerous specific features for every software project. These features should be implemented manually, in particular, by means of making modifications of generated code. REAL-IT allows save such modifications during regeneration process by providing support for the following roundtrip approaches:

- Inheritance with polymorphism. Code modifications are specified as subclasses and original classes containing generated code become superclasses. The subclasses redefine properties of superclasses using polymorphism. During the code regeneration REAL-IT replaces only superclasses, and subclasses remain unchanged that insures safety of previously made code modifications. The approach is quite suitable for Java since this language has a built in inheritance. However, if we apply this approach to Visual Basic (VB) we have to emulate inheritance. Moreover, the user interface controls in VB are not represented as VB-code but specified in some special format. Therefore, a totally different mechanism needs to support manual changes of controls' properties.
- Supporting mechanism of window controls' changes. The manual changes of a window control properties are possible to localize automatically in the generated code. In REAL-IT a special algorithm analyzes the previous version of the generated code and insures the presence of the found manual changes in the newly regenerated code.
- Registration modifications of the user in the log. When the developer modifies generated code than all his actions are saved in a special log. After regeneration these actions are automatically reproduced for the regenerated code.

The above described mechanisms are not fully automated. The developer still has to do some additional work. For example he/she provides for REAL-IT toolset an extra data that can not be extracted automatically. It should be noted that the above described approaches do not work when the volume of manual code modifications is comparable with the volume of the automatically generated code. Otherwise, synchronization algorithms may work incorrectly or add a great computational overhead to the synchronization.

ENVIRONMENT

REAL-IT is integrated with UML CASE-tool (IBM Rational Rose, REAL (ref. 9)) and MS Visual Studio/Visual Basic, as Figure 4 shows.

REAL-IT toolset is deeply integrated with MS Visual Studio/VB because this development environment has a special format for storing information related to the UI. MS Visual Studio/VB provides a special program interface to support the integration with user applications. On the contrary, when we use Java environments all we need to support integration is to generate a plain Java-code. For REAL-IT/Java we use IntelliJ IDEA. However, it is possible to use any other Java development environment.

INDUSTRIAL APPLICATIONS

REAL-IT was successfully used in the development of two industrial systems.

The first one was the information system STUDENT designed to support a faculty level business processes of Saint-Petersburg State University. This project was developed by Information Institute of Saint-Petersburg State University. The system was developed on the base of MS VB/Access/MS SQL Server by 10 employees during 3 years.

The second system was the pilot application of a government system for network management of telecommunication equipment. This project was developed by LANIT-TERCOM Ltd. (Russia) on the base of Java/CORBA/Oracle by 5 employees during 1.5 years.

Results of applying REAL-IT in these projects are presented in tables 1 and 2. The rows of these tables indicate different window types of the UI, the columns specify the manner in which the code for these widows was developed. Captions of the tables can be clarified as follows:

- “Fully generated” means that windows were fully generated by REAL-IT.
- “Manually modified” means that windows were generated by REAL-IT, changed manually and supported by roundtrip REAL-IT facilities.
- “Generated, but manually supported” means that generated windows were manually changed, but after that were developed outside of REAL-IT.
- “Manually developed” means that windows from the very beginning were developed outside of REAL-IT.

For the first project 95% of the UI has been developed in REAL-IT, about 60% has been fully generated. For the second project 87% of the UI has been developed in REAL-IT and 20% has been fully generated. In the second case REAL-IT was not so effective because this system had specific business logic (for network monitoring). This specificity makes the volume of manual changes to the generated code higher. Many windows were unique and were developed outside REAL-IT.

FOCUS ON WEB-APPLICATIONS

Widely spreading Web-applications stimulate creation of advance development environments. There are several model-based approaches in this area (ref. 7, 8).

REAL-IT is quite suitable for Web-application development because a large number of Web-systems are data intensive systems. It should be noted that WebML approach (ref. 8) and WebRatio toolset[6] are very similar to REAL-IT, but they do not support iterative development process, usage of collaboration diagrams, and other REAL-IT advantages.

At the present moment we have done the following work in order to orient REAL-IT for Web-application modeling:

- We have designed the REAL-IT runtime library for J2EE.
- We have partially ported the generator and the runtime into J2EE platform.
- We have developed the web-prototype of STUDENT system using REAL-IT.

CONCLUSIONS

The orientation of REAL-IT to the defined type of information systems allows us to find out the balance between poor/rich models and simple/complex development process. REAL-IT provides generation of non-trivial target code and quite simple development process. In STUDENT project the level of professional skills of REAL-IT/VB developers needed was significantly less than the usual VB developers.

[6] http://www.webratio.com

Industrial projects that are presented in this paper were developed in collaboration with authors of REAL-IT. In present days, the independent development team of Information Technologies Institute of Saint-Petersburg State University successfully uses REAL-IT to develop and support various information systems.

TABLES

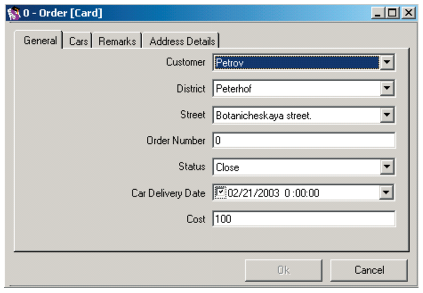
Table 1: REAL-IT in STUDENT-project.

Window type	Fully generated	Manually modified	Generated, but manually supported	Manually developed	Total
Data browser	69	16	6	0	91
Card	37	21	16	0	74
Relation	17	1	4	0	22
Others	0	0	0	6	6
Total	123	38	26	6	193

Table 2: REAL-IT in development of the pilot application of a government system for network management.

Window type	Fully generated	Manually modified	Generated, but manually supported	Manually developed	Total
Data browser	12	27	5	0	44
Card	9	22	7	0	38
Relation	0	0	1	0	1
Others	0	0	0	13	13
Total	21	49	13	13	96

PICTURES

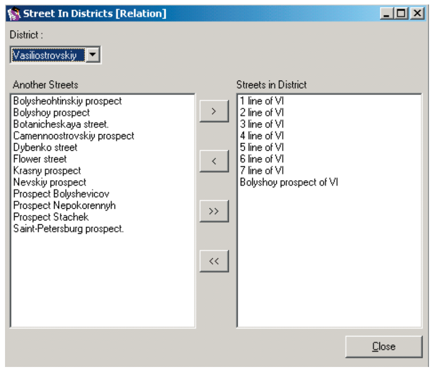


0 - Order [Card]

General | Cars | Remarks | Address Details

Customer	Petrov
District	Peterhof
Street	Botanicheskaya street.
Order Number	0
Status	Close
Car Delivery Date	✓ 02/21/2003 0 :00:00
Cost	100

(a)

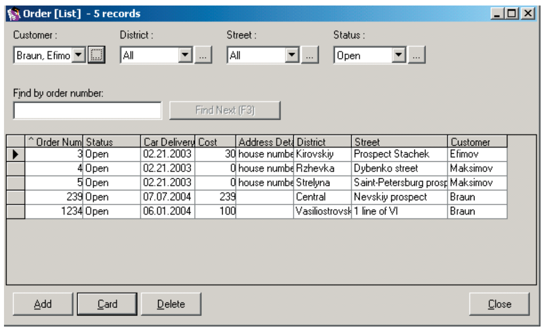


Street In Districts [Relation]

District : Vasiliostrovskiy

Another Streets	Streets in District
Bolysheohtinskiy prospect	1 line of VI
Bolyshoy prospect	2 line of VI
Botanicheskaya street.	3 line of VI
Camennoostrovskiy prospect	4 line of VI
Dybenko street	5 line of VI
Flower street	6 line of VI
Krasny prospect	7 line of VI
Nevskiy prospect	Bolyshoy prospect of VI
Prospect Bolyshevicov	
Prospect Nepokorennyh	
Prospect Stachek	
Saint-Petersburg prospect.	

(b)



Order [List] - 5 records

Customer : Braun, Efimo
District : All
Street : All
Status : Open

Find by order number:

^ Order Num	Status	Car Delivery	Cost	Address Deta	District	Street	Customer
3	Open	02.21.2003	30	house numbe	Kirovskiy	Prospect Stachek	Efimov
4	Open	02.21.2003	0	house numbe	Rzhevka	Dybenko street	Maksimov
5	Open	02.21.2003	0	house numbe	Strelyna	Saint-Petersburg prosp	Maksimov
239	Open	07.07.2004	239		Central	Nevskiy prospect	Braun
1234	Open	06.01.2004	100		Vasiliostrovsk	1 line of VI	Braun

(c)

Figure. 1: REAL-IT window types: (a) card; (b) relation; (c) data browser.

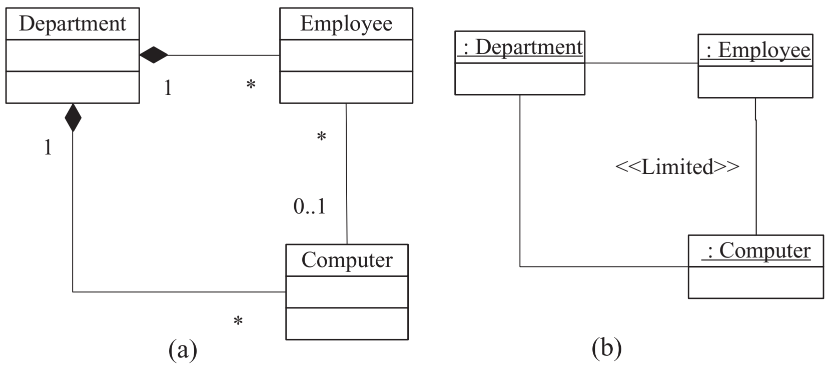


Figure 2: (a) a fragment of a database schema of a business company;
(b) a data dependence for this fragment.

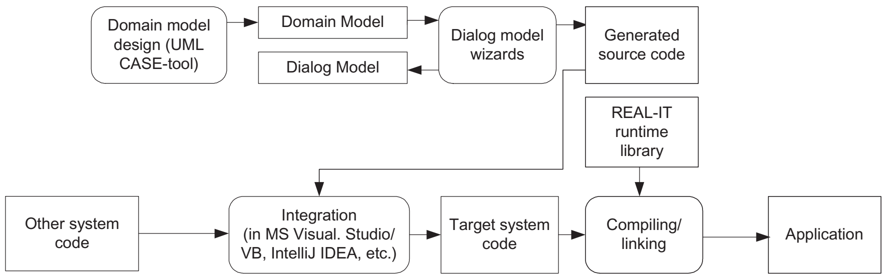


Figure 3: Baseline of user interface development process with REAL-IT.

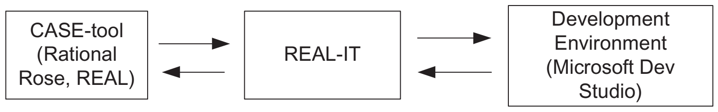

CASE-tool (Rational Rose, REAL)
REAL-IT
Development Environment (Microsoft Dev Studio)

Figure 4: REAL-IT and its environment.

REFERENCES

1. Da Silva P. P., “User Interface Declarative Models and Development Environments: A Survey”, *Lecture Notes in Computer Science*. Vol. 1946, 2000, pp. 207-226.

2. Wiecha C. and Boies S., “Generating user interfaces: principles and use of ITS style rules”, *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*, ACM Press, 1990, pp. 21-30.

3. Szekely P., Sukaviriya P., Castells P., Muthukumarasamy J., and Salcher E., “Declarative Interface Models for User Interface Construction Tools: the MASTERMIND Approach”, *IFIP Conference Proceedings,* Vol. 45, Chapman & Hall, London, UK, 1996, pp. 120-150.

4. Janssen C., Weisbecker A., and Ziegler J., ”Generating User Interfaces from Data Models and Dialogue Net Specifications”, *Proceedings of the SIGCHI conference on Human factors in computing systems*, New York, NY, ACM Press, 1993, pp. 418-423.

5. Grffths T., Barclay P., McKirdy J., Paton N., Gray P., Kennedy J., Cooper R., Goble C., West A., and Smyth M., “Teallach: A Model-Based User Interface Development Environment for Object Databases”, *Proceedings of User Interfaces to Data Intensive Systems*, September 1999, Edinburgh, UK, IEEE Press, pp. 86-96.

6. Schlungbaum E., “Model-based User Interface Software Tools: Current state of declarative models”, *GIT-GVU-96-30*, November 1996.

7. Schattkowsky T., Lohmann M., “Rapid Development of Modular Dynamic Web Sites Using UML”, *Lecture Notes in Computer Science*, Vol. 2460, 2002, pp. 336-350.

8. Ceri S., Fraternali P., Bongio A., “Web Modeling Language (WebML): a modeling language for designing Web sites”, *Computer Networks*, 33 (1-6), 2000, pp. 137-157.

9. Terekhov A. N., Romanovskii K. Yu., Koznov D. V., Dolgov P. S., and Ivanov A. N., “RTST++: Methodology and a CASE Tool for the Development of Information Systems and Software For Real-Time Systems”, *Programming and Computer Software*, Vol. 25, No. 5, 1999, pp. 276-281.

10. Aßmann U., “Automatic Roundtrip Engineering”, *Electronic Notes in Theoretical Computer Sci*ence. 82(5), 2003.

11. Koznov D., Kartachev M., Zvereva V., Gagarsky R., Barsov A., “Roundtrip Engineering of Reactive Systems”, *Proceedings of 1st International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, Paphous, Cyprous, 2004, pp.343-347.

12. Sommerville I., “Software Engineering”, *Addison-Wesley*, 6th edition, 2001.

13. Ivanov A. N., “Graphic Language for Describing Constraints on Diagrams of UML Classes”, *Programming and Computer Software*, Vol. 30, No. 4, 2004, pp. 204-208.

FLEXIBILITY IN MODELING LANGUAGES AND TOOLS: A CALL TO ARMS[1]

Eric Van Wyk and Mats Per Erik Heimdahl
Department of Computer Science and Engineering
University of Minnesota

ABSTRACT

In model-based development, the software development effort is centered around a formal description of the proposed software system; a description that can be subjected to various types of analysis and code generation. Based on years of experience with model-based development and formal modeling we believe that the following conjectures describe fundamental obstacles to wide adoption of formal modeling and the potential for automation that comes with it; (1) no single modeling notation will suit all, or even most, modeling needs, (2) no analysis tool will fit all, or even most, analysis tasks, and (3) flexible and stable tools must be made available for realistic evaluations and technology transfer. These conjectures form the basis for the call to arms outlined in this report.

To make automated software engineering techniques more useful for more types of developers and allow us to move forward as a community it is crucial that we develop the foundation for building extensible and flexible modeling language processing tools. New common-infrastructure-based approaches are needed as traditional approaches based in file-based processing of intermediate language representations are not adequate. In this report we outline and illustrate the problem and discuss a possible solution. To initiate the discussions in the community, we hypothesize that languages and tools built using higher-order attribute grammars with forwarding can serve as a basis for such flexible language processing tools; tools that will allow us to unify our efforts and help bring our collective work to a broader audience.

INTRODUCTION

Traditionally, software development has been largely a manual endeavor. Informal natural language requirements have been manually captured, a design satisfying the requirements has been manually derived, and code implementing the design has been manually coded. Recently, there has been a move away from such manual techniques to a new paradigm commonly called *model-based development*. In this paradigm, the development effort is centered around a formal description of the proposed software system—the `model' in model-based development. For validation and verification purposes, this *formal model* can then be subjected to various types of analysis, for example, completeness and consistency analysis, *e.g.,* [1], model checking, *e.g.,* [2], theorem proving, *e.g.,* [3], and test case generation, *e.g.,* [4]. There are currently several commercial and research tools that attempt to provide these capabilities—for example, the commercial tools Esterel Studio (with its graphical notation Safe State Machines) and SCADE Studio from Esterel Technologies [5], Statemate from i-Logix [6], Simulink and Stateflow from The Mathworks Inc. [7], and SpecTRM from Safeware Engineering [8]; examples of research tool are SCR [9] and RSML^{-e} [10].

Our goal has for the last decade been to dramatically increase the quality and productivity of software development for critical control systems by centering the development around fully formal models extensively supported by tools. These tools must allow engineers to specify system requirements in an appropriate and familiar notation and to *effectively* analyze the specifications to ensure that safety critical properties are satisfied. In the course of our work we developed an approach to simulation and validation of formal specifications for process-

[1] This work was partially funded by NSF CAREER Award #0347860 and NSF CCF Award #0429640.

control systems called specification-based prototyping [10]. Specification-based prototyping combines the advantages of traditional formal specifications (e.g., preciseness and analyzability) with the advantages of rapid prototyping (e.g., risk management and early end-user involvement)—goals that are now shared in the move towards model-based development. To enable specification-based prototyping, we developed the fully formal specification language RSML^{-e} (Requirements State Machine Language, without events) [11] and its execution and analysis environment NIMBUS [12]. We have successfully evaluated the capabilities on various case studies from the avionics, transportation, and mobile robotics domains [10, 11], and the environment has been used for several years in industrial research projects [12, 13, 14]. Note that our experience is primarily in the critical embedded systems domain, but we believe our observations and proposed solutions are applicable to other domains as well. Based on our years of experience, we believe that the following conjectures describe the fundamental obstacles that must be overcome for model-based development to have the dramatic real-world impact we, and many others in the community, envision. These conjectures form the basis for our call to arms outlined in this report.

Conjecture 1: No modeling language will be universally accepted, nor universally applicable. Even closely related domains, such as avionics and medical technology, have justifiably different and entrenched views of what notations and features a modeling language should have to be suitable for their domains. Nevertheless, certain *classes* of languages do have wide appeal, for example synchronous languages such as, Safe State Machines [5], SCADE [5], and SCR [9].

Conjecture 2: No verification and validation tool will satisfy all of a user's analysis needs. Analysis tools are quite specialized and new sophisticated analysis tools are constantly emerging. Somehow mating a wide and growing collection of analysis tools with a variety of modeling languages (Conjecture 1) is inevitable.

Conjecture 3: To make progress in model-based development, practicing engineers must evaluate proposed solutions on practical problems; if proposed theories, methods, and tools do not solve real problems they are of little or no value. Therefore, the methods and tools must be flexible enough to easily adapt and be improved based on what is learned from using the tools on real-world problems.

The goal with this report is to generate awareness of what we perceive to be a serious problem and attempt to build a community around an effort to develop the foundation for building extensible and flexible modeling language processing tools that can satisfy the following three goals derived from the above conjectures—successful modeling tools must (1) allow easy extension and modification of formal modeling notations to meet the domain specific needs of a class of users, (2) allow easy construction of high-quality translations from the modeling notations to a wide variety of analysis tools, and (3) enable controlled reuse of tool infrastructure to make tools extensions cost effective.

Currently, we are aware of no tools that support the easy customization (or complete redefinition) of a modeling language necessary to accommodate the needs and likes of stake-holders in different domains. Some modeling notations, notably UML [15], attempt to appeal to different domains by incorporating a wide variety of rich constructs. The richness of such languages, however, makes them unsuitable for most formal analysis. In another approach, an intermediate model notation is used to accommodate the integration of a wide variety of analysis tools—various modeling formalisms are translated to the intermediate notation that can in turn be mapped into suitable analysis tools. Nevertheless, with an intermediate notation valuable information from the source language, for example, model structure, may be lost in the translation to the intermediate notation and there is a risk for exponential growth in model size during translation. We believe other techniques are needed to satisfactorily solve this problem.

BACKGROUND AND RELATED WORK

There has been an immense amount of research done in formal modeling tools, mappings from modeling notations to analysis tools, and translation technology. The coverage in this section is by necessity cursory, but we will discuss our previous experiences and the core problem, and then present a short overview of efforts related to our research agenda.

The Problem

Our conjectures are based on extensive experience in this domain and they have a large impact on how languages and tools are adopted in industry (or not adopted as is often the case). In any larger project there will be a need to use several different modeling notations. For example, notations such as Simulink [7] or SCADE [5] are suitable for the control oriented aspect of a system design whereas Safe State Machines [5] or Statecharts [6] are more suited for the discrete aspect of a system. Other notations, for example, SCR [16] and RSML^{-e} [10] are considered better suited for high-level models used in the requirements domain. Thus, there must be the flexibility to select the right notation for the right task without having to "retool" the entire organization. In addition, after working on numerous projects in collaboration with industry, we have noticed that the adoption of new modeling tools is often hampered for nontechnical reasons. Typical obstacles to tools adoption are seen in comments such as "*We like what you are doing, but you do not have internal events like Esterel—we find events essential*", "*We like SCR, can you work with that?*", and "*We think the guard-conditions in Statecharts are messy, could you provide the nice tables we saw in RSML?*". These obstacles apply to both academic and commercial tools, and attempting to accommodate these wishes is typically economically infeasible—modifying tools and languages is simply too costly. Furthermore, as formal modeling techniques are slowly being adopted in industry, the need for powerful analysis tools becomes acute. As would be expected, no analysis tool is suitable for all tasks. Given the rapid change in verification and analysis technology, we will most likely see a steady stream of new exciting analysis tools that need to be incorporated into tools in the future.

To summarize, there is a critical need to accommodate numerous notations (and to modify these notations to the customers individual problems and taste) and a multitude of analysis tools. Because tool vendors and research groups cannot currently support these needs, promising tools and techniques are not adopted for superficial reasons and useful new analysis techniques are not adopted because of cost and technical difficulties. Therefore, we have come to the conclusion that we need a catalytic infrastructure for the design, development, and deployment of formal modeling tools that will serve two purposes; (1) it will allow the research community to evaluate and quickly deploy new ideas in a stable environment and (2) provide a blueprint for tool vendors for how to build tools that can be customized to meet the customers' needs. We hope this paper will serve as a catalyst and start of a community-wide initiative that will dramatically improve the penetration of formal modeling and recent analysis research results in industry.

From Models to Analysis Tools

The most common current solution to get from modeling tools to analysis tools is to develop separate translators from each modeling notation to each analysis tool. As the numbers of notations and analysis tools grow, this is an unsustainable solution since we may need $n \times m$ translators to map n modeling notations to m analysis tools. Providing such a collection of high-quality translators is not economically feasible.

To address this problem, an intermediate notation can be used as an interchange format between modeling notations and the analysis tools; for example, DC for synchronous languages [17], IF for the exchange of models between model checkers [18], and SAL for general purpose applications [19]. An intermediate language sitting between the modeling notations and the analysis tools reduces the number of translators needed to map n notations to m analysis tools to $n+m$ translators. There are, however, problems with this approach. For example, it is quite difficult to choose the *right level of abstraction for the intermediate language*, even when one knows all of the modeling-languages and target-languages. Consider choosing a low-level intermediate language, for example, Lustre, that does not have template types. If we have a modeling-language that has template types, such as SCADE, the translator must instantiate the template type for each of its uses. Since these template types can be immensely useful to model, for example, generic communications channels in a systems architecture, we may have dozens of occurrences to instantiate (possibly one for each connector in the architecture). This may be appropriate when translating to an analysis tool such as NuSMV which does not allow template types, but it may be wholly inappropriate when targeting an analysis tools such as PVS which does allow template types. Because of this information loss, the translation through an intermediate language to PVS can not take advantage of all of the capabilities of the target language thus making verification in PVS more difficult and time consuming than strictly needed. On the other hand, if the intermediate language has many high-level constructs, such as template types, the translations to the different target languages becomes much more difficult. Since we believe that new notations will constantly evolve and new analysis tools will become available in a constant stream (as an example, consider the dramatic evolution of SAT based model checkers the last few years), the selection of an appropriate intermediate notation will become impossibly difficult and we do not see this as a feasible long term solution.

Extensible Language Techniques

Our stated goal of allowing domain specific language features to be added to a modeling language has been studied in the area of programming languages and there are many tools and techniques that attempt to solve this problem. Besides adding new syntactic forms to a modeling language, we also require that these new constructs be able to specify some semantic analysis so that they can generate domain specific error messages, debugging behavior, as well as specify their direct translations to target languages when appropriate. Although these requirements are addressed by some of the many tools and techniques for language extensibility in the literature, no single approach addresses all of them. Traditional syntactic, hygienic [20], and programmable [21] macros systems and embedded domain specific languages [22] do allow new language constructs to be added to a language. However, they lack an effective way perform the necessary static analysis. On the other hand, meta-object protocol systems, *e.g.,* [23], provide limited opportunities to add new language constructs but can perform the static analyses needed to, for example, check for domain specific errors.

Attribute grammars [24] provide the foundation for what we believe will be a successful direction of inquiry. Language constructs are specified by productions and their explicit semantics can be defined by attribute definitions. The problem of modular language definition and extensibility has received much attention from the attribute grammar community, *e.g.,* [25, 26]. Some systems are guided by functional programming ideas and use, in essence, higher order functions as attributes in their quest for modular specifications, *e.g.,* [27]. Others are inspired by the object-oriented paradigm and employ inheritance to achieve a separation of concerns [28]. Of most use to us are higher-order attributes [29] that allow abstract syntax trees to be attribute values and reference attributes [30] that point to possibly remote nodes in the abstract syntax tree.

Most closely related to the extensible language framework described below is Microsoft's Intentional Programming system (IP) [31, 32, 33 Chapter 11]. This system allowed programmers to add domain specific features, called *intentions*, to their programming language in a style similar to attribute grammars. Although not as crisply defined in IP as in attribute grammars, IP did contain the essence of reference attributes and higher order attributes. The main innovative feature of IP was *forwarding*, a technique used to define new constructs in terms of host language constructs. In [34], we showed how forwarding can be used in attribute grammars to modularly specify languages and how the absence of forwarding hinders the modularity we seek and makes the addition of new language features more difficult. Our main criticism of IP [32] was its ad-hoc nature that prevented any static analysis of language extensions to test their compatibility.

A PROPOSED FRAMEWORK

In this section we will illustrate how attribute grammars extended with forwarding [34] can be use to define language extensions on top of a host-language. Note here that we do not categorically state that we believe this is the best solution to the problem outlined in the previous section, we simply outline what we see as a promising direction to start a community-wide dialogue with the goal to establish a tools architecture that will satisfy our need for flexibility and extensibility.

We hypothesize that an *extensible language* implemented using attribute grammars with forwarding [34] can serve as a *host-language* for (1) a plethora of domain specific *language-extensions* that can be combined to construct new *modeling-languages* suitable for different audiences and domains, (2) the translation of these extension constructs into their semantically equivalent representation in the host-language and the host-language translation into various target-languages, and (3) the direct translation of a language-extension construct to an analysis tool's language when the default translation provided through the host-language is inadequate.

In our framework, a language extension is specified as an attribute grammar *fragment* that contains new productions and attribute definitions for these productions and those in the host language. The activity of creating an extended language specification can be as simple as taking the union of all the productions and attribute definitions in the host language and language extension specifications. This is performed by the framework tools and provides for a significant degree of modularity between language extensions. In traditional attribute grammars the modularity and reuse of language features specified as attribute grammar fragments is achieved only by writing attribute definitions that "glue" new fragments into the host language attribute grammar. These attribute grammars, which do not have forwarding, cannot implicitly define semantics for a language construct and this is crucial for the modularity we seek.

To give the reader an introduction to attribute grammars and forwarding, we will first illustrate below the mechanism of forwarding in terms of a construct familiar from the programming language domain—a *foreach* loop. We will then illustrate how a well defined language suitable for the embedded systems domain—Lustre—can be used as a host-language and illustrate how extensions can be defined to start building up a modeling-language on top of this host-language. We provide some examples of using language extension to allow for flexibility in the modeling notation in following section. In the section on translation we show how forwarding is used to easily implement high-quality translations to many analysis engine languages while avoiding many pitfalls of intermediate languages.

Forwarding in Attribute Grammars

Forwarding is a *unifying technique that allows us to mimic common language extension processing techniques* like macro expansion, simple term rewriting, and meta-object protocols inside an attribute grammar framework and thus makes it possible to declaratively specify expressive language extensions. To use *forwarding*, a language construct specifies a *semantically equivalent* construct that defines the semantics not explicitly defined by the "forwarding" construct. In attribute grammar terms, a production defines a *distinguished* attributed abstract syntax tree that provides default values for synthesized attributes that are not explicitly defined by the production.

```
foreach:  for<Stmt> ::= elemType<Type> elem<Id> collection<Expr> body<Stmt>
   for.pp = "foreach" + elemType.pp + elem.lexeme + "in" + collection.pp + "do" + body.pp
   for.errors = if collection.type.implements(Collection)  then   no-error   else  mkError … for.pp …
     forwardsTo   parse  "{  `elemType.pp`  `elem` ;
                                  for ( Iterator `iter` = `collection`.iterator( ) ; `iter`.hasNext( ) ; )
                                       { `elem` = ( `elemType.pp`) `iter`.next( ) ;   `body` } "
                 where iter = generate_new_unique_Id ( )
```

Figure 1. The production specifying the *foreach* loop extension.

A familiar example from programming languages will clarify. Consider adding a simple *foreach* construct to Java to iterate over Java Collections. An attribute grammar production for doing so is shown in Figure 1. This puts syntactic sugar on a popular programming idiom but also defines its own simple error-checking semantic analysis. The *foreach* named production has a left-hand side *<Stmt>* non-terminal named *for* and right-hand side non-terminals *<Type>*, *<Id>*, *<Expr>*, and *<Stmt>* named as indicated. It explicitly defines the synthesized pretty-print *pp* and *errors* attributes allowing it to generate errors messages containing code written by the programmer. (We use the familiar dot (.) notation to access attribute values.) It also specifies its *forwards-to* tree as the expected host-language block-loop construct built by parsing the string and using the "unquote" operator (` `) to reference right-hand side subtrees, their attributes and the identifier *iter*. The reference attribute *type* points to a node in the program abstract syntax tree defining the collection's type. Its *pp* attribute is used to cast the Java *Object*-type value returned from the iterator method *next*. The left-hand side node generated by the production *foreach* is called the "forwarding-node." When it is queried for an attribute that it *does not explicitly define*, for example *jbc*, its Java byte code, it passes, or "forwards", that request to the "forwards-to" node, the block-loop construct, that returns the attribute's value. Thus, we re-use all attributes defined on the block-loop except those with explicit overriding definitions. Because the forwards-to tree will require inherited attributes, these are copied from the forwarding-node unless they are explicitly defined. Note that in some cases the forwards-to node may also forward the query; we will eventually find a value for the attribute since *all language extensions forward (directly or indirectly) to constructs in the host-language*. Forwarding is similar to macro expansion in that both reuse the semantics of existing language constructs, but unlike macros, forwarding productions also define semantics, as attributes, that here generate proper error messages. Also note that we have not specified how the concrete syntax of this extension is specified since this is done in a straight-forward and expected way. We thus limit our discussions to the more interesting issues in specifying the semantics of language extensions via attribute grammars.

Modeling-Languages as Extensions to a Host-Language

To illustrate how a new modeling-language can be specified as a set of language extensions to a host-language, we will use a simple example—the Altitude Switch (ASW). The ASW is a (somewhat hypothetical) avionics system that turns power on to another system when the aircraft descends below a threshold altitude and turns it off when the aircraft ascends above the threshold altitude plus a hysteresis factor. As an example, we focus on one of the state variables that models the ASW behavior—the *AltStatus* variable used to track whether the aircraft should be considered above or below the threshold.

A Lustre-like specification of *AltStatus* is shown in Figure 2. In it, the initial value of *AltStatus* is undefined (indicated by the '*Unknown →' construct*) and thereafter the variable is assigned by the nested if-expression. We assign *AltStatus* the value *Above* if the altitude readings are reliable (*AltQuality = Good*) and we are either (1) classifying *AltStatus* for the first time (the previous *AltStatus* was *Unknown*) and we are above the threshold or (2) *AltStatus* has been established and we are above the threshold plus the hysteresis. We declare *AltStatus* to be *Below* if we have reliable altitude readings and the altitude is less than or equal to the threshold. If the altitude readings are not reliable *AltStatus* is *Unknown*.

```
type Status = enum { Unknown, Above, Below } ;
node ASW (AltQuality:Q, AltThres:int, Hyst:int)
           returns (AltStatus:Status);
  let AltStatus = Unknown ->
     if AltQuality = Good and Altitude > AltThres and
        (PREV(AltStatus) = Unknown or Altitude > AltThres + Hyst)
     then Above
     else if AltQuality = Good and (not Altitude > AltThres)
          then Below
          else if not AltQuality = Good then Unknown else pre(AltStatus) ;
  tel
```

Figure 2. A Lustre-like definition of the state variable *AltStatus*.

Lustre as a host-language:

We hypothesize that Lustre [35] may be a suitable host-language in our domain of interest for two reasons. First, Lustre is expressive enough to capture a large class of interesting behaviors and it has a well defined and simple semantics making it suitable for formal treatment by various tools [36]. Second, Lustre is supported by commercial tools [5], for example, a code generator that has been qualified for use in safety critical applications, thus, making it of interest to industrial partners. Lustre may be implemented as an extensible host language by writing an attribute grammar that defines its language constructs and defines attributes that perform semantic analysis and translation. We may define an *errors* attribute similar to the one in the *foreach* example. It may also involve defining string-type translation attributes named *smv_trans*, *pvs_trans*, and others that specify how constructs are translated to various target languages.

Below we illustrate how a Lustre host language can be extended with modeling language features that may be useful in different domains or preferred by different user communities. We describe how RSML^{-e} state variables and events can be added as language extensions.

Implementing RSML^{-e} as a collection of language extensions

Since the Lustre host-language may not be the most suitable for review by domain experts (for example, pilots or air traffic controllers) an alternate notation would be of interest. The modeling notations SCR and RSML^{-e} are such notations that have been well-received and shown to be relatively easy to understand and use [16, 37, 38]. Figure 3 shows a fragment of an RSML^{-e} specification of the ASW. The figure shows the definition of the state variable *AltStatus* discussed above. The conditions under which the state variable changes value are defined in the *Equals* clauses in the definition. The tables are adopted from the original RSML notation [37]—each column of truth values represents a conjunction of the propositions in the leftmost column (*F* represents the negation of the proposition and a "*" represents a "don't care" condition). In a table with several columns we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form.

```
STATE_VARIABLE AltStatus :
  VALUES : { Unknown, Above, Below }
  INITIAL_VALUE : Unknown

  EQUALS Above IF
   TABLE
    PREV(AltStatus) = Unknown  : T * ;
    AltQuality = Good          : T T ;
    Altitude > AltThres        : T T ;
    Altitude > AltThres + Hyst : * T ;
   END TABLE

  EQUALS Below IF
  TABLE
   AltQuality = Good          : T ;
   Altitude > AltThres        : F ;
  END TABLE

  EQUALS Unknown IF
   TABLE
    AltQuality = Good : F ;
   END TABLE
END STATE_VARIABLE
```

Figure 3. RSML^{-e} like definition of the State Variable *AltStatus*.

As we stated above, a language extension is an attribute grammar *fragment* that specifies productions that define the abstract syntax of any new language constructs, attribute definitions for these new productions, and attribute definitions for existing productions in the host-language. To implement the *State_Variable* construct in Figure 3 we follow this pattern and define a set of productions to define the syntax of state variable constructs as well as attributes on these constructs that check for errors and assist in building the semantically equivalent Lustre language construct that the state variable will forward to.

The *State_Variable* construct is essentially a variable assignment statement like those in the host-language with the exception that it also serves as a declaration of the assigned variable. The production defining a state variable has on its right hand side an identifier *name* (in this case *AltStatus*), a non-terminal for the possible values, its initial value *init<Expr>*, and an expression *expr<Expr>* for subsequent values. The implicit semantics of a state variable are provided by the Lustre assignment statement that it forwards to and is semantically equivalent to, essentially, *name = init → expr*. Since a *State_Variable* production also declares a variable, this production builds the declaration as a Lustre declaration, of the form *name : type,* and indicates that this declaration should be *lifted* to the enclosing *node*. This is done by placing the declaration, implemented as a higher-order attribute in a synthesized attribute that collects such declarations and moves them up the tree to the point that they can be inserted into the enclosing Lustre node construct. We do not provide the attribute grammar specification here. The specification for constructing the disjunctive normal form expression that becomes the condition of the if-then-else that the table translates to is more verbose than it is difficult.

Events as an extension

In Figure 4 is part of the same altitude switch, this time specified in the host-language extended with a notion of events. Here, the occurrence of the externally defined event *AltRecvEvt* is used in the computation of *AltStatusEvt*. These computations also activate (or throw) the *AltClassifiedEvt* and *AltLostEvt* events to indicate the outcome of the assignment of the *AltStatusEvt* value. We have abbreviated the three expressions that appear in the original Lustre ASW in Figure 2 as C1, C2, and C3 here to simplify the presentation.

```
var AltClassifiedEvt : Event ;
    AltLostEvt : Event ;
AltStatus = Unknown ->
 if catch AltRecvEvt and C1
 then throw AltClassifiedEvt return Above
 else if catch AltRecvEvt and C2
      then throw AltClassifiedEvt return Below
      else if catch AltRecvEvt and C3
           then throw AltLostEvt return Unknown
           else pre(AltStatus)
```

Figure 4. Event/Action style constructs added to the host-language.

Events can also be implemented as a language extension. The attribute grammar specification for events defines three primary productions—one to generate or "throw" an event, another that evaluates to *true* if it "catches" the specified event, and an event declaration. The specifications for these productions is straight forward, but they do require a more complex set of supporting attributes to generate the boolean variable declarations and their defining expressions that form their Lustre implementation that these extensions forward to. Declarations of events need to be transformed into declarations of boolean variables. These variables replace events and are true under the condition in which the original event would have been thrown and false otherwise. Thus, the *catch* production simply forwards to a reference to the boolean variable emulating the event since both are true under same conditions. The *throw* production simply forwards to the expression that it "returns" since when events are emulated the boolean event variable and its defining expression effectively replace the throw constructs.

Of particular interest is how the assignment statements for the event-emulating boolean variables are generated. In the example, the boolean variable *AltClassifiedEvt* is generated to replace the *AltClassifiedEvt* event. The assignment statement below (with catch constructs yet to be removed) is also generated to set this boolean variable to *true* under the conditions that the event was thrown.

```
AltClassifiedEvt = False -> (catch AltRecvEvt and C1) or
  (not (catch AltRecvEvt and C1) and (catch AltRecvEvt and C2))
```

The *AltClassifiedEvt* boolean variable is naively computed to be *true* if the first if-condition (in Figure 4) is *true* or if the first if-condition is *false* and the second if-condition is *true*. For an event *e*, it is a relatively straight forward process to compute this expression. For each *throw e* construct, we build an expression from the enclosing if-condition's that must be *true* to cause that throw instance to "fire." Again, we leave out the implementation of this since a *inherited* higher order attribute can be specified to pass down the enclosing if condition <*Expr*>-trees to a throw statement. It constructs an expression that is true when the event is thrown. Such expressions are collected from each throw, via a synthesized attribute, and the expression that is the disjunction of them is created to become the expression in the defining Boolean assignment as seen in the example above.

Given the mechanisms discussed above, it is easy to combine events and state-machine transitions into a "transitions with events" notation similar to the original definition of RSML [37]. This flexibility in introducing new language constructs that can be targeted for various stake-holder needs is highly desirable. In addition, the attribute grammar framework provides an opportunity for static analysis at the appropriate level of abstraction, that is, in the modeling-language rather than in the host-language. For example, in all of these extensions, we would want to define the basic error checking attribute *errors* on the productions defining the new language constructs. If we do not define the error checking attributes any type errors will be reported by the *errors* attribute on the forwarded-to construct. Thus, all errors will be discovered but the generated error message will be in terms of the forwarded to expression, not the transition expression that was written by the user. This demonstrates one aspect of how attribute grammars with forwarding provide a convenient yet powerful mechanism for specifying language extensions. By specifying additional attributes, in this case *errors*, our language extensions behave more and more like first-class constructs in the language and provide all of the capabilities that one expects from “built-in” language features.

Translation

Translators, either for execution, debugging, or analysis purposes, can also be defined as a language extension consisting of a set of attribute definitions for the host-language and some language extension productions. For a translation-based evaluator, we would specify definitions for an *eval_trans* attribute on the host-language productions that specify a translation into, for example, C that can then be compiled and executed. This model is also used for translating specifications to various analysis tools such as the NuSMV model checker and the PVS theorem prover. To specify these translations, we would specify two string-valued attributes, *pvs_trans* and *smv_trans*, that on a language construct define its translation into, respectively, PVS and NuSMV. Definitions for these attributes would be required for all productions in the host-language. Thus any language extension, such as events, will have a default translation into NuSMV and PVS. For example, the PVS translation for a *throw* expression is determined by retrieving the *pvs_trans* attribute from the semantically equivalent construct that it forwards to.

For other language extension constructs, this translation through the host-language may lose information (as described in background section) and result in a degraded representation in the target language that makes the analysis more expensive. Such constructs should instead provide a definition for the *pvs_trans* or *smv_trans* attributes to specify their direct non-lossy translation. Consider adding template types as a language extension. Template types exist in PVS but not in NuSMV or our host-language. We want our translation to PVS to maintain this information and thus the productions that specify the template type extension would provide explicit definitions for *pvs_trans* that specifies the PVS-template code that implements the language extension template types. This ensures a high-quality translation to PVS. Since the host-language does not support template types, the extension must specify, via a collection of attribute definitions, how each node declaration with template type parameters can be *instantiated* into several instances that do not contain template types. Each generated instance is the result of the concrete types used in a specific node call construct. This is not a trivial analysis or transformation, but it can be done with forwarding attribute grammars and follows the techniques for instantiating C++ templates. These transformations create the host-language constructs that the template types forward to in the host-language. The translation of template types to NuSMV then relies on forwarding and the definition of *smv_trans* on the host-language productions. This approach avoids the pitfalls of intermediate notations described above in the background section.

To summarize, as we have outlined in this section, we believe attribute grammars with forwarding can be used to capture the semantics of a host-language and define language-extensions that can be combined to create new modeling-languages, as well as define how these languages are translated to various target-languages. We believe that this formalism is highly suitable as a foundation for model-based tools and that it will help us provide the high-level of flexibility, ease of change, and high quality for which we aim. Do note again, however, that we are presenting this idea to generate a dialogue in the formal methods, model-based, and static analysis communities with the goal of evolving a consensus for the architecture of the next generation of hyper-flexible modeling and analysis tools.

A CALL TO ARMS

In this report we assert that a critical obstacle to the widespread adoption of modeling and analysis tools lies in the lack of flexibility to accommodate customer preferences when considering the adoption of tools. We believe that the following conjectures describe fundamental obstacles to wide adoption of formal modeling and the potential for automation that comes with it; (1) no single modeling notation will suit all, or even most, modeling needs, (2) no analysis tool will fit all, or even most, analysis tasks, and (3) flexible and stable tools must be made available for realistic evaluations and technology transfer.

To make automated software engineering techniques more useful for more types of developers and allow us to move forward as a community it is crucial that we develop a foundation for building extensible and flexible modeling language processing tools. Such tools must satisfy, at a minimum, the following requirements—successful modeling tools must (1) allow easy extension and modification of formal modeling notations to meet the domain specific needs of a class of users, (2) allow easy construction of high-quality translations from the modeling notations to a wide variety of evolving analysis tools, and (3) enable controlled reuse of tool infrastructure to make tools extensions cost effective.

To initiate the discussions in the community, we hypothesize that languages and tools built using higher-order attribute grammars with forwarding can serve as a basis for such flexible language processing tools; tools that will allow us to unify our efforts and help bring our collective work to a broader audience. Naturally, although we like to think so, we do not presume that our direction is the best (or even practicable) and it serves only as an illustration as to what we would like to achieve in a unifying language processing framework. The aim of this report is to stimulate the discussion and, hopefully, move the community into action so we can join forces in developing the solid tools infrastructure needed to have an impact on software development practice.

REFERENCES

1. M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

2. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

3. S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proc. of Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, 1999.

4. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, Nov. 1999.

5. T. Esterel. Corporate web page. www.esterel-technologies.com, 2004.

6. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Enginee*ring, 16(4):403–414, April 1990.

7. MathWorks. The MathWorks Inc. web page. http://www.mathworks.com, 2004.

8. N. G. Leveson, M. P. Heimdahl, and J. D. Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 127–145, September 1999.

9. C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

10. J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 163–179, September 1999.

11. J. M. Thompson, M. W. Whalen, and M. P. Heimdahl. Requirements capture and evaluation in NIMBUS: The light-control case study. *Journal of Universal Computer Science*, 6(7):731–757, July 2000.

12. S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of the International Symposium of Formal Methods Europe (FME 2003)*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, September 2003. Springer.

13. A. Joshi, S. P. Miller, and M. P. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *Proceeding of the 22nd Digital Avionics Systems Conference*, October 2003.

14. S. Rayadurgam, A. Joshi, and M. P. E. Heimdahl. Using PVS to prove properties of systems modelled in a synchronous dataflow language. In *Proceedigns of the 5th International Conference on Formal Engineering Methods, ICFEM 2003*, pages 167–186, Singapore, November 2003.

15. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1998.

16. K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, 6(1):2–13, January 1980.

17. A. Poigne and L. Holenderski. On the combination of synchronous languages. In W. de Roever, H. Langmaack, and A. Pnueli, editors*, Proceedings of International Symposium on Compositionality*, volume 1536 of *Lecture Notes in Computer Science*, Bad Malente, Germany, September 1997. Springer-Verlag.

18. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, L. Mounier, and J. Sifakis. IF: An intermediate representation for sdl and its applications. In *Proceedigns of the SDL-Forum'99*, page 423–440. Elsevier Science, 1999.

19. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Ruess, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, Proc. of LFM 2000: *Fifth NASA Langley Formal Methods Workshop*, pages 187–196. NASA Langley Research Center, June 2000.

20. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. *In Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.

21. D. Weise and R. Crew. Programmable syntax macros*. ACM SIGPLAN Notices*, 28(6), 1993.

22. P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es), 1996.

23. S. Chiba. A metaobject protocol for C++. *In Proc. of Object-oriented programming systems, languages, and applications*, pages 285–299. ACM Press, 1995.

24. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127--145, 1968. Corrections in 5(2):95–96, 1971.

25. H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Science of Computer Programming*, 3(3):223–278, 1983.

26. U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

27. D. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *The Computer Journal*, 33(2):164–172, 1990.

28. G. Hedin. An object-oriented notation for attribute grammars. In *Proceedings of the European Conference on Object-Oriented Programming, ECOOP'89*, 1989.

29. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *Conference on Programming Languages Design and Implementation*, pages 131–145, 1990. Published as ACM SIGPLAN Notices, 24(7).

30. G. Hedin. Reference attribute grammars. In *2nd International Workshop on Attribute Grammars and their Applications*, 1999.

31. C. Simonyi. The future is intentional. IEEE *Computer*, May 1999.

32. E. Van Wyk, O. de Moor, G. Sittampalam, I. Sanabria-Piretti, K. Backhouse, and P. Kwiatkowski. Intentional Programming: a host of language features. Technical Report PRG-RR-01-21, Computing Laboratory, University of Oxford, 2001.

33. K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

34. E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of Conf. on Compiler Construction,* volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002.

35. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Proc. of IEEE*, 79(9):1305–1320, September 1991.

36. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language Lustre*. IEEE Transactions on Software Engineering*, pages 785–793, 1992.

37. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

38. M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In Proc*. 24th Conf. on Software engineering*, pages 33–43, Orlando, Florida, May 2002. ACM Press.

SATISFIABILITY SOLVING FOR SOFTWARE VERIFICATION

David Déharbe[*]
DIMAp/UFRN (Natal, Brazil)
Silvio Ranise
INRIA-Lorraine (Nancy, France)

ABSTRACT

Many approaches to software verification require to check the satisfiability of (possibly quantified) first-order formulae in theories modeling user-defined data types, the memory model used by the programming language, and so on. For such verification techniques, it is of crucial importance to have satisfiability solvers which are both predictable and flexible, i.e. capable of automatically discharging the largest possible number of proof obligations coming from the widest range of verification problems. In this paper, we describe our approach to build predictable and flexible satisfiability solvers by combining (an extension of) resolution theorem proving, arithmetic reasoning, Boolean solving, and some transformations on the proof obligations (such as definition unfolding or theory reduction). We show the viability of the approach by describing the experimental results obtained with an implementation of the proposed techniques on a set of proof obligations extracted from various software verification problems, in particular the certification of auto-generated aerospace code.

CONTEXT AND MOTIVATION

Many approaches to software verification, ranging from applications of Hoare logic to software model checking (see e.g., [1]) and, more recently, to program analysis,[1] require to discharge some proof obligations, i.e. checking that some formula (usually of first-order logic with equality) is satisfiable in a given theory modeling the user-defined data types of the software system under scrutiny, the memory model used by the programming language, its type system, and so on. For such verification techniques, it is of crucial importance to have *satisfiability solvers* which are both *predictable* and *flexible*, i.e. capable of automatically discharging the largest possible number of proof obligations coming from the widest range of verification problems. Indeed, this task is far from simple.

We identify two key challenges to build predictable and flexible satisfiability solvers:

- to provide satisfiability procedures for the theories most commonly used in software verification that capture interesting classes of software properties so to have the highest possible degree of predictability (i.e. the guarantee of termination with a correct answer), and
- to widen the scope of applicability of the available satisfiability procedures so to be able to discharge proof obligations which are usually expressed in extensions of the supported (decidable) theories (see [2] for an extensive discussion on this issue).

In this paper, we describe our approach to build satisfiability solvers for software verification by tackling the above challenges. A careful integration of the following three techniques is at the core of our approach:

- Superposition theorem proving (see e.g., [3]) and satisfiability checking for Linear Arithmetics to reason about the data flow of software systems;
- Boolean solving to handle their control flow;
- Two heuristics to reduce—*a priori*—the search space of the satisfiability checking problem.

[*]The fi rst author has received fi nancial support of *Centro Nacional de Desenvolvimento Científico e Tecnológico* (CNPq), a Brazilian federal government agency for scientifi c and technological development.

[1]`http://www.mit.edu/~vkuncak/projects/jahob`

```
function Bool + $T$($\phi$: quantifier-free formula)
1     $\phi^p \longleftarrow fol2prop(\phi)$;
2     $A^p \longleftarrow fol2prop(Atm(\phi))$;
3     while Bool-$satisfiable(\phi^p)$ do
4         $\beta^p \longleftarrow pick\_total\_assign(\phi^p)$;
5         $(\rho, \pi) \longleftarrow T\text{-}satisfiable(prop2fol(\beta^p))$;
6         if ($\rho ==$ sat) then return sat;
7         $\phi^p \longleftarrow \phi^p \wedge \neg fol2prop(\pi)$;
8     end while;
9     return unsat;
end
```

Figure 1: A simplified view of traditional SMT algorithms

SATISFIABILITY OF QUANTIFIER-FREE FORMULAE

Let T be a theory with decidable satisfiability problem, i.e. it is decidable to check whether an arbitrary conjunction of literals of T is satisfiable. The *Satisfiability Module Theory (SMT) problem* is the problem of checking if a quantifier-free first-order formula ϕ of T is satisfiable, i.e. whether $T \wedge \phi$ is satisfiable. An *SMT algorithm* is a decision procedure for the SMT problem.

Satisfiability Modulo Theory checking

In the following, β^p is used to denote a propositional assignment; π is used to denote a conjunction of ground first order literals, and π^p its boolean abstraction; in general, we use the superscript "$...^p$" to denote boolean formulas or assignments. We represent propositional assignments β^p indifferently as sets of propositional literals $\{l_i\}_i$ or as conjunctions of propositional literals $\bigwedge_i l_i$; in both cases the intended meaning is that a positive literal v (resp. a negative literal $\neg v$) denotes that the variable v is assigned to true (resp. false).

Figure 1 depicts Bool$+T$, the SMT algorithm underlying (with different variants) several systems such as Mathsat [4], DPLL(T) [5], TSAT [6], ICS [7], CVC-Lite [8], and **haRVey**[2]. For the sake of simplicity, we only give a naive formulation, based on the enumeration of total assignments and we ignore more realistic representations based on DPLL-style assignment enumeration (for this the interested reader is pointed to one of the papers above). The algorithm relies on the existence of a bijection between the atoms of ϕ and a suitable set of propositional letters. In particular, we call Atm the set of ground atoms in ϕ and P_{Atm} be a set of propositional letters s.t. the cardinality of Atm is equal to that of P_{Atm} and $Atm \cap P_{Atm} = \emptyset$. Let *atm2prop* be a bijective function from Atm to P_{Atm}. *fol2prop* is a mapping from the quantifier-free formula ϕ to a propositional formula ϕ^p as the homomorphic extension of *atm2prop* to ϕ; *prop2fol* is the inverse of *fol2prop*; its result is a conjunction of ground first-order literals. Basically, the truth assignments for (the propositional abstraction of) ϕ are enumerated, and checked against T. The procedure either concludes satisfiability if one such model is found, and returns with failure otherwise. The ancillary functions in Figure 1 are assumed to satisfy the following requirements:

- $pick\_total\_assign$ returns a total assignment to the propositional variables in ϕ^p. So, β^p is a conjunction of propositional literals;
- T-*satisfiable*(β) detects if β is satisfiable in T. If so, it returns (sat, $\emptyset$); otherwise, it returns (unsat, π), where π is satisfiable in T and $\pi \subseteq \beta$ is called a *theory conflict set*.

It is easy to show that the algorithm terminates, and it is correct and complete (see e.g., [9] for more details). Termination holds since there exists only a finite number of possible assignments, and none is considered more than once: the set of propositional assignments to ϕ^p is monotonically decreasing, since each iteration excludes at least one of them. Correctness follows from the basic definitions of truth assignment and interpretation: sat is returned only if β^p propositionally satisfies ϕ, and there exists an interpretation refining it. Completeness follows from the fact that we never eliminate a consistent assignment.

[2]See `http://combination.cs.uiowa.edu/smtlib` for more details and pointers to these systems

Superposition-based satisfiability procedures

To be able to execute the algorithm in Figure 1, we are left with the problem of building the function $T\text{-}satisfiable$. In order to do this, we adopt the rewriting approach described in [10] which uses superposition as a framework to synthesize satisfiability procedures for various theories of interests for verification.

The superposition calculus (see [3] for an overview) checks the satisfiability of arbitrary sets of first-order clauses. It consists of a set of rules which are derived from resolution and which are especially designed for the efficient treatment of equality. Although equality dramatically enlarges the search space of resolution based provers, the effectiveness of superposition lies in some powerful criteria (such as term ordering) to prune the search space while maintaining completeness. Any fair application of the rules of the calculus to an unsatisfiable set of clauses derives $\bot$ (also called the empty clause).[3] Roughly, a saturation prover (such as the E prover [12]) amounts to a clever mechanization of the exhaustive application of the rules of the calculus to any finite set of first-order clauses (indeed, a lot of sophistication is required to obtain efficient implementations). In general, the process of applying the rules of the calculus to a set of clauses may not terminate since first-order logic is undecidable. If for a class $\mathcal{C}$ of clauses, we are able to prove that this process terminates, then we are entitled to conclude that the calculus is a satisfiability procedure for $\mathcal{C}$ given its refutation completeness. The methodology to build satisfiability procedures described in [10] is based on this simple observation and it is organized in two phases.

Let $Ax(\mathcal{T})$ be a finite set of equational clauses axiomatizing $\mathcal{T}$ and β a conjunction of ground literals.[4] The first phase amounts to flattening all ground literals in β. A flat literal is either an equality of the form $f(c_1, ..., c_n) = c_{n+1}$ or the negation of an equality of the form $c_1 \neq c_2$ (where $c_1, ..., c_{n+1}$ are constants and f is an n-ary function symbol).[5] Flattening is done by extending the signature with "fresh" constant symbols for all the distinct non-constant sub-terms in β. It is easy to see that flattening preserves satisfiability. Let β' be the result of flattening β. The second phase consists of exhaustively applying the rules of the superposition calculus to the clauses in $Ax(\mathcal{T}) \wedge \beta'$. As shown in [10], the second phase terminates for many interesting theories such as the theory of lists, arrays, sets, and their combination. We illustrate the approach on a simple example. Example Let us consider the theory $\mathcal{A}$ of arrays. The signature $\Sigma_{\mathcal{A}}$ contains the binary function symbol read, the ternary function symbol write (abbreviated below with rd and wr, respectively), and a finite set of constant symbols (written in small letters). The theory $\mathcal{A}$ is axiomatized by the following set $Ax(\mathcal{A})$ of axioms:

$$\forall A, I, E.(\mathsf{rd}(\mathsf{wr}(A, I, E), I) = E) \tag{1}$$

$$\forall A, I, J, E.(I \neq J \Rightarrow \mathsf{rd}(\mathsf{wr}(A, I, E), J) = \mathsf{rd}(A, J)), \tag{2}$$

where capitalised letters are implicitly universally quantified variables. Let β be the following conjunction of literals:

$$a = \mathsf{wr}(\mathsf{wr}(a, i, \mathsf{rd}(a, j)), j, \mathsf{rd}(a, i)) \wedge \mathsf{rd}(a, i) \neq \mathsf{rd}(a, j).$$

Flattening β yields the conjunction of the following (ground) literals:

$$\begin{aligned} c_1 &= \mathsf{rd}(a, i) && (3)\\ c_2 &= \mathsf{rd}(a, j) && (4)\\ c_3 &= \mathsf{wr}(a, i, c_2) && (5)\\ c_4 &= \mathsf{wr}(c_3, j, c_1) && (6)\\ a &= c_4 && (7)\\ c_1 &\neq c_2 && (8) \end{aligned}$$

where c_1, c_2, c_3, c_4 are "fresh" constants. The second phase amounts to exhaustively apply the rules of the superposition calculus given in [3] to $Ax(\mathcal{A}) \wedge \beta'$. For this example, standard rewriting and the following simplified version of the calculus is sufficient:

$$\frac{l' = r' \quad l = r}{(l[r'] = r)\sigma}\ \textit{Superposition}, \qquad \frac{l' = r' \quad l \neq r}{(l[r'] \neq r)\sigma}\ \textit{Paramodulation}, \qquad \frac{s \neq t}{\bot}\ \textit{Reflection},$$

[3]Fairness means that if some inference is possible, it will be performed at some step unless one of the parent clauses gets simplified or deleted (see, e.g. [11] for a formal definition).

[4]Here, we are assuming that the theory $\mathcal{T}$ can be finitely presented by a set of first-order clauses. As shown in [10], many theories of practical interest admits such an axiomatization.

[5]To check satisfiability, predicate applications can be turned into function application as follows. For p an n-ary predicate symbol is in the input set of literals, the literal $p(t_1, ..., t_n)$ is translated to $f_p(t_1, ..., t_n) = \mathtt{tt}$ where f_p is a "fresh" n-ary function symbol and $\mathtt{tt}$ is a distinct constant. Similarly, $\neg p(t_1, ..., t_n)$ is translated to $f_p(t_1, ..., t_n) \neq \mathtt{tt}$. So, via this satisfiability preserving transformation, flattening can be applied also to non-equational literals.

where σ is the most general unifier between l' and a sub-term of l, for *Superposition* and *Paramodulation*, and the terms s and t must be unifiable, for *Reflection* (we have omitted the provisos about ordering constraints for simplicity).

A *Superposition* between (6) and axiom (1) yields $c_1 = \mathsf{rd}(c_4, j)$, which can be rewritten to $c_1 = \mathsf{rd}(a, j)$ by (7). A *Superposition* between $c_1 = \mathsf{rd}(a, j)$ and (4) gives $c_1 = c_2$, which by *Paramodulation* with (8) yields $c_1 \neq c_1$. By applying *Reflection* to $c_1 \neq c_1$, we immediately obtain $\bot$, which proves that β is $\mathcal{A}$-unsatisfiable.

In order to implement T-*satisfiable*(β), it is sufficient to implement flattening (which can be easily done), and then run a superposition theorem prover on $Ax(T) \wedge \beta'$. Furthermore, all superposition theorem provers are capable of returning a proof of the unsatisfiability of a set of clauses. By analyzing such a proof, it is possible to identify the sub-set of (unit) clauses in β' which are used to derive the empty clause. This offers an elegant and uniform way to obtain the theory conflict set π. Since the proof found is not necessarily the one using fewest assumptions, the returned set of literals is not guaranteed to be minimal but it is usually a good over-approximation.

It is important to emphasize the *flexibility* offered by this approach with respect to specialized decision procedures. In fact, when a decision procedure for a new theory is needed, it is only necessary to feed the superposition theorem prover with the axioms of the theory and the literals to be proved (un-)satisfiable, whereas with specialized decision procedures a major design and coding effort should be undertook.

The Nelson-Oppen combination method

The price to pay for using superposition to build satisfiability procedures in a flexible and uniform way is a restriction in the class of theories which can be handled. In fact, it is well-known that superposition theorem proving has difficulties in handling (decidable fragments of) arithmetics. The technique described above inherits this limitation so that it can efficiently handle (equational) theories which are axiomatized by a finite set of clauses. In order to overcome this limitation, we can use the Nelson and Oppen (N&O) combination schema [13] to combine saturation theorem proving and arithmetic reasoning (for details and preliminary experiments, see [14]).

The N&O combination method enables us to solve the problem of checking the satisfiability of a conjunction Φ of quantifier-free literals in the union of two signature-disjoint theories T_1 and T_2 for which two satisfiability procedures are available. Since the literals in Φ may be built over symbols in T_1 or in T_2, we need to *purify* them by introducing fresh constants to name sub-terms. This process leaves us with a conjunction $\Phi_1 \wedge \Phi_2$ which is equisatisfiable to Φ where Φ_i contains only literals with symbols of T_i, for $i = 1, 2$.[6] In this way, literals in Φ_i can be *dispatched* to the available decision procedure for T_i.

To show the correctness of the N&O method (see e.g., [15]), the theories T_1 and T_2 must be stably-infinite. Roughly, a theory is *stably infinite* if any satisfiable quantifier-free formula is satisfiable in a model having an infinite cardinality. All theories considered in this paper (the theory of equality, the theory of arrays, and the theory of Linear Arithmetic) are stably infinite.

An efficient implementation of the N&O method is based on the availability of satisfiability procedures with (at least) the following properties (see [16] for an in depth discussion on this and other efficiency issues):[7]

Deduction completeness. It must be capable of efficiently detecting elementary clauses (i.e. a clause whose literals are equalities between constants which occur in purified literals belonging to both theories) which are implied by the input conjunction of literals.

The N&O method for satisfiability procedures satisfying the requirements above is depicted in Figure 2 when T_1 is the theory of equality for which the superposition calculus is known to be a satisfiability procedure (see e.g., [10]) and T_2 is Linear Arithmetic (LA) for which various satisfiability procedures are available (see e.g., [18]). Such a combination method simply consists of exchanging elementary clauses between the two procedures until either unsatisfiability is reported for one of the two component theories or no more elementary clauses can be exchanged. In the first case, we report the unsatisfiability of the input formula; in the second case, we report its satisfiability. Only finitely many elementary clauses can be constructed by using the constants of both Φ_1 and Φ_2, and so the N&O method terminates.

[6]Notice that flat literals are purified.

[7]Incrementality and resettability are two other common requirements in this context. Unfortunately, it is difficult to modify modern state-of-the-art superposition provers so to make them incremental and resettable. So, we do not discuss here these issues. However, see [17] for a possible way to overcome this problem.

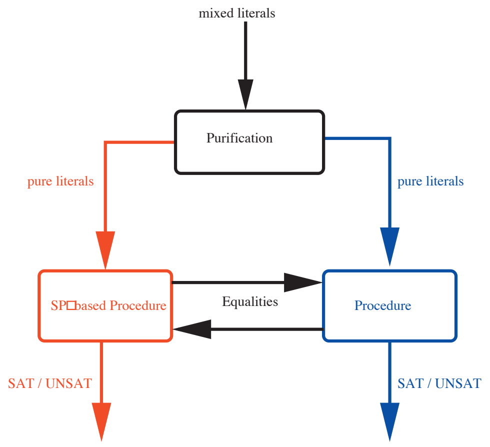


Figure 2: The Nelson-Oppen Combination Method

It is sufficient to exchange only elementary equalities when combining convex theories. A theory is *convex* if for any conjunction of equalities Γ, a disjunction D of equalities is entailed by Γ if and only if some disjunct of D is entailed by Γ. Examples of convex theories are the theory of equality, the theory of lists, and the theory of Linear Arithmetic over the Rationals ($LA(\mathcal{R})$). Since both procedures are assumed to be deduction complete, the combination method only needs to pass around elementary equalities between the procedures as soon as they discover them.

When combining at least one non-convex theory such as the theory of arrays or the theory of Linear Arithmetic over the Integers ($LA(\mathcal{I})$), the combination method is more complex since the procedures should exchange elementary clauses. Although the procedures are capable of deriving the entailed elementary clauses, their processing is problematic since they are only capable of handling conjunctions of literals. The standard solution is to case-split on the derived elementary clauses and then consider each disjunct in turn by using a backtracking procedure.

Although surprising (since superposition is not known to be complete for consequence finding, i.e. we are not guaranteed that a clause which is a logical consequence of a set of clauses will be eventually derived by applying the rules of the calculus), satisfiability procedures obtained by superposition are deduction complete, i.e. derive sufficiently many elementary clauses to be efficiently combined *à la* N&O with other procedures (see [17] for details). We illustrate the combination schema with a simple example. Example Let us consider the theory T obtained as the union of LA($\mathcal{R}$) and the theory $\mathcal{A}$ of arrays (cf. Example). A deduction complete satisfiability procedure for $\mathcal{A}$ can be built by superposition [17]. A deduction complete satisfiability procedure for LA($\mathcal{R}$) can be built by modifying the Fourier-Motzkin elimination method (see e.g., [18] for details). Now, consider the problem of checking the satisfiability (in T) of the following conjunction of literals (which are already purified):

$$\mathsf{rd}(b,i) = e_2 \land \mathsf{wr}(a,i,e_1) = b \land \tag{9}$$
$$e_1 + e_2 \neq 2e_2 \tag{10}$$

(notice that e_1 and e_2 occur both in literals of LA($\mathcal{R}$), cf. (10), and of $\mathcal{A}$, cf. (9). First, (9) is sent to the superposition prover which terminates without deriving the empty clause. However, it is capable of performing the following derivation (again we consider the simplified superposition calculus introduced in Example):

$\mathsf{rd}(b,i) = e_1$ by *Superposition* between (1) and the second literal of (9)

$e_1 = e_2$ by rewriting the previous with the first literal of (9)

Then, the derived elementary (unit) clause $e_1 = e_2$ is sent with (10) to the satisfiability procedure for LA($\mathcal{R}$) which immediately reports unsatisfiability. The N&O combination schema reports the unsatisfiability of the conjunction of (9) and (10).

SATISFIABILITY OF ARBITRARY FIRST-ORDER FORMULAE

We now consider two extensions of the SMT problem, introduced at the beginning of the previous section. First, we consider a theory T whose satisfiability problem is not necessarily known to be decidable. This is quite common in software verification where theories are frequently obtained by modularly composing and extending small theories. To be concrete, let T be obtained by extending LA with finitely axiomatizable theories (which themselves may be obtained as combinations/extensions of others) whose signatures are not necessarily disjoint with that of LA. We describe a semi-decision procedure for the satisfiability problem of T based on an extension of the N&O method, where the superposition prover generates instances of the axioms of the theories and then sends such facts (hence, a superset of the elementary clauses) to the satisfiability procedure for LA. To make this approach practical, we briefly present two heuristics which we have found particularly useful to reduce the search space of the superposition prover: definition unfolding and reduction of large theories [19]. The second extension of the SMT problem we consider here is checking the satisfiability of first-order formulae containing quantifiers. We describe a technique which allows us to reduce the satisfiability problem of quantified formulae in a theory T to the satisfiability problem of quantifier-free formulae in an extension of the original theory T'.

An Extension of the Nelson-Oppen Combination Schema

First of all, let us consider the situation of a theory T which does not contain LA and it is finitely axiomatized. Recall that we use a superposition prover to build satisfiability procedures for theories which can be finitely axiomatized. Since superposition can handle (at least in theory) any sets of clauses, we can use the SMT algorithm in Figure 1 (at least) as a semi-decision algorithm whenever we consider a theory T which is axiomatized by a finite set of formulae, which are easy to translate to Conjunctive Normal Form (CNF).

Now, let us consider the situation in which T is an extension of LA by a finite set Ax of axioms. There are two sub-cases two consider. First, Ax does not contain symbols of LA. In this case, T can be seen as the disjoint combination of LA and the theory axiomatized by Ax. If this last is stably-infinite then the N&O schema of Figure 2 is a satisfiability procedure for T; otherwise, it is only a semi-decision procedure. The second, more complex, case to consider is when the axioms of Ax contain symbols of LA. In this situation, we need to extend the schema of Figure 2, since exchanging elementary clauses is no more sufficient, even in simple situations. The key idea is to use the superposition prover as a mechanism to find ground instances of the quantified axioms which can be suitably used by the procedure for LA to detect unsatisfiability. As a consequence, the N&O schema must be modified in two ways: (i) the whole set of flat literals (i.e. also the arithmetic literals) is sent to the superposition prover and (ii) the superposition prover must send to the procedure for LA (and/or the module for case-splitting, cf. Figure 2) also the ground arithmetic facts which it has derived. We illustrate the technique by means of an example. Example Let us consider the theory T obtained as the extension of LA($\mathcal{R}$) with the following two axioms:

$$\forall U, V.(0 \leq U \quad \Rightarrow \quad \mathtt{uir}(V,U) \leq U)) \tag{11}$$
$$\forall U, V.(0 \leq U \quad \Rightarrow \quad 0 \leq \mathtt{uir}(V,U)) \tag{12}$$

where `uir` is a function which returns a random number constrained to be in the range satisfying the constraints above. Let us consider the problem of checking the T-satisfiability of the following formula in Disjunctive Normal Form (DNF):[8]

$$\begin{array}{ll} 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq (5-1) \wedge 0 \not\leq 0 & \vee \\ 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq (5-1) \wedge 0 \not\leq \mathtt{pv51} & \vee \\ 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq (5-1) \wedge \mathtt{pv51} \not\leq (5-1) & \vee \\ 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq (5-1) \wedge 0 \not\leq \mathtt{uir}(1, ((135300-1)-0)) & \vee \\ 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq (5-1) \wedge \mathtt{uir}(1, ((135300-1)-0)) \not\leq (135300-1) & \end{array} \tag{13}$$

Indeed, if each disjunct of (13) is T-unsatisfiable then (13) is T-unsatisfiable. The first three disjuncts are obviously unsatisfiable: the first is LA($\mathcal{R}$)-unsatisfiable because of $0 \not\leq 0$, the second is Boolean-unsatisfiable because of $0 \leq \mathtt{pv51}$ and $0 \not\leq \mathtt{pv51}$, and the third is also Boolean-unsatisfiable because of $\mathtt{pv51} \leq (5-1)$ and $\mathtt{pv51} \not\leq (5-1)$. It is easy to see that (13) can be simplified to

$$\begin{array}{ll} 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq 4 \wedge 0 \not\leq \mathtt{uir}(1, 135299) & \vee \\ 0 \leq \mathtt{pv51} \wedge \mathtt{pv51} \leq 4 \wedge \mathtt{uir}(1, 135299) \not\leq 135299 & \end{array}$$

while building its Boolean abstraction by evaluating ground terms. We are now left with the problem of checking the T-unsatisfiability of two conjunctions of literals which are not trivially unsatisfiable. Let us consider each case separately.

1. Each literal in the conjunction is turned into a unit clause and it is sent to the superposition prover together with the CNFs of (11) and (12), i.e.

$$0 \not\leq U \quad \vee \quad \mathtt{uir}(V,U) \leq U \tag{14}$$
$$0 \not\leq U \quad \vee \quad 0 \leq \mathtt{uir}(V,U) \tag{15}$$

[8]This proof obligation corresponds to `cl5_nebula_array_0020` in the benchmark set Array.

where U and V are implicitly universally quantified variables. Among many other facts, superposition derives (by resolution between (15) and the last literal) that $0 \not\leq 135299$, which is then sent to the satisfiability procedure for LA($\mathcal{R}$) so that unsatisfiability is immediately detected. Hence, we are entitled to conclude the T-unsatisfiability of this disjunct.

2. This case is similar to the previous. The only difference is that superposition derives $0 \not\leq 135299$ by resolution on (14) and the last literal.

Since the last two disjunct are T-unsatisfiable, we can conclude the T-unsatisfiability of (13).

Heuristics: definition unfolding and large theories

In order to make the extension of the N&O schema efficient, it is crucial to reduce the search space of the superposition prover, so that the least possible number of ground instances of the axioms in Ax should be considered. Even when the theory T does not contain LA, it is important to reduce the search space of the superposition prover in order to augment its predictability. A similar observation has already been done in [19] when using resolution-based theorem provers for software verification. We have found definition unfolding and the reduction of large theories particularly useful in this respect.

Definition unfolding

Expanding definitions is a crucial concern for automated theorem proving. Definitions represent concepts in a theory and are an important structuring mechanism. For example, the subset relationship $\subseteq$ is defined in terms of set membership $\in$. How a theorem prover handles such definitions can have a significant effect on its performance. Resolution-based theorem provers are often very weak on problems involving definitions because all formulae must be preliminary translated to CNF. So, for example, it is not possible to replace $x \subseteq y$ with $\forall e.(e \in x \Rightarrow e \in y)$ in a clause since this would introduce a new quantifier and the quantifiers have already been eliminated in the translation to CNF. A further problem which frequently arises in software verification is given by predicates defined by case-analysis on the values of their arguments; a large number of clauses are created by the translation to CNF so that the search space of the prover is significantly larger.

To overcome these difficulties and to make the prover more predictable, we have adapted the technique of definition unfolding of [20]. Let ϕ be a formula to be checked for satisfiability in the theory T, obtained as the extension of LA by a set Ax of axioms which contain (non-recursive) predicate definitions. We replace all the occurrences of predicate applications by suitable instances of the body of the definition. Let ϕ' be the formula obtained by this transformation and Ax' be obtained from Ax by deleting all predicate definitions. It is easy to see that ϕ is satisfiable in T iff ϕ' is satisfiable in T', obtained by extending LA with the axioms in Ax'.

If the body of the definition contains a quantifier, then there are two cases to be considered. If the occurrence of the predicate application has a positive (resp., negative) polarity and the quantifier is existential (resp., universal), then it can be Skolemized by replacing its bound variables with Skolem constants. Hence, we obtain a ground formula on which the SMT algorithm of Figure 1 can be directly invoked. Otherwise, i.e. the occurrence of the predicate application has a negative (resp., positive) polarity and the quantifier is universal (resp., existential), then we have two choices: (i) do not expand the definition so to avoid introducing new quantifiers in the formula and hope that superposition will be capable of coping with it or (ii) expand the definition and further processing the formula to eliminate the newly introduced quantifier as explained below before invoking the SMT algorithm. We illustrate our technique with an example not involving quantifiers. Example Let us consider the theory T obtained by extending LA($\mathcal{R}$) with the following predicate definition:

$$\forall n.(\mathit{IsInt0\text{-}199}(n) \Leftrightarrow n = 0 \vee n = 1 \vee \cdots \vee n = 198 \vee n = 199) \tag{16}$$

expressing the fact that the number n is an integer in the range between 0 and 199. We want to check the unsatisfiability in T of the following formula:

$$\mathit{IsInt0\text{-}199}(x) \wedge (\neg 0 \leq x \vee \neg x \leq 199). \tag{17}$$

Now, if we do not perform definition unfolding, (16) is translated to CNF, thereby resulting in 201 clauses: 200 of the form $U \neq k \vee \mathit{IsInt0\text{-}199}(U)$ for $k \in \{0, ..., 199\}$ and one of the form $\bigvee_{k=0}^{199} U = k \vee \neg\mathit{IsInt0\text{-}199}(U)$ for U an implicitly universally quantified variable. By resolution between $\mathit{IsInt0\text{-}199}(x)$ of (17) and the last clause above, we get the clause $\bigvee_{k=0}^{199} x = k$, which must be passed to the the case-splitting module (cf. Figure 2) so that each one of 200 cases is considered first with $\neg 0 \leq x$ and then again with $\neg x \leq 199$ (for a grand total of 400 case splits). So, all the burden is on the case-splitting module while the available satisfiability solver of the SMT algorithm (cf. Figure 1), which is much more suited to

```
function grounding ($\phi$: quantified formula)
    $\phi_0 \longleftarrow existential\_closure(\phi)$
    $\phi_1 \longleftarrow minimize\_scope(\phi_0)$
    $\phi_2 \longleftarrow drop\_existentials(\phi_1)$
    $(\phi_3, \Delta) \longleftarrow rename\_and\_define(\phi_2)$
    return $(\phi_3, \Delta)$
end
```

Figure 3: Handling Quantifiers

handle large formulae, remains idle. To avoid this ineffective use of the resources, it is better to expand the definition (16) in (17) so to obtain the following formula:

$$(x = 0 \vee x = 1 \vee \cdots \vee x = 198 \vee x = 199) \wedge (\neg 0 \leq x \vee \neg x \leq 199) \tag{18}$$

which must be checked for unsatisfiability in LA($\mathcal{R}$). In this way, the 400 case splits are handle by the available satisfiability solver and only the satisfiability procedure for LA($\mathcal{R}$) can be used, thereby avoiding the overhead of invoking the superposition prover to derive suitable instances of the definition.

Reduction of large theories.

It is not unusual that software specifications comprise hundreds of axioms and the proofs of the unsatisfiability of conjectures are usually shallow and consist of analyzing a large number of cases. Resolution-based theorem provers are known to have quite impressive performances on problems consisting of few axioms and conjectures (usually extracted from mathematical problems) whose proofs are quite deep and contain simple case analyses. Hence, it is not surprising that resolution-based theorem provers do not perform well in the context of software verification (see e.g., [21] for an in-depth discussion on this issue). The problem is that provers are lost in the search space although the majority of the axioms are irrelevant to the proof under consideration.

In [19], a technique to find out an approximation of the relevant axiom set for the proof of a conjecture is presented. We use an adaptation of such a technique as a pre-processing step of the SMT algorithm of Figure 1 by introducing suitable syntactic constructs to structure the set Ax of axioms. The axiomatization is structured into *theories*. A theory T_i defines the semantics of a set of symbols S_i by means of a set of axioms Ax_i, and possibly the use of previously defined theories via an explicit importation clause.

The idea underlying the reduction algorithm is to use a directed acyclic graph whose nodes are associated to the theories; an edge from the node n_1 to the node n_2 represents the fact that the theory T_1 associated to n_1 is a superset of the theory T_2 associated to n_2, i.e. S_1 is obtained by extending S_2. So, given a formula ϕ, it is sufficient to compute the set of symbols occurring in ϕ, find which the set N_ϕ of nodes with theories containing the symbols of ϕ and to form the relevant axiom set by transitive traversal of the nodes reachable from N_ϕ. Finally, in order to incorporate the treatment of arithmetic in this framework, which is not finitely axiomatized in our case, our algorithm considers an implicit theory T_a, such that Ax_a is empty and S_a are the arithmetic symbols, and adds an implicit edge to N_a from all nodes representing theories where these symbols appear. For more details and an extensive discussion, the interested reader is referred to [19].

We evaluate the impact of these two techniques on a set of benchmarks.

Handling quantified formulae

So far, we have considered the problem of checking the (un-)satisfiability of ground formulae in a given theory. In many software verification scenarios, considering only ground formulae is too restrictive. The crux to lift the SMT algorithm of Figure 1 to handle quantified formulae is again the usage of a superposition prover to implement $T\text{-}satisfiable$.

For simplicity, here we consider a theory T axiomatized by the a set of clauses Ax. However, the technique can be straightforwardly adapted to consider LA($\mathcal{R}$). Let ϕ be a first-order formula (containing quantifiers), the idea is to transform ϕ into a ground formula ϕ_g by replacing the quantified sub-formulae of ϕ with fresh propositional letters and to add the definitions Δ of these to the axioms Ax in such a way that $Ax \wedge \phi$ is satisfiable iff $Ax \wedge \Delta \wedge \phi_g$ is. The algorithm for pre-processing is given in Figure 3. Let ϕ be a first-order formula and $v_1, ..., v_n$ its free variables. The formula returned by $existential\_closure(\phi)$ is $\exists v_1, ..., v_n.\phi$. It is easy to see that ϕ is satisfiable iff $existential\_closure(\phi)$ is.

Since we want to add a set Δ of "small" formulae to Ax and to preserve as much as possible the Boolean structure of the formula (so to maximally exploit the Boolean solver of the SMT algorithm in Figure 1), $minimize\_scope$ implements rules to move quantifiers as far inwards as possible. Roughly, we use all the rules to transform a formula into prenex form[9] but in the opposite direction [20]. For example, the formula $\forall x.(\phi \wedge \psi)$ is transformed to $(\forall x.\phi) \wedge \psi$ if the variable x does not occur in ψ and the formula $\exists x.(\phi \Rightarrow \psi)$ to $(\forall x.\phi) \Rightarrow \psi$, again if x does not occur in ψ. These transformations are equivalence preserving and terminate as they always reduce the depth of a formula starting with a quantifier. In practice, these classic (anti)prenexing rules do not always yield an optimal result, as they take into account neither the associativity and commutativity of conjunction and disjunction, nor the properties of multi-variables quantifications. Consider indeed the following :

$$\begin{aligned}
\forall x, y \bullet [p_1(x) \wedge p_2(x,y) \wedge p_3] &= \forall x \bullet [\forall y \bullet [p_1(x) \wedge (p_2(x,y) \wedge p_3)]],\\
&= \forall x \bullet [p_1(x) \wedge (\forall y \bullet [p_2(x,y) \wedge p_3])],\\
&= \forall x \bullet [p_1(x) \wedge (\forall y \bullet [p_2(x,y)] \wedge p_3)],\\
&= \forall x \bullet [p_1(x) \wedge ((\forall y \bullet [p_2(x,y)]) \wedge p_3)].
\end{aligned}$$

At this point, no simplification rule can be applied, as x is a free variable in both operands of the conjunction under the scope of the outermost quantification, which still applies to p_3. However, there is a better solution:

$$\forall x, y \bullet [p_1(x) \wedge p_2(x,y) \wedge p_3] = p_3 \wedge \forall x \bullet [p_1(x) \wedge \forall y \bullet [p_2(x,y)]].$$

The next example illustrates the influence of the variable order in the quantification:

$$\forall x, y, z \bullet [p_1(x,z) \wedge p_2(y,z)] = \forall x \bullet [y \bullet [z \bullet [p_1(x,z) \wedge p_2(y,z)]]].$$

Here, no simplication rule can be applied. However, there is a better solution:

$$\forall x, y, z \bullet [p_1(x,z) \wedge p_2(y,z)] = \forall z \bullet [\forall x \bullet [p_1(x,z)] \wedge \forall y \bullet [p_2(y,z)]].$$

The rules implemented in the $minimize\_scope$ routine are aware of aforementioned properties and provides a more aggressive minimization of quantifier scope than the classical rules [20].

We are now ready to start the elimination of quantifiers. We begin by eliminating the existential quantifiers by a restricted form of Skolemization. Invoking $drop\_existentials$ on a first-order formula ϕ returns the formula obtained by repeatedly eliminating existential quantifiers in an outermost way. More precisely, $drop\_existentials$ exhaustively applies the following transformations: $\phi[\exists x.\psi]_p$ ($\phi[\forall x.\psi]_p$) is rewritten to $\phi[\psi[x/c]]_p$, where c is a fresh constant, p is the outermost position at which a positive (negative, resp.) occurrence of an existentially (universally, resp.) quantified sub-formula is in ϕ, and $\exists x.\psi$ ($\forall x.\psi$, resp.) does not contain free variables. This process terminates since each application of the rules removes an existential or a universal quantifier. We can eliminate the remaining quantifiers by substituting each quantified sub-formula ψ with a fresh propositional letter q and recording its definition $q \Leftrightarrow \psi$. The function $rename\_and\_define$ performs this with some optimizations. More precisely, invoking $rename\_and\_define$ on a first-order formula ϕ returns the pair (ϕ', Δ) which is obtained by exhaustively applying the following transformations: $\phi[\psi]_p$ is rewritten to $\phi[q]_p$, where q is a fresh propositional letter, p is the outermost position at which a quantified sub-formula occurrence is in ϕ, and ψ does not contain free variables. Furthermore, we add $q \Rightarrow \psi$ ($\psi \Rightarrow q$) to Δ if ψ is a positive (negative, resp.) sub-formula occurrence of ϕ. Since there are only finitely many (quantified) sub-formulae, this process obviously terminates. It is not difficult to prove that the formula ϕ is satisfiable iff $\phi_g \wedge \Delta$ is, where $(\phi_g, \Delta) = rename\_and\_define(\phi)$ (see [9] for a formal development).

The formula ϕ_g and the theory axiomatized by $Ax \cup \Delta$ can be sent to the SMT algorithm of Figure 1.

IMPLEMENTATION AND EXPERIMENTS

We have implemented the techniques described above in a system called **haRVey**.[10] It makes use of several existing tools: Flotter[11] to transform the axioms of the background theory to CNF, D. Long's BDD library[12] or zChaff for Boolean reasoning, the E prover[13] for T-$satisfiable$, and the ATerm library[14] for transforming formulae and axioms as well as a basis

[9]A formula is in prenex form if it has the following structure $Q_1x_1...Q_nx_n.\phi$, where Q_i is either $\forall$ or $\exists$, x_i is a variable ($i = 1, ..., n$), and ϕ is a quantifi er-free formula whose free variables are $x_1, ..., x_n$.

[10]`http://www.loria.fr/equipes/cassis/softwares/haRVey`

[11]`http://spass.mpi-sb.mpg.de`

[12]`http://www-2.cs.cmu.edu/~modelcheck/bdd.html`

[13]`http://www4.informatik.tu-muenchen.de/schulz/WORK/eprover.html`

[14]`http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ATermLibrary`

Table 1: Experimental Results.

	haRVey (%)	NASA (%)
Array	100	96.4
Init	94.4	76.8
In-use	78.9	68.4
Symm	100	50.0
Norm	70.8	51.8

for the communication between the various tools. **haRVey** accepts as input the (possibly structured) axiomatization Ax extending LA($\mathcal{R}$) and the first-order formula φ to be proved (un-)satisfiable in a LISP-like syntax. The set Ax of axioms is searched for definitions which are expanded in the input formula φ (related heuristics are presented below), quantified sub-formulae are eliminated, and the set of axioms suitably modified (as explained below). At this point, the theory is reduced and passed to the superposition prover. Finally, the ground formula obtained after unfolding and "grounding" (cf. Figure 3) is passed to the SMT algorithm of Figure 1. If arithmetic reasoning is required, the superposition prover may be combined with a satisfiability procedure for LA by using the extension of the N&O schema described in this paper.

For benchmarks, we have considered the proof obligations generated by the certification of auto-generated aerospace software [22]. For certification, the goal is not to ensure full correctness but to check that a program satisfy a certain level of safety. A typical security problem is that a program does not access out-of-bound elements in an array. Using a Hoare logic approach, one can automatically add annotations to a program, generate the corresponding proof obligations, and then discharge them. In [22], five safety properties are considered on four auto-generated aerospace programs written in C (ranging from around 400 to more than 1000 lines of code): 366 valid and 2 invalid proof obligations are obtained. The mean value of the cardinality of the set Ax of axioms is 79.

The experiments have been carried out on a Pentium IV 2Ghz running Linux with 256 Mb of RAM and 1Gb of disk space. We have set a time-out of 30 seconds per proof obligation. The results are depicted in Table 1. The first column lists the five safety properties used to generate the proof obligations. The second (third) column records the percentage of proof obligations successfully discharged by **haRVey** (the system developed by NASA and described in [22], respectively).[15] In the cases of Array and Symm, we successfully discharge all proof obligations, for Init our system scores definitely better than NASA, while for In-use and Norm **haRVey** is still better but not as good as we would like. The problem with In-use is that corresponding proof obligations are large and the system frequently times out. For Norm, the problem is that only a partial axiomatization of the operator `sum`, which takes a vector and returns the sum of its values, is supplied. This is so because it is not possible to have a complete axiomatization for this operator in first-order logic as explained in [22].

To evaluate the impact of the proposed heuristics, we have repeated our experiments in the following configurations of the system. First, we have disabled theory reduction: we have drastically reduced our success rates of 60%. Second, we have prevented the expansions of definitions: we have reduced our success rates of only 3%. Finally, to evaluate the effectiveness of our integration of LA in the system, we have eliminated all the axioms regarding LA from the benchmarks in Array and we have re-rerun the system using our extension of the N&O schema on the resulting proof obligations. **haRVey** is still capable of discharging all modified proof obligations.

We believe that these results are encouraging and confirm the viability of our approach for effective automated software verification tasks such as those arising in the engineering of areospatial systems. Other successful applications of the techniques described in this paper can be found in [9, 23, 14].

References

[1] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001*, volume 2057 of *LNCS*, pages 103–122, 2001.

[2] R.S. Boyer and J S. Moore. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. *Machine Intelligence*, 11:83–124, 1988.

[3] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Hand. of Automated Reasoning*. 2001.

[15] We use the results obtained on the 366 proof obligations and not those obtained after applying simplifi cations. See again [22] for details.

[4] M.Bozzano, R.Bruttomesso, A.Cimatti, T.Junttila, P.v.Rossum, S.Schulz, and R.Sebastiani. The mathsat 3 system. In *Conference on Automated Deduction (CADE-20)*, 2005.

[5] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Proc. of the 16th Int. Conf. on Computer Aided Verification (CAV'04)*, LNCS. Springer, 2004.

[6] Alessandro Armando, Claudio Castellini, Enrico Giunchiglia, Massimo Idini, and Marco Maratea. TSAT++: an Open Platform for Satisfiability Modulo Theories. In *Proc. 2nd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'04)*, 2004.

[7] J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: Integrated Canonization and Solving (Tool presentation). In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 246–249. Springer-Verlag, 2001.

[8] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th Intl. Conf. Computer Aided Verification (CAV 2004)*, 2004.

[9] D. Déharbe and S. Ranise. Light-Weight Theorem Proving for Debugging and Verifying Units of Code. In IEEE Comp. Soc. Press, editor, *Proc. of the Int. Conf. on Software Engineering and Formal Methods (SEFM03)*, 2003.

[10] A. Armando, S. Ranise, and M. Rusinowitch. A Rewriting Approach to Satisfiability Procedures. *Info. and Comp.*, 183(2):140–164, June 2003.

[11] M. Rusinowitch. Theorem-proving with Resolution and Superposition. *JSC*, 11(1&2):21–50, January/February 1991.

[12] Stephan Schulz. E – a brainiac theorem prover. *AI Communications*, 2002.

[13] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. of the ACM*, 27(2):356–364, 1980.

[14] D. Déharbe, A. Imine, and S. Ranise. Abstraction–Driven Verification of Array Programs. In *In Proc. of AISC'04*, LNCS, 2004.

[15] S. Ranise, C. Ringeissen, and D.-K. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *In Proc. of ICTAC'04*, volume 3407 of *LNCS*, 2004.

[16] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Lab., 2003.

[17] H. Kirchner, S. Ranise, C. Ringeissen, and D. K. Tran. On Superposition-Based Satisfiability Procedures and their Combination. In *Proc. of the 2nd International Colloquium on Theoretical Aspects of Computing (ICTAC'05)*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[18] P. van Hentenryck and T. Graf. Standard Forms for Rational Linear Arithmetics in Constraint Logic Programming. *Ann. of Math. and Art. Intell.*, 5:303–319, 1992.

[19] W. Reif and G. Schellhorn. *Automated Deduction—A Basis for Applications*, volume 1, chapter Theorem Proving in Large Theories. Kluwer Academic Pub., 1998.

[20] Andreas Nonnengart and Christoph Weidenbach. *Handbook of Automated Reasoning*, chapter Computing Small Clause Normal Forms. Elsevier Science, 2001.

[21] J. Schumann. Automated Theorem Proving in High-Quality Software Design. *Intelletics and Computational Logic*, 19, 2000. Applied Logic Series.

[22] E. Denney, B. Fischer, and J. Schumann. Using automated theorem provers to certify auto-generated aerospace software. In *Proc. of Int. Joint Conf. On Automated Reasoning (IJCAR'04)*, volume 3097 of *LNCS*, 2004.

[23] Jean-Franois Couchot, Frédéric Dadeau, David Déharbe, Alain Giorgetti, and Silvio Ranise. Proving and debugging set-based specifications. *Electronic Notes in Theoretical Computer Science*, 95:189–208, 2004. Proc. of the 6th Workshop on Formal Methods.

MODEL CHECKING C SOURCE CODE FOR EMBEDDED SYSTEMS

Bastian Schlich and Stefan Kowalewski
Embedded Software Laboratory, RWTH Aachen University
Ahornstr. 55, 52074 Aachen, Germany
{Schlich, Kowalewski}@informatik.rwth-aachen.de
http://www-i11.informatik.rwth-aachen.de

ABSTRACT

In this paper, the applicability of existing model checking approaches for C code to embedded systems is studied. We first provide a survey of four methods and fourteen tools from the recent literature. Then the major challenges for the application of these approaches to embedded C code are presented. As a basis for evaluation, we discuss the limitations of the different C model checkers in the light of those challenges. The main part of the contribution is a case study in which *CBMC* as one representative model checker is applied to a specific piece of micro-controller code. Based on the experiences of this study we summarize the main problems which still have to be solved, if C model checkers shall be applied to embedded code, and suggest directions for future work.

INTRODUCTION

Model checking is often considered as a promising future tool for analyzing embedded software and improving its quality. The corresponding industries have become interested, and several research projects produced tools, in a few cases even offered and supported commercially. However, model checking technology is still far from being a well established standard tool in the development of embedded systems. Apart from the well known complexity issues, one of the limiting factors to applicability is that most of the available model checking tools have a proprietary input model. Consequently, developers would have to re-model their specifications and implementations in order to feed them into the model checker. This is usually not considered to be worth the effort. Very few model checking environments provide interfaces to standard development tools (e.g. Statemate[1], MATLAB/Simulink[2], or ASCET[3]) or are integrated into such tools (e.g. SCADE[4]). In these cases, however, it is only the logic and discrete dynamics part of the models which can be analyzed. The equally important arithmetic and continuous control parts are neglected. It is our experience that this is regarded as a major disadvantage by developers in industry. A further issue is the fact that often several different development tools are used in the same company. This would raise the need for either the same number of corresponding model checkers or translations between the different representations.

A remedy to these problems could be to apply model checking to the C code. A representation of the software in C code can be expected to exist in some development phase for any embedded software project. It is either generated from tools like MATLAB/Simulink or developed manually. So, it is not necessary to consider different formats of different development tools. Apart from that, the C code is more comprehensive, as it includes the logic and arithmetic parts of the specification models as well as the C code, which is embedded in these models. This idea motivated a study we presented in a technical report[5]. In this study we surveyed fourteen C source code model checkers with emphasis on their applicability to embedded software.

[1] http://www.ilogix.com/statemate/statemate.cfm
[2] http://www.mathworks.com/
[3] http://en.etasgroup.com/products/ascet/index.shtml
[4] http://www.esterel-technologies.com/products/scade-suite/overview.html
[5] Schlich, Bastian and Kowalewski, Stefan, "C model checking: A survey." Technical Report RWTH-I11-2005-2, Informatik XI, RWTH Aachen University, 2005.

This paper is organized as follows. In section “Model Checking Approaches” we describe the different approaches of model checking C code which we found in the literature. Then in section “Challenges” we describe the challenges that arise when model checking C code for embedded systems. In section “Evaluation” we evaluate the model checkers. After that we present a case study in which we tried to model check C code for the ATMEL ATmega 16[6]. In the next section we summarize the problems we discovered and propose solutions to these problems. In the end we conclude.

MODEL CHECKING APPROACHES

In the following we describe the four most often used ways to model check C code. These four approaches are in alphabetic order:

- Bounded Model Checking (BMC)
- model checking with predicate abstraction using a theorem prover
- model checking with predicate abstraction using a SAT solver
- translation of the C code into a model of an existing standard model checker

For each approach we give a list of model checkers that use it. All C model checkers known to us are listed in Table 1. In the next section we evaluate each of these model checkers in detail. There are other ways to model check C code than listed here, but they are less often used. These four are used to handle the potentially infinite state space of a C program. In all these approaches the program is in some way transformed into an abstracted program that has a finite state space. This is required because the model checking algorithm has to visit all states.

In Bounded Model Checking this transformation is done by unwinding the potentially infinite constructs (e.g. *while* loops) only *n* times. This number *n* is the upper bound. For this reason this method is called Bounded Model Checking. An example for a Bounded C Model Checker is *CBMC*[7] (ref. 1). In the majority of cases *CBMC* is able to determine the upper bound *n*. If it fails, the user can provide an upper bound that is then used by *CBMC*. If the user provides this upper bound, it cannot be guaranteed that there is no counter-example that is longer than the upper bound. In this case *CMBC* can only be used as a tool to find errors and not to prove correctness, since errors can be missed.

The next two approaches in the table use predicate abstraction (ref. 2, 3). Model Checking using predicate abstraction is done via abstracting the data by predicates on this data. In the abstract program Boolean variables represent the predicates and the original data variables are eliminated. This type of abstract program is called Boolean program (ref. 4). Model Checking is then applied to this Boolean program. Since this abstract program is created by a conservative approximation it may happen that the model checker finds an error which has a trace that is not feasible. Then a refinement process has to adjust the set of predicates and create a finer abstraction. This loop is called Counter-Example Guided Abstraction Refinement (CEGAR) (ref. 4, 5). The difference between model checking with predicate abstraction using a theorem prover and using a SAT solver is the way in that the Boolean program is constructed. A model checker that uses a theorem prover builds the Boolean program by repeatedly calling a theorem prover. Whereas a model checker, that uses a SAT solver, only makes one call to the SAT solver. In this call the SAT solver computes the abstraction of the concrete transition relations. Examples for model checkers that use a theorem prover are: *BLAST*[8] (ref. 6), *BOOP*[9] (ref. 7), *MAGIC*[10] (ref. 8, 9), *MOPS*[11] (ref. 10) and *SLAM*[12] (ref. 11, 12). An example for a model checker that uses a SAT solver is *SatAbs*[13] (ref. 13, 14).

[6] http://www.atmel.com/dyn/products/product_card.asp?part_id=2010
[7] http://www-2.cs.cmu.edu/~modelcheck/cbmc
[8] http://www-cad.eecs.berkeley.edu/~blast
[9] http://boop.sourceforge.net
[10] http://www-2.cs.cmu.edu/~chaki/magic
[11] http://www.cs.ucdavis.edu/~hchen/mops/
[12] http://research.microsoft.com/slam/
[13] http://www-2.cs.cmu.edu/~modelcheck/satabs/

In the last approach the C code is transformed into a model used by a general purpose model checker. This has the advantage that these model checkers are widely spread and have efficient algorithms. On the other hand it has the disadvantage that all special knowledge of the C code and the hardware has to be used in the abstraction process, because these model checkers are not aware of this information. The different C model checkers that use this approach, all use abstractions to generate models that are finite. *FeaVer*[14] (ref. 15) and *FocusCheck* (ref. 16) are two model checkers that use this approach. *FeaVer* transforms C code into Promela code. This Promela code is then checked with the Spin[15] model checker. *FocusCheck* transforms the C code into XSB Prolog. From XSB Prolog a push-down transition system is generated and model checked.

Additional information can be found in our technical report.

CHALLENGES

In this section we describe challenges that arise when model checking C code for embedded systems and give hints if and how they could be handled. First we describe restrictions related to the C source code and then restrictions related to the chosen model checking approach. In section "Evaluation" we discuss the restrictions of the different C model checkers listed in Table 1 in detail.

Challenges: C Source Code

As aforementioned the described C code model checkers target at the verification of ANSI C code. But C source code found in e.g. micro-controllers is often not compatible to ANSI C. Incompatible features include for example:

- direct hardware accesses
- in-line assembly language statements
- compiler specific instructions
- hardware specific instructions

Dependent on the chosen micro-controller the adjustments to the chosen model checker are more or less complex. If the micro-controller uses the GCC compiler or an adaptation of the GCC compiler, the adjustments are less complex, since most of the C model checkers use GCC for preprocessing. If the chosen micro-controller uses its own compiler, the preprocessing process of the model checker has to be changed. This can be done by writing a parser that transforms the C code into the internal representation of the model checker. But this task is time consuming. All language constructs present in the language for the micro-controller, which are not handled by the model checker, have to be abstracted. An alternative to this is changing the model checker to handle these constructs. But this is often a complex task. Unwanted side effects could be introduced.

Challenges: Model Checking Approach

Furthermore there are restrictions originated from the way model checking is done in the model checker. In section Model Checking Approaches we described the different model checking approaches. In this section we describe the challenges that arise from these different approaches.

In Bounded Model Checking there exists the aforementioned restriction that errors can be missed if the bound *n* is not chosen properly. Anyhow even if *n* is chosen to small, this method can be used for debugging purposes.

In model checking with predicate abstraction using a theorem prover there are some more restrictions:

- exponential number of calls to a theorem prover (ref. 13)
- limited number of supported C constructs (ref. 13)
- limited support for pointer arithmetic (ref. 13)
- negligence of possible arithmetic overflows (ref. 13)
- negligence of data flow

[14] http://cm.bell-labs.com/cm/cs/what/modex
[15] http://spinroot.com/spin/whatispin.html

There are heuristics that try to handle the exponential number of calls to a theorem prover by e.g. limiting the number of calls per assignment. But that often leads to an increase in the number of refinement steps due to spurious counter-examples. At present there is no solution to this problem. The limited number of supported C constructs and the limited support for pointer arithmetic stems from the use of a general purpose theorem prover. Not all C operators can be handled by these theorem provers. In these general theorem provers, the program variables are modeled as unbounded integer values. Thus possible overflows in the C program are ignored. In the domain of embedded systems there are very restricted resources, e.g. 8 bit char variables are used, and Boolean values are bundled into char variables. A model that uses unbounded integer values is not precise enough, many errors will be missed. The only solution to this problem would lead to an exchange of the theorem prover. But at present we do not know a general purpose theorem prover that considers bounded data types. The last problem derives from the use of predicate abstraction. This technique depends on the abstraction of the data variables and hence cannot accurately observe the data flow. In C programs for embedded systems the data flow is more important than it is in drivers or protocols. This restriction cannot be addressed in this approach. In model checking with predicate abstraction using a SAT solver there are not that much restrictions:

- little restrictions on allowed C constructs (ref. 13):
    - no recursion
    - no dynamic memory allocation
- negligence of data flow

The restrictions on the allowed C constructs stem from the use of a SAT solver. The C program is transformed into a Boolean formula and this Boolean formula has to be finite. But recursion and dynamic memory allocation are two constructs that are potentially infinite. These restrictions cannot be addressed in this approach. The inaccurate consideration of the data flow stems from the use of predicate abstraction as aforementioned. Nevertheless this way of model checking can be used to model check C code for embedded systems, if these two constructs are not present in the C code.

The only general restriction stemming from the model transformation approach is that general purpose model checkers are used. These ignore the domain specific characteristics of the models that we are interested in. All other restrictions stem from the specific implementation of this approach. We discuss them in the next section.

EVALUATION

In this section we present the implementation specific restrictions of all model checkers that are shown in Table 1. This table gives an overview about all C code model checkers known to us. Only two of these, namely *CBMC* and *StEAM*, are capable of full ANSI C. The model checker *SatAbs* has little restrictions on the allowed C constructs. All other C model checkers have more rigid restrictions, or it is not known to us if they are able to model check full ANSI C. It is important to notice that all these model checkers focus on ANSI C source code. This is due to the fact that most of them deal with the verification of drivers and protocols. Most of them concentrate on the control flow and neglect the data flow. For software for embedded systems this data flow can be important.

BLAST, BOOP, MAGIC and *SLAM* have restrictions on the C source code because they use a general purpose theorem prover in their approach. General purpose theorem provers like e.g. Simplify or Zapato support only linear arithmetics on real numbers. So the other constructs used, have to be approximated by means of uninterpreted functions (ref. 14). To adapt the constructs that are not allowed they would have to change their approach and/or the theorem prover they use. Since *KISS* is an extension to *SLAM*, only *SLAM* has to be changed.

CBMC is capable of full ANSI C. It suffers from the already mentioned restriction regarding the upper bound. In section Case Study we present a case study in which we tried to extend *CBMC* to model check C code for embedded system. More information can be found there.

DiVer and *F-SOFT* are only assumed to have full ANSI C. NEC gives no detailed information about these model checkers.

FeaVer uses the tool Modex to translate the C source code into Promela. The translation uses a table that tells Modex how to abstract the different C constructs. The default abstraction of Modex is in many cases very conservative. Many constructs are abstracted to no operation (noop). It takes big manual effort to change that table for C programs for embedded systems. In many cases this results in a model that is quite too big and thus not manageable by Spin.

FocusCheck used to model check Cimpel but now it model checks ANSI C (ref. 16). Since they use CIL like *MAGIC* does, to translate C source code into a push-down system, they have some restrictions on the C source code. But they do not mention these restrictions in their paper. To adapt more C constructs they have to change from CIL to other C language frameworks.

MOPS is based on the compiler GCC version 2.x. It is only able to model check GCC 2.x compatible programs. Additionally these programs have to adhere to some other restrictions. The main focus of *MOPS* is the control flow of the program. Subsuming these facts it is not possible to use it for model checking C code for embedded systems.

SatAbs uses predicate abstraction like *BLAST, BOOP, MAGIC*, and *SLAM* do. But instead of using a theorem prover it uses a SAT solver. The little restrictions that *SatAbs* has on the source code arise from the use of a SAT solver. Every Boolean program has to be finite and therefore no recursion and no dynamic memory allocation are allowed.

StEAM compiles C source code to machine code for a virtual machine called Internet Virtual Machine (IVM). Then this virtual machine is steered and monitored while it simulates the code. To extend *StEAM* to handle constructs used in C code for embedded systems the compiler has to be extended. It could also be possible that the virtual machine has to be changed.

Zing transfers C code into a proprietary model. Then it model checks that model. This model transformation has to be changed in order to handle the needed C constructs. But we don't know yet if the *Zing* model is expressive enough to represent these constructs.

We have to mention that all C model checkers described in this section are well suited for the purpose that they were written for. In most cases this is the verification of drivers and protocols. We only wanted to evaluate if it is possible to use them for the verification of C code for embedded systems. More information to this topic can be found in our technical report.

CASE STUDY

In this section we describe a case study that we made at our institute. In this study we tried to extend the model checker *CBMC* to be able to model check C code for the ATMEL ATmega 16 micro-controller. The handling of the used C code should be easy for *CBMC*, since both *CBMC* and the ATmega16, use GCC as the compiler. The decision to use *CBMC* was made during the evaluation of the C code model checkers. We chose it because it is able to check full ANSI C and it uses Bounded Model Checking. By using Bounded Model Checking *CBMC* it able to find errors in programs, for which model checking without a bound would not terminate. We chose the ATmega 16 micro-controller because it is well structured and widespread. We can easily get real-world examples for testing purpose.

The program we want to analyze is shown in Figure 1. It is an implementation of the “intelligent” light switch example (ref. 17). If the button is pressed once, the light should be switched on to the dimmed state. If the button is pressed a second time within two seconds, the light is switched on completely. If the button is pressed a second time after more than two seconds, the light is switched off. In our implementation the LEDs (simulating the light) are connected to port A. The button is connected to one of the pins' of port B. We use two timers. Timer zero is used to debounce the button and timer one is used for measuring the two seconds delay. We use active polling without interrupts. The specification we want to prove is as follows:

- If the light is off and the button is pressed once, the light is in the dimmed state afterwards.
- If the light is off and the button is pressed twice within two seconds, the light is in the bright state.
- If the time between the both pushes of the button is greater than two seconds, the light is off.

When the program is given to *CBMC*, it gives an error message telling that it found a "+", but it expected a pointer. To understand this error message we have to analyze our source code. *CBMC* tells us that this error happens in line eight. Line eight reads as follows: `DDRA = 0xFF`. DDRA is the Data Direction Register of port A (DDRA). This instruction sets all pins at port A as output. The preprocessor transforms this instruction via header files into: `(*(volatile uint8_t *)((0x1A) + 0x20)) = 0xFF`. *CBMC* sees a dereference of a typecast of a pointer. But at the point where it expects the pointer, it finds an addition of two constants. Then the resulting constant is typcasted to pointer type and then dereferenced. For *CBMC* this is not valid because it is a direct memory access to a fixed memory location. But this is exactly what this instruction is expected to do. It should write the value `0xFF` into memory location `0x1A + 0x20`. The I/O-registers of the ATmega 16 have fixed addresses and reside in the memory at their address incremented by a fixed offset of `0x20`. For the model checking process we can replace this direct memory access by an access to an array. This array simulates the memory. To achieve this we could change our program, *CBMC*, or the header files. We do not want to change all our programs to model check them. By changing *CBMC* we could introduce side effects. Therefore we decided to change the header files. This only needs to be done once for each micro-controller. These header file can only be used for model checking, as the object code of a program compiled with these header files is useless.

We replace:
```
#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _MMIO_WORD(mem_addr) (*(volatile uint16_t *)(mem_addr))
```

by:
```
unsigned char array_for_data_address_space[96];
#define _MMIO_BYTE(mem_addr) array_for_data_address_space[mem_addr]
#define _MMIO_WORD(mem_addr) array_for_data_address_space[mem_addr]
```

The size of the array is 96, because the ATmega 16 has got 96 registers and I/O-registers. Type `char` is chosen, because the memory of the ATmega 16 is 8 bit wide. After this change the preprocessor transforms `DDRA = 0xFF` into `array_for_data_address_space[(0x1A) + 0x20)] = 0xFF`. Now *CBMC* can parse our program. But after some minutes of doing unwindings *CBMC* terminates with a core dump. The cause for this problem is a part of the `delay()` method. *CBMC* tries to determine the upper bound for unwinding the `while` loop, but it fails. The expression `!(TIFR & (1 << OCF0))` does not change in the scope of *CBMC*. `TIFR` is the Timer/Counter Interrupt Flag Register and `OCF0` is the Output Compare Flag 0 that is part of the `TIFR`. This flag is set by the hardware when a timer overflow occurs. This register and hence the flag are part of our introduced array. It is set to zero in the initialization phase but it is never changed, since the behavior of the hardware is not modeled. We have three possibilities to fix this error. We could submit the upper bound via command line. But this bound would then be used for all infinite constructs in our program. Another solution would be to let *CBMC* choose a value for this flag non-deterministically. However this would not reflect the correct behavior of the timer. We know that finally the timer will overflow, and the bit will be set until it is reset by our program. A non-deterministic choice would not reflect this behavior. The third solution would be to develop a module that simulates the behavior of the hardware. The timers, interrupts, and the I/O-registers could be simulated by this module. But from our point of view the coupling between *CBMC* and this module would be rather tight. The effort of changing this module to simulate another micro-controller would be too high. The implementation of this module would be easier, if *CBMC* would provide parallelism. In this case this module could be implemented as a parallel process. Parallelism would also be needed to implement an extension that model checks programs with interrupts.

Beside these technical issues we examined the provided specification capabilities of *CBMC*. It provides the following ones:
- pointer safety
- array bounds

- exceptions
- user-provided assertions

It is not possible to express our timed specification with these specification capabilities. Even the following approximation of our specification could not be expressed without introducing new variables:

- If the light is off and the button is pressed once, the light is in the dimmed state afterward.
- If the button is pressed twice, the lights are in bright state or off afterward.

Recapitulating we can say, that at this point it is possible to model check our program for pointer safety, array bounds, and exceptions. But it makes no sense for us to model check for user-provided assertions, without editing our program, introducing new variables, and changing the access to I/O-registers to calls to a function that returns non-deterministic values. We do not want that much changes in our C code because these changes can be very difficult in industrial-size programs.

Again we have to mention that *CBMC* is well suited for the purpose it is written for that is the verification of hardware. But it shows limitations when embedded C code has to be verified.

PROBLEMS AND SOLUTIONS

In this section we summarize the existing problems and state those that we want to address. Thereafter we propose solutions to the two main problems.

Problems

As shown before, the existing C code model checkers are not well suited for model checking C code for embedded systems. The first reason for this is that most of them aim for the verification of drivers and protocols only. They can handle ANSI C, but ignore the underlying hardware. They try to model check C code for every existing platform. We think that this goal is not achievable. For us it seems to be important to use the information, which we have about the hardware platform. The result is a model checker that is only suited for a special hardware platform. But this makes it possible to model check C code that otherwise would be uncheckable. Such an approach is promising because e.g. in the automotive industry a special hardware platform is used for a longer time period. The same hardware is used in many cars. For each new car new software has to be developed. If there would exist a model checker that could model check software for this hardware, this would increase the confidence in the software.

To afford this, we address the following problems of the existing C code model checkers:

- missing consideration of the hardware
    - interrupts
    - timer
    - I/O- register
    - Direct memory accesses
    - Embedded assembly language
- unsuitable specification techniques:
    - reachability in the source code
    - specification via function calls

The two main problems are the missing consideration of the hardware and the unsuitable specification techniques. The missing consideration of the hardware causes ignorance of the interrupts, the timers, and the I/O-registers. Direct memory accesses are reported as errors in many of the existing model checkers. And embedded assembly language statements are ignored in all cases. Thus C code for embedded systems cannot adequately be analyzed in these model checkers. The second important problem stems from the partly unsuitable specification techniques. Many of the model checkers do tests like pointer analysis, array bounds, etc. For these tests no specification is needed. But in case of C code for embedded systems these test often fail due to the aforementioned ignorance of the hardware. For specification purpose often reachability in the C code is used. But if specification of e.g. invariants is needed, one has to insert new variables to the source code. From our point of view this is not desirable, since this can introduce unwanted side effects. In some model checkers specification is done via function calls. The user specifies the parameter values and the resulting return values. But in many C programs for embedded systems only very few function calls are used. Therefore only a small part of the specification can be represented in these model checkers.

Solution Proposal

We propose the following solution. We plan to develop a model checker that model checks C code for the ATMEL ATmega 16 micro-controller. The model checker has to handle all constructs found in C code for this micro-controller. Our approach is depicted in Figure 2. In our approach the C code is compiled by the AVR-GCC to the assembly language. We keep debug information in the assembly language file. Then the model checker module checks the assembly language file. There are some advantages using the assembly language. First, all difficult C constructs are compiled to semantically equivalent assembly language constructs. These are easier to handle. The second advantage is that embedded assembly language now is also checked by the model checker. Possibly some errors introduced by the compiler are also found. Another advantage is that the assembly language code is the code that is finally transferred to the hardware. In the model checker there are two modules. One is responsible for the model checking process; the other is responsible for the simulation of the hardware. In the module that simulates the hardware, all the details for the handling of the timers, interrupts, I/O-registers, etc. are managed. We choose this modular approach to make the model checker adaptable to new hardware platforms. The coupling between the two modules should be loose. In a later extension also static timing analysis could be made for the assembly language files. The errors that are found by the model checking module should be shown in the C code file and not in the assembly language file. This is possible by using the debug information from the assembly language file.

Our second proposal refers to the specification techniques. In the model checker there should exist two different views for specification purpose: the users view and the developers view. The user only wants to specify about the states he can distinguish from outside the micro-controller. Those are the different states of the I/O-registers. If he uses the neXt (X) operator in his temporal logic formula, he has in mind the next change in the states that he can observe. He often is not interested in the changes inside the micro-controller. In contrast to this the developer wants to specify the behavior concerning the internal states of the micro-controller. For him, it should be possible to tell the model checker which variables are of interest to him. The same applies to the other temporal operators.

In this model checker it should also be possible to check for the compliance to company-wide design directives. These could include design directives for interrupts:

- If a special interrupt is activated, an interrupt service routine has to exist for this interrupt.
- If a special interrupt is activated, the global interrupts have to be activated.
- The global interrupts must not be activated, if no special interrupt is activated.
- A function is not allowed to change the state of the interrupts, unless it is the `init()`, `activateInterrupts()`, or `deactivateInterrupts()` function.

There can be design directives for other micro-controller specific behaviors.

A goal for the remote future would be to make it possible to model check timed behaviors of micro-controllers. In our case study we already gave an example of such a timed specification. The assembly language code could be annotated with the accurate number of CPU cycles. Then an average time per CPU cycle could be chosen. This model could then be checked for its timing behavior. But at the moment this goal seems to be out of range to us. We did some experiments in which we did this annotation and transformation by hand. But when we tried to model check the resulting models, we realized that the state space even for simple programs with simple specifications was quite too big.

CONCLUSION

The paper has discussed different C source code model checkers. The overall result is that none of them is currently able to model check C source code for embedded systems out of the box. All of them originate from an academic or research oriented environment. Most of them deal with the verification of drivers or protocols and are not intended to model check software for embedded systems. The main reasons for the lack of adequacy these C model checkers have for verifying C programs for embedded systems are the following:

- too restricted a set of analyzable C constructs
- inability to deal with hardware particularities
- partly inappropriate specification techniques

We have presented a case study that showed the existing problems in some detail. Despite the limitations shown here, C model checking is still a promising approach to the verification of embedded software. For every project it has to be checked if one of the existing model checkers handles all the constructs that are used in this project. If this is not the case but only small changes have to be made, one could extend an existing model checker as described in section Case Study. Bigger changes in existing model checkers should be avoided. The performance could be decreased and side effects could be introduced.

Since it has turned out that the existing model checkers for C code are not suited to the verification of embedded systems, we have listed requirements for a new C code model checker for embedded systems. We have proposed a new approach in which the assembly language code is used for the model checking process. This is promising because the constructs found in the assembly language are easier to handle. We do not have to predict compiler behavior. Another benefit of model checking at assembly level is that the embedded assembly statements can be analyzed. The most important point in our approach is that a module is integrated which will simulate the hardware features. This module should be interchangeable to allow changes in the hardware platform that is used. We propose also a specification technique that provides two views to the system, namely the user's view and the developer's view. With these views different granularities for the specification should be possible.

C model checking is far from being used in software development for embedded systems as a routine tool. However, it might be used in individual cases. Some of the introduced tools show the potential for model checking embedded C source code, but there is still a lot of work to be done in order to adapt them to industrial source code for embedded systems.

TABLES

Model Checker	**Institute**	**Model**	**Method**
BLAST	UC Berkeley	restricted ANSI C	predicate abstraction, CEGAR, theorem prover
BOOP	IST Graz	restricted ANSI C	predicate abstraction, CEGAR, theorem prover
CBMC	CMU	ANSI C	Bounded Model Checking
DiVer	NEC, CU Boulder	ANSI C (assumed)	no information provided
FeaVer	Bell Labs	restricted ANSI C	model translation into Promela
FocusCheck	Iowa State U.	Cimpel/ANSI C	translation into push-down system, constraint-solver
F-SOFT	NEC	unknown	no information provided
MAGIC	CMU	restricted ANSI C	predicate abstraction, CEGAR, theorem prover
MOPS	UC Davis, UC Berkeley	restricted ANSI C	Model Checking on CFG
SatAbs	CMU	restricted ANSI C	predicate abstraction, CEGAR, SAT solver
SLAM	Microsoft Research	restricted ANSI C	predicate abstraction, CEGAR, theorem prover
StEAM	University Dortmund	ANSI C	simulation with virtual machine IVM
KISS	Microsoft Research	-	extension to SLAM
Zing	Microsoft Research	ANSI C (assumed)	translation into Zing Model

Table 1. *Model Checker List*

FIGURES

```
//Intelligent Light Switch

#include "avr/io.h"                  //access to I/O-registers
char Brightness = 0;
char ButtonPressed = 0;

void init(void) {
    DDRA  = 0xFF;              //PORTA as output
    PORTA = 0xFF;
    DDRB = 0x00;               //PORTB as input
    PORTB = 0xFF;              //activate pull-up
    OCR0 = 0x46;               //timer 0
    OCR1A = 0x2DC7;            //timer 1
}

void delay (void) {
    TCNT0 = 0x00;
    TCCR0 = 0x0D;
    while (!(TIFR & (1 << OCF0))) {
    //noop
    }
    TCCR0 = 0x00;
    TIFR = (1 << OCF0);
}

int main (void) {
    init();
    while(1) {
        if (!(PINB & (1 << 0))) {
                delay();
                if (PINB & (1 << 0)) {
                        ButtonPressed = 1;
                }
        }
        if (ButtonPressed) {
                switch (Brightness) {
                case 0:
                        PORTA = 0x55;
                        Brightness = 1;
                        //activate timer 1
                        TCNT1 = 0x00;
                        TCCR1B = 0x0D;
                        break;
                case 1:
                        if (TIFR & (1 << OCF1A)) {
                                PORTA = 0xFF;
                                Brightness = 0;
                        } else {
                                PORTA = 0x00;
                                Brightness = 2;
                        }
                        //deactivate timer 1
                        TCCR1B = 0x00;
                        TIFR = (1 << OCF1A);
                        break;
                case 2:
                        PORTA = 0xFF;
                        Brightness = 0;
                        break;
                }
                ButtonPressed = 0;
        }
    }
    return(1);
}
```

Figure 1. *Implementation of the "intelligent" light switch example (ref. 17).*

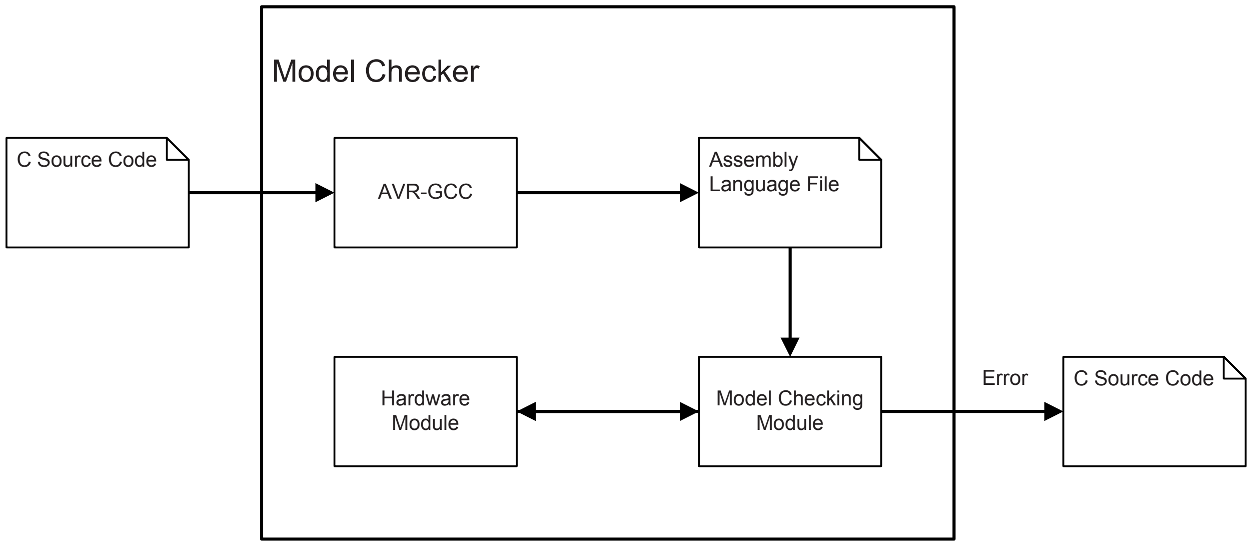


Figure 2. *Proposal for an architecture for a C model Checker*

REFERENCES

1. E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs." *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of Lecture Notes in Computer Science, 2004, pp. 168–176.

2. M. Colon and T. Uribe, "Generating finite-state abstractions of reactive systems using decision procedures." *Computer Aided Verification*, 1998, pp. 293–304.

3. S. Graf and H. Saidi, "Construction of abstract state graphs with PVS." *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, 1997, pp. 72–83.

4. T. Ball and S. Rajamani, "Boolean programs: A model and process for software analysis." Technical Report 2000-14, Microsoft Research, 2000.

5. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement." *Computer Aided Verification*, 1998, pp. 154–169.

6. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction." *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002, pp. 58-70.

7. G. Weißenbacher, "An Abstraction/Refinement Scheme for Model Checking C Programs." Master Thesis, Institute for Software Technology at Graz University of Technology, March 2003. – http://prdownloads.sourceforge.net/boop/thesis.ps.gz?download

8. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C." *Transactions on Software Engineering (TSE)*, Volume 30, Number 6, June 2004, pp. 388–402.

9. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav, "Efficient Verification of Sequential and Concurrent C Programs" *Formal Methods in System Design (FMSD)*,Volume 25, Issue 2-3, September-November 2004, pp. 129–166.

10. Chen and D. Wagner, “MOPS: An Infrastructure for Examining Security Properties of Software.” *ACM CCS 2002*, 2002, pp. 235–244.

11. T. Ball, S. Rajamani, “Automatically Validating Temporal Safety Properties of Interfaces.” *SPIN 2001*, Workshop on Model Checking of Software volume 2057, 2001, pp. 103–122.

12. T. Ball, and S. Rajamani, “The SLAM Toolkit.” *CAV’01,* volume 2102, July 2001.

13. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Predicate abstraction of ANSI–C programs using SAT.” *Formal Methods in System Design (FMSD)*, volume 25, September–November 2004, pp. 105–127.

14. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, “Satabs: Sat-based predicate abstraction for ANSI-C.” *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of Lecture Notes in Computer Science, 2005, pp. 570–574.

15. G.J. Holzmann and M.H. Smith, “Software Model Checking: Extracting verification models from source code.” *Formal Methods for Protocol Engineering and Distributed Systems (FORTE/PSTV99)*, October 1999, pp. 481–497,.

16. C. W. Keller, D. Saha, S. Basu, and S. A. Smolka, “Focuscheck: A tool for model checking and debugging sequential c programs.” *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of Lecture Notes in Computer Science, February 2005, pp. 563–569.

17. K. G. Larsen and P. Pettersson, “Timed and hybrid systems in Uppaal2k.” Presentation at MOVEP 2000, June 2000.

TOWARDS THE VERIFICATION OF HUMAN-ROBOT TEAMS

Michael Fisher*, Edward Pearce, Mike Wooldridge
Department of Computer Science, University of Liverpool, UK

Maarten Sierhuis, Willem Visser
RIACS/NASA Ames Research Center, Moffett Field, CA, USA

Rafael H. Bordini
Department of Computer Science, University of Durham, UK

ABSTRACT

Human-Agent collaboration is increasingly important. Not only do high-profile activities such as NASA missions to Mars intend to employ such teams, but our everyday activities involving interaction with computational devices falls into this category. In many of these scenarios, we are expected to *trust* that the agents will do what we expect and that the agents and humans will work together as expected. But how can we be sure? In this paper, we bring together previous work on the verification of multi-agent systems with work on the modelling of human-agent teamwork. Specifically, we target human-*robot* teamwork. This paper provides an outline of the way we are using formal verification techniques in order to analyse such collaborative activities. A particular application is the analysis of human-robot teams intended for use in future space exploration.

INTRODUCTION

In our previous work, we have developed techniques for verifying (using model checking) multi-agent systems [1, 2]. The agents involved are rational/intelligent [25] and are represented in a high level language describing their beliefs, intentions, etc. While this has potential to be used in scenarios where rational agents are used to control critical applications, there is a need to consider the verification of situations with more human involvement.

Our work on agent verification has been developed in collaboration with NASA. Rather than deploying completely autonomous, human-free, space missions, NASA is now turning to joint human-agent (typically, human-robot) teams as a way to handle more advanced space missions

*Principal Contact: `M.Fisher@csc.liv.ac.uk` within the future. Yet, it is still important to ensure that the humans and agents will work together to achieve their goals.

Fortunately, attempts have been made to model human-agent activity, specifically via the Brahms framework [21, 20]. Brahms describes teamwork at a high level of abstraction yet, since its intended semantics is quite close to BDI models of agency [17], there is a possibility that agent verification techniques can be relevant here. In particular, if we can characterise (simple forms of) human behaviour as agent behaviour, then we ought to be able to utilise our previous work to verify human-agent interactions. This paper describes our approach in this area.

The structure of the paper is as follows. We first provide some background concerning agents and their use in space missions, then we outline our previous work on the verification of multi-agent systems. The following section then describes the need for human-agent (specifically human-robot) teams, and after that we address the problem of verifying the behaviour of such human-robot teams. Some concluding remarks are given at the end.

BACKGROUND

Rational Agents

An agent can be seen as an *autonomous* entity. Thus, the agent makes its own decisions about what to pursue. We are particularly concerned with *rational agents*, which can be seen as agents that make *explainable* decisions on how to act so as to achieve their goals in the best possible way. Thus, the key new aspects that such agents bring is the need to consider, when designing or analysing them, not just what they do but *why* they do it. Since agents are autonomous, understanding why an agent chooses a course of action is vital. In [26], a number of additional aspects of agents are described, in particular their *pro-active*, *reactive*, and *social* attributes. Although we do not see these as essential, most of the agents considered here do, indeed, incorporate such notions.

A number of logical theories of (rational) agency have been developed, such as the BDI [16, 17] and KARO [12, 14] frameworks (BDI stands for “Belief-Desire-Intention”, and KARO for “Knowledge, Abilities, Results, and Opportunities”). One reason for using logic in agent-based systems is that a formal semantics comes (more or less) for free. From this it follows that the semantics of logic-based agents is strongly dependent on their underlying logic. The foundations of such agent description frameworks are usually represented as (often complex) non-classical logics. In addition to their use in agent theories, where the basic representation of agency and rationality is explored, these logics often form the basis for agent-based formal methods and, as we will see later, agent-based formal verification techniques.

Towards Rational Agents in Space Missions

Software is fast becoming an enabling technology for space missions. For example, the International Space Station (ISS) contains millions of lines of code that amongst other things supports the inter-operability of US and Russian modules. The true future of software in space applications are however embodied in missions such as the Remote Agent from Deep-Space 1 [18] and the Demonstration of Autonomous Rendezvous Technology (DART) [8] where spacecraft were autonomously controlled by software systems. Deep space missions require such autonomous behaviour since communication delays make earth-controlled missions almost impossible. Autonomy is also a major cost driver for NASA since human controlled missions require large earth-based teams for support. See Figure 1 for an outline of why *rational agents* are increasingly used to provide a high-level abstraction/metaphor for building complex/autonomous space systems.

$$\begin{tikzcd} \left.\begin{matrix}\mathit{Uncertain}\\ \mathit{environments}\end{matrix}\right\} \arrow[r] & \text{more `Intelligence'} \arrow[d] \\ \left.\begin{matrix}\mathit{Cooperation}\\ \mathit{Coordination}\end{matrix}\right\} \arrow[r] & \begin{matrix}\textbf{RATIONAL}\\ \textbf{AGENTS}\end{matrix} \\ \left.\begin{matrix}\mathit{Communication}\\ \mathit{problems}\end{matrix}\right\} \arrow[r] & \text{more Autonomy} \arrow[u] \end{tikzcd}$$

Figure 1: Motivations for using Rational Agents in Space Applications

Autonomous software is, however, hard to verify due to the uncertain/nondeterministic environments in which it executes. Yet, it is essential to attempt such verification since autonomous systems are amongst the most complex (and error prone) systems to develop. For example, the Remote Agent had a software deadlock during flight (although the mission completed successfully) and the DART mission failed (although the reasons are not clear yet) [11].

It is also very likely that autonomous systems will contain many rational agents that will need to interact and share information and resources: multiple space probes requiring collision avoidance, multiple surface rovers, autonomous docking between vehicles, etc. Additionally, NASA's new focus on doing human-robotic exploration of the Moon and Mars, brings human "agents" into the picture - this might simplify some of the autonomy requirements, but increase safety and certification concerns.

Thus, there is a clear path, not only towards rational agents, but also towards teams of such agents and humans. Such teams must be able to coordinate their activities, for example to avoid collisions and to cooperate to achieve some common goal. Although there has been relatively little work on the verification of BDI agents, there has been considerably more work on analysing teamwork, in particular where humans and agents interact to achieve common goals. A prominent approach is that of TEAMCORE developed by M.Tambe and colleagues [15], and an example of a significant application they have developed is the DEFACTO system [19], which coordinates the action of agents (e.g., representing fire engines) and humans for disaster response.

VERIFYING AGENT BEHAVIOUR

As seen above, the technology offered by autonomous software agents, particularly rational agents, is very appealing. Rational agents can:

- adapt to uncertain environments;

- solve problems independently;
- communicate and collaborate with other agents;
- learn new behaviour; etc.

However, can we be sure such agents will behave as expected? If rational agents are to have even partial control of critical missions, then we have to be able to trust them. But how can we check this? Our approach is to use *logical* verification techniques.

Logics for Precise Description — Logics provide an unambiguous formalism for describing the behaviour of systems. There is a wide variety of logics that can be developed for different scenarios, such as:

- dynamic communicating systems $\longrightarrow$ temporal logics;
- systems managing information $\longrightarrow$ logics of knowledge;
- autonomic/intelligent systems $\longrightarrow$ logics of goals, intentions, aims;
- situated systems $\longrightarrow$ logics of belief, contextual logics;
- systems in uncertain environments $\longrightarrow$ probabilistic logics.

Importantly, combinations of such logics are needed, so that we can specify high-level properties such as:

> "a rational agent has a problem that it $\boldsymbol{A}$ims to solve, but $\boldsymbol{B}$elieves that it needs help (i.e., the agent cannot solve the problem itself), so in the $\boldsymbol{N}$ext moment in time, its $\boldsymbol{G}$oal will be to get help."

in a compact and precise (unambiguous) formula like this:

$$(\boldsymbol{A}\,\texttt{solve-problem} \land \boldsymbol{B}\,\texttt{need-help}) \rightarrow \boldsymbol{N}\,(\boldsymbol{G}\,\texttt{get-help}).$$

When writing specifications of the properties required of agent-based systems in particular, it helps to use the same kinds of abstraction that we use to build the system itself (such as those mentioned above, particularly beliefs and goals).

Verifying Logical Descriptions — Analysing agent systems in order to make sure they will run as expected is a complex task. However, if we can provide both an abstract description of the agent system in question, and a logical description of the requirements/properties to be checked, then we can carry out logical verification, using a variety of techniques.

One such technique is Model Checking [7], and one of the reasons for this technique being particularly popular is that it can, potentially, make the whole verification process completely automated. This led to the development of very sophisticated model checkers such as SPIN [13] and NASA's Java Pathfinder [24, 10].

Verifying Multiple Agents — This form of verification (by model checking), which checks all possible behaviours of an agent situated in an environment, extends naturally to multi-agent scenarios, as seen in Figure 2.

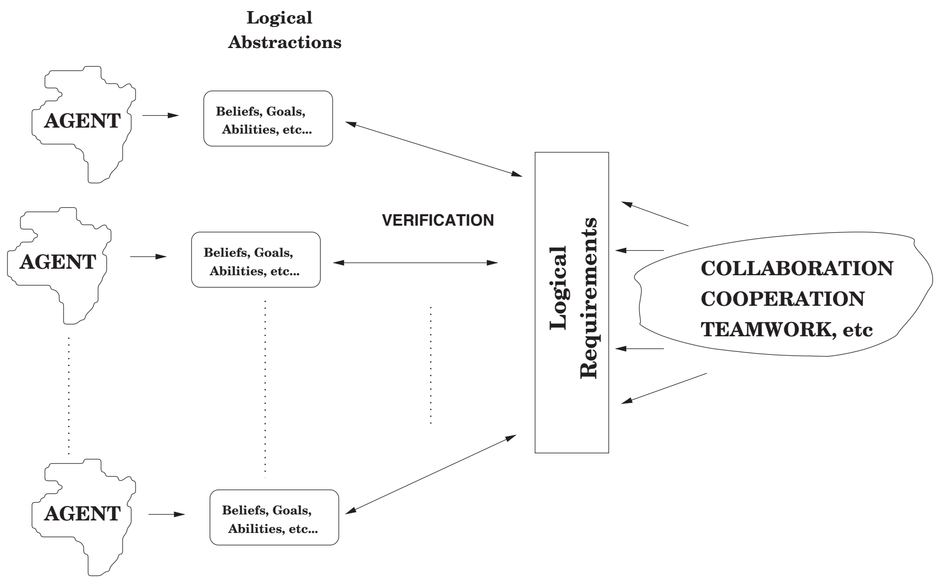


Figure 2: Verifying Systems of Multiple Rational Agents

As seen in the figure, we can combine the abstract representation of the various agents in the system and verify properties that refer not only to the properties that we require of each individual rational agent, but also to the properties that we require of the whole agent team. In particular, we can refer to its collaborative operation. Our particular approach to multi-agent verification is an instance of this general scheme.

Our Approach to Multi-Agent Verification — Our approach to verification of agent-based systems was first introduced in [1] (see also [2] for an overview of the approach). The idea is to do verification of agent programs directly, rather than a high-level design of the system. In particular, we have produced tools that can translate an agent-oriented programming language into the input language of the SPIN model checker as well as Java, thus allowing us to use NASA's Java Pathfinder as target model checker. We also defined a (relatively simple) logical language that allows us to write — using the types of abstractions typical of rational agents (such as beliefs, goals, etc.) — specifications of the properties the system is expected to satisfy. With this, the model checker can automatically verify whether the system satisfy the required properties or not.

Examples of properties we can verify are as follows. Imagine a scenario in which an astronaut is interacting with a robot on Mars. The robot is given some tasks for the day by the ground team, but the astronaut can interrupt the robot at any time and give alternative tasks. The original plan was for the robot to take panorama pictures at locations `l1` and `l2`, but the astronaut tells the robot to take a panorama at location `l3` instead. We may want to make sure that the robot will interrupt its original task as soon as the new course of action is suggested by the astronaut, that the requested panorama will be taken exactly at `l3` (i.e., the right location), that the original tasks are resumed once the astronaut instructs the robot to do that, that panoramas of locations `l1`, `l2`, and `l3` will all be eventually available, and so forth.

HUMAN-AGENT TEAMS

Human-agent teams refer to a complex human-agent work system in which people interact not only with each other, but also with software and hardware systems coordinated by rational software agents. This creates extra constraints, because even though software agents interact according to well-defined communication protocols, people do not. In this section we first discuss the reason why human-agent teams are relevant for space exploration. We then briefly describe the Brahms environment, developed at NASA Ames Research Center. Brahms was specifically developed for modelling human-machine interaction and work practice. Finally, we describe a typical human-agent teamwork scenario.

Why Human-Robot Teams in Space?

Although completely autonomous space missions (i.e., missions consisting solely of autonomous agents) are possible, they present a number of problems. One is that autonomous programs are difficult to control and analyse. An equally important one is that, for critical missions, human decision making is still essential at some level. Thus, there is a move towards combining the advantages of humans and agents into *human-agent teams*, specifically for planetary exploration. Crucially, human-agent (specifically, human-robot) teams are expected to carry out collaborative activity.

However, while such teams are appealing, in principle, several problems remain. The most significant are that such teams are difficult to program and analyse,

- individually, e.g. agents understanding what humans will do, and
- globally, e.g. programming and analysing teamwork and joint goals.

In spite of this, the future of space exploration (in particular planetary exploration) is likely to involve such human-agent teams. For example, NASA is planning to use human-robot teams in Mars exploration. Closer to home, we can see that the future of ubiquitous/pervasive computing essentially involves cooperation and collaboration amongst human-agent teams.

Brahms: Modelling Human-Agent Teams

Brahms is a multi-agent rule-based language developed at NYNEX (New York and New England Baby Bell Telephone Company, now Verizon), at the Institute for Research on Learning (IRL), and since 1998 at NASA Ames Research Center. The Brahms environment consists of a language definition, compiler, an integrated development environment (the Composer) and a Brahms Virtual Machine (the BVM) running on top of the Java virtual machine to load and execute Brahms models. Brahms was originally developed as a multi-agent language for modelling and simulating human work practice behaviour in organisations [6]. While Brahms can run in simulation mode, and is still used as a simulation environment [22], we have extended the BVM by allowing agents to run as real-time software agents without a simulation clock and event scheduler to synchronise the agents. This makes Brahms both a simulation and a software agent development environment. With Brahms you can test a multi-agent system by running the system as a simulation. When the system is debugged (using the Brahms AgentViewer), you can "flip the switch" and run the same system as a real-time distributed agent system. We refer to this as *from simulation to implementation*, a software engineering method that uses simulation as a system design and integration test environment.

Brahms agents are BDI-like agents (recall the Belief-Desire-Intention rational agent architecture mentioned earlier). However, Brahms does not use a goal-directed approach, but rather an approach we refer to as activity-based [5, 23]. Brahms agents are both deliberative and reactive. Each Brahms agent has a separate subsumption-based inference engine [3]. Brahms agents execute multiple activities at different levels at the same time. At each belief-event change (creation or changing of beliefs), situated-action rules (i.e., workframes) and production rules (called "thoughtframes") are evaluated at every active activity-level.

Brahms is a modelling language designed to model human activity. Agents, therefore, were developed to represent people. Brahms agents can belong to one or more group, inheriting attributes, initial beliefs, and activities, workframes and thoughtframes from multiple groups (multiple inheritance). This allows the abstraction of agent behaviour into one or more groups. Because Brahms was developed to represent people's activities in real-world context, Brahms also allows the representation of artifacts, data and concepts in the form of classes and objects. Both agents and objects can be located in a model of the world (the geography model) giving agents the ability to detect objects and other agents in the world and have beliefs about objects. Agents can move from one location in the world to another by executing a move activity, simulating the movement of people.

The Brahms modelling approach is based on a method that divides any system to be modelled into a number of more or less interdependent sub-models: the Agent, Object, Geography, Knowledge, Activity and Communication models. The Brahms model development environment — the Composer — supports this model-based approach, and allows the modeller to create groups and agents using a graphical user interface.

Group Hierarchy

The agent model consists of a group hierarchy representing the social, organisational, or functional groups of which agents are members. In the mission operations domain we can represent the mission operation workers according to their functional roles, such as the science team. Members of the science team are responsible for the science deliverables of the mission. They are often world-class scientists in specific domains, such as specialised science instruments that are carried onboard the robot. The science team members are divided into science theme groups that represent the functional roles during the mission, such as the "instrument synergy team", the "science operations team", and the "data analysis and interpretation team". Excerpt 1 shows the definition of some of the groups in Brahms source code (the excerpt shows partial source code: '`...`' means that source code is left out).

Excerpt 1. Partial Agent Model

```
group MyBasegroup memberof BaseGroup {
        attributes:
                public symbol groupMembership;
}

group VictoriaTeam memberof BaseGroup {...}

group ScienceTeam memberof VictoriaTeam, MyBaseGroup {
        location: Building244;

        attributes:
                ...
        initial_beliefs:
        // everyone knows where the rover is at the start of the sim
                (VictoriaRover.location = ShadowEdgeInCraterSN1);

        activities:
                ...
        workframes:
                ...
        thoughtframes:
}

group ScienceOperationsTeam memberof ScienceTeam {...}

agent Agent1 memberof ScienceOperationsTeam {
        initial_beliefs:
                (current.groupMembership = ScienceOperationsTeam);

        intial_facts:
                (current.groupMembership = ScienceOperationsTeam);
}
```

We will go step-by-step through the source code of Excerpt 1 explaining how groups and agents are defined. Note that this excerpt describes the definition of four groups and one agent. Bold font is used to denote keywords of the Brahms language. Every Brahms language element definition is actually placed in a separate source file, but is here shown as if it were part of one source code file for illustration purposes.

The first two groups are `MyBaseGroup` and `VictoriaTeam`. `MyBaseGroup` is a group defined by the modeller and is used to define common features for all groups. It is a non-domain specific "root" of the group hierarchy, used by the modeller to define common group properties. `MyBaseGroup` and `VictoriaTeam` are both members of the group `BaseGroup`, which is the root of all groups and is part of a base library that comes with the Brahms language, with certain predefined standard attributes. Here the `MyBaseGroup` group defines a common attribute for all groups, i.e., the `groupMembership` attribute. The `groupMembership` attribute is used in the model to allow agents to know to what group they belong. The third group that is defined is `ScienceTeam`. The group `ScienceTeam` is a member of two parent groups, `VictoriaTeam` and `MyBase`. This example shows that Brahms supports multiple inheritance for groups and agents. Group inheritance means that the subgroups and/or agents inherit all the elements defined in the parent group. The Brahms compiler will recognise naming conflicts in multiple inheritance and will report these at compile time. At this moment Brahms does not support "late-binding" and thus there are no possible inheritance conflicts at run-time. Next, the group `ScienceOperationsTeam` is defined as a member of the `ScienceTeam` group. Finally, we see the definition of an actual agent. The keyword **`agent`** declares agents, and in this example `Agent1` is an agent that is a member of the `ScienceOperationsTeam` group. Thus, the definition of groups and agents in Excerpt 1 explicitly defines the group hierarchy in Figure 3.

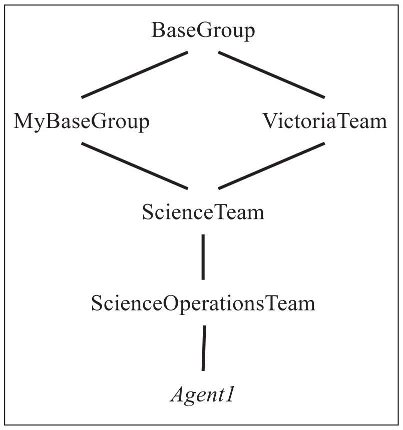


Figure 3: Group Hierarchy from Excerpt 1

Workframes

In Excerpt 2, two workframes are shown: a "high-level" workframe called `wf_SearchForWaterIce` (at the end of the excerpt), and a workframe part of the `FindingWaterIce` activity called `wf_WaitingForData`. Workframes allow for the execution of activities and the representation an agent's activity execution constraints. Since activities take time, a workframe has a duration based on the time that the activity takes. Workframes "fire" according to a pattern-matching process in which workframe preconditions are tested and workframe variables are bound. The body of a workframe (i.e., the *do*-part) can have conclude statements and activity calls. Conclude statements in workframes are meant to represent the belief-state of the agent in relation to the activity that is going to be executed (i.e., before the activity call) or has finished executing (i.e., after the activity call), and are not meant to represent reasoning of the agent (for this we use the thoughtframes).

One way of thinking about the role of workframes is to view them as constraints on when an agent can perform an activity. Workframe (WFR) `wf_SearchForWaterIce` constrains when the agent can perform the FindingWaterIce activity. The constraints are represented as the preconditions of the workframe. The preconditions encode what beliefs the agent needs to have in its belief-set to enable it to perform the activity or activities (there can be more than one activity call in the workframe body). In plain English `wf_SearchForWaterIce` says: "When I believe that the `VictoriaRover` is currently in the activity `SearchForWaterIce` and I believe that the `VictoriaRover` is currently located in a crater, first bind the name of the crater to the variable `rover-loc`, then execute the workframe body with priority zero" (Brahms allows for parallel execution of workframes, but uses a "time-sharing" approach using priorities). Note also that `wf_SearchForWaterIce` has the `repeat:false` statement at the top. This means that this workframe will only fire once for a particular set of beliefs that match all its preconditions. The result is that the agent will only execute `wf_SearchForWaterIce` once for any crater the `VictoriaRover` visits.

When the agent's inference engine has determined that the preconditions of `wf_SearchForWaterIce` are satisfied (due to finding matching beliefs in the agents belief-set) and it is the WFR with the highest priority, the agent will start executing the first statement in

Excerpt 2. Partial Activity Model for the ScienceOperationTeam Group

```
composite_activity FindingWaterIce (Crater crater, int pri) {
        priority: pri;

        activities:
                primitive_activity WaitingForData( ) {
                        priority: 0;
                        max_duration: 3600;
                } //end activity
                ...
        workframes:
                workframe wf_WaitingForData {
                        repeat: true;
                        priority: 0;
                        detectables:
                                detectable ReceiveHydrogenData {
                                 detect((VictoriaRover.nextSubActivity = DoDrilling))
                                 then abort;
                                } //end detectable
                                ...
                        when (knownval(current.nextSubActivity = WaitForData))
                        do {
                                WaitingForData( );
                        } //end do
                } //end workframe
                ...
        thoughtframes:
                ...
} //end composite_activity

workframe wf_SearchForWaterIce {
        repeat: false;
        variables:
                foreach(Crater) rover-loc

        when (knownval(VictoriaRover.currentActivity = SearchForWaterIce) and
              knownval(VictoriaRover.location = rover-loc))
        do {
              conclude((current.currentActivity = SearchForWaterIce));
              FindingWaterIce(rover-loc, 0);
        } //end do
} //end workframe
```

the body of the WFR, which in Excerpt 2 is the conclude statement that creates the belief for the agent that says that its current activity is `SearchForWaterIce`. This represents that the agent knows that it is currently in the activity of searching for water ice. Next, the engine calls the activity `FindingWaterIce`. Matching of beliefs preconditions, binding variables and firing the workframe, executing the conclude statement and calling the activity `SearchForWaterIce`, is all done in the same simulation time-event. Thus, although these processes take actual CPU time, they do not take any simulation time for the agent.

An important aspect of making Brahms into a software agent development language is the ability to seamlessly integrate Brahms agents within Java. Brahms has a JAPI defined to write agent activities in Java, so that they can be called from workframes. Brahms agents can also be completely written in Java, which enables the wrapping of existing external systems as a Brahms agent, enabling Brahms agents to communicate with external systems. For a more detailed description of the Brahms language we refer the reader to [21] and [4].

Sample Human-Agent Teamwork Scenario

There follows an outline of a scenario where human-agent (specifically, human-robot) teams can be used. This essentially concerns ensuring robust network connectivity for planetary exploration through the use of multi-agent (and human-agent) teamwork.

Two surface astronauts are going on an extra-vehicular activity (EVA) to explore a region, defined by the crew in an EVA plan. One or more network relays provide connectivity to each astronaut. A network relay can be one of two robots, or a robotically deployed relay device. The robots can be autonomous relays, as well as astronaut assistants, carrying tools and sample bags, taking pictures and panoramas. Each astronaut can "team up" with a robot. When a robot is teamed up with an astronaut, the robot's personal agent automatically performs the "following" and "watching" activities, and also automatically begins to monitor network connectivity to the astronaut (the astronaut's personal agent automatically begins to monitor connectivity to the robot). The personal agents also monitor network connectivity back to the habitat. The personal agent notifies the astronaut when network connectivity fails. Using a defined model of teamwork the robot and astronaut personal agents will provide simple recovery steps to try to reestablish communication.

VERIFYING HUMAN AGENT TEAMS

Semantics of Brahms Descriptions

The Brahms language is organised around the following representational constructs:

```
Groups of groups containing
        Agents who are located and have
              Beliefs that lead them to engage in
                  Activities specified by
                  Workframes
Workframes in turn consist of
        Preconditions of beliefs that lead to
              Actions, consisting of
                  Communication Actions
                  Movement actions
                  Primitive Actions
                  Other composite activities
              Consequences of new beliefs and facts
              Thoughtframes that consist of
```

Preconditions and
Consequences

Using Verification Techniques

Because of the success of the previous work in verifying AgentSpeak agents and multi-agent systems (written using AgentSpeak(F), a restricted version of AgentSpeak), and because Brahms, like AgentSpeak, extends the Belief-Desire-Intention notions of rational agency, it would seem reasonable to follow one or both of these approaches:

1. converting Brahms models into AgentSpeak models or
2. duplicating, with Brahms, as far as possible, the techniques used to make AgentSpeak verifiable.

The advantage of (1) is that it builds on existing work, and because it would involve translating one (essentially) BDI-based language into another, the task appears simpler. It would, however, require finding areas of common linguistic (syntactic/ontological) ground, which in turn could involve the simplification or re-formatting of the Brahms language, a step that might itself require significant formalisation and proof or verification.

A much larger task would be to convert Brahms models directly into a model checker's input language (e.g., Promela for Spin, Java for JPF). The complexity here centres on the fact that these input languages do not have BDI notions as part of their foundation and, for Spin, on the limitations of the Promela language. As this has been successfully done with programs written in AgentSpeak, such work might be used as an inspiration for doing similar translations with Brahms descriptions. Brahms, however, is more complex syntactically than AgentSpeak and, again, the translation process might require altering Brahms descriptions and would require a formal translation process (to ensure semantic equivalence between two quite different languages).

A pictorial view, following the previous description of multi-agent verification, of such human-agent teams is given in Figure 4.

Note. There is perhaps a subtle difference in the rationale behind using Brahms and using other agent languages. Brahms is fundamentally an application for modelling work systems, i.e., environments which include agents, objects, locations and other concepts and in which agents will typically collaborate. An agent language tends to focus more on the development and behaviour of the agents themselves, which may include competitiveness and may lack a framework of shared achievement.

In translating, the fundamental aim is to replicate the semantics of the source in the target language, and formal approaches often exist to make this as unambiguous as possible. In such a situation, the logic of the Brahms model would need to be described and this description would act as a pre-programming formal specification for the target language. Ideally, formal specifications will exist that were used in building the Brahms scenario in the first place and these may be a starting point for describing the underlying logic. As Brahms is modelling task-oriented behaviour

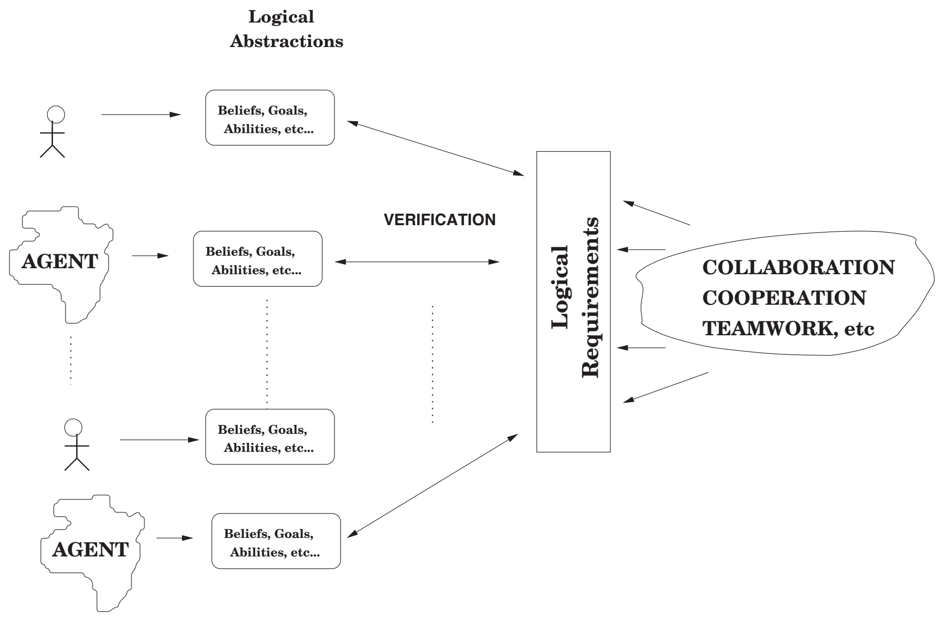

Logical Abstractions
Beliefs, Goals, Abilities, etc...
AGENT
Beliefs, Goals, Abilities, etc...
VERIFICATION
Beliefs, Goals, Abilities, etc...
AGENT
Beliefs, Goals, Abilities, etc...
Logical Requirements
COLLABORATION
COOPERATION
TEAMWORK, etc

Figure 4: Verifying Teams of Humans and Agents

in a dynamic, collaborative environment, many of the range of logics listed in the “Background” section may be applicable. In particular, BDI and temporal logics can be used to describe the agents’ mental states and associated activities. In Brahms, plans are represented as workframes, for example:

```
workframe wf_moveToRestaurant {
    repeat: true;
    variables:
        forone(Diner) dn;
        forone(Building) bd;

        when(knownval(current.howHungry > 20.00) and
        knownval(current.chosenDiner = dn) and
        not(current.location = dn.location) and
        knownval(dn.location = bd ))
        do {
        moveToLocation(bd);
        conclude((current.readyToLeaveRestaurant = false),
                 bc:100, fc:0);
        }
}
```

The `when` descriptor effectively describes the belief context and `do` section shows the actions taken. It also includes a `conclude` which describes the state at the end of the activity. It can be seen that a temporal logic formula can be produced to describe this workframe in the broad sense. For greater conformity to the AgentSpeak model, how desires and intentions are differentiated in Brahms and how plan and option generation and selection are performed, will need to be considered.

A further option is to examine the interpretation cycle of an AgentSpeak agent (e.g., given in the documentation for *Jason*, an interpreter for an extended version of AgentSpeak [9]), which is closely representative of a practical reasoning system, and again to compare that structure with a Brahms agent and to assess how the latter would need to be altered to give it some sort of equivalence to the AgentSpeak agent cycle. The cycle describes the process of perceiving the environment, updating beliefs, retrieving plans according to perceived events, selecting plans as intended means, and taking action. These are all actions that are present in the Brahms language semantics, but obviously in a different form and perhaps with different characteristics.

CONCLUDING REMARKS

Our approach to the formal verification of human-robot (and, in general, human-agent) teams consists of two parts:

1. the formalisation of team activity, using the Brahms framework, and
2. the use of our agent verification tools to analyse the team formalisations.

In particular, our aim is to analyse whether our approach to verification of BDI (Belief-Desire-Intention) agents, for example through the verification of AgentSpeak programs, can capture (simplified) Brahms descriptions. One direction is to consider the syntax of both AgentSpeak and Brahms in order to assess whether, or how, they can be matched. For example, consider the following excerpt from the grammars of both languages:

AgentSpeak **Brahms**

```
ag ::= bs ps        agent   ::=  agent
                                 agent-name {GRP.group-membership}
                                 {
                                  {GRP. attributes}
                                  {GRP. relations}
                                  {GRP. initial-beliefs}
                                  {GRP. initial-facts}
                                  {GRP. activities}
                                  {GRP. workframes}
                                  {GRP. thoughtframes}
                                 }
```

Comparing the languages semantically reveals conceptual differences in the way that models are constructed and may provide a basis for determining how, should it be necessary, Brahms descriptions can be simplified or re-formatted (inside or outside of Brahms) for the purposes of verification specifically.

In this paper, we described our approach to verifying Belief-Desire-Intention agent-based (space) applications using sophisticated model checkers, specifically by means of our model-checking techniques for the high-level AgentSpeak language. If successful, this provides a mechanism for verifying (at least part of) Brahms team activities.

There remain, of course, problems. Specifically, there is likely to be a complexity boundary separating viable agent verification from inviable ones. There also remain the problems of comprehensively (and accurately) describing team activity and representing the environment appropriately. Finally, there must be an appropriate logic language available for specifying the properties to be checked, which again can be inspired by our existing work on model checking for AgentSpeak.

Our aim is to develop this approach further, and to use realistic scenarios taken from NASA examples, in which it is essential to make sure that human-agent teams can always recover from error situations/problems. Also of interest are scenarios involving human intervention/override, such as advisory agents and semi-autonomous spacecraft docking.

REFERENCES

[1] R. H. Bordini, M. Fisher, C. Pardavila, and M. Wooldridge. Model checking AgentSpeak. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2003), Melbourne, Australia, 14–18 July*, 2003.

[2] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Model checking rational agents. *IEEE Intelligent Systems*, 19(5):46–52, September/October 2004.

[3] Brooks, R., A. Intelligence without representation. *Artificial Intelligence*, 47:139-159. 1991

[4] Brahms Website: `http://www.agentisolutions.com`, Ron van Hoof, 2000.

[5] W. J. Clancey. Simulating Activities: Relating Motives, Deliberation, and Attentive Coordination. *Cognitive Systems Research* 3(3):471-499. 2002.

[6] W. J. Clancey, P. Sachs, M. Sierhuis and R. van Hoof. Brahms: Simulating practice for work systems design. *International Journal on Human-Computer Studies* 49:831-865. 1998.

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[8] DART. `http://www.nasa.gov/mission_pages/dart/main/`.

[9] *Jason*. `http://jason.sf.net`.

[10] Java PathFinder. `http://javapathfinder.sf.net`.

[11] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J.L. White. Formal Analysis of the Remote Agent Before and After Flight. In *5th Langley Formal Methods Workshop*, June 2000.

[12] W. van der Hoek, B. van Linder, and J-J. Meyer. A Logic of Capabilities. Rapport IR-330, Vrije Universiteit, Amsterdam, July 1993.

[13] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, November 2003.

[14] B. van Linder. *Modal Logics for Rational Agents*. PhD thesis, Universiteit Utrecht, 1996.

[15] D. V. Pynadath and M.Tambe. Automated teamwork among heterogeneous software agents and humans. *Journal of Autonomous Agents and Multi-Agent Systems*, 7:71–100, 2003.

[16] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Second International Conference (KR'91)*, pages 473–484. Kaufmann, San Mateo, CA, 1991.

[17] A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.

[18] Remote Agent. `http://citeseer.ist.psu.edu/bernard98design.html`.

[19] N. Schurr, J. Marecki, P. Scerri, J.P. Lewis, and M. Tambe. The defacto system: Coordinating human-agent teams for the future of disaster response. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 8. Springer-Verlag, 2005.

[20] M. Sierhuis. Multiagent Modeling and Simulation in Human-Robot Mission Operations. (See `http://ic.arc.nasa.gov/ic/publications`).

[21] M. Sierhuis. *Modeling and Simulating Work Practice. BRAHMS: a multiagent modeling and simulation language for work system analysis and design.* PhD thesis, Social Science and Informatics (SWI), University of Amsterdam, SIKS Dissertation Series No. 2001-10, Amsterdam, The Netherlands, 2001.

[22] M. Sierhuis and W. J. Clancey. Modeling and Simulating Work Practice: A human-centered method for work systems design. *IEEE Intelligent Systems* 17(5) (Special Issue on Human-Centered Computing, 2002).

[23] L. A. Suchman. *Plans and Situated Action: The Problem of Human Machine Communication*. Cambridge, MA, Cambridge University Press, 1987.

[24] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *International Conference on Automated Software Engineering (ASE)*, September 2000.

[25] M. Wooldridge. *An Introduction to Multiagent Systems.* John Wiley & Sons, 2002.

[26] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents*. Springer-Verlag, 1995.

FORMALIZING INTEROPERABILITY FOR TEST CASE GENERATION PURPOSE

Alexandra Desmoulin and César Viho
IRISA/Université de Rennes 1
Campus de Beaulieu
35042 Rennes Cedex
France
{adesmoul, viho}@irisa.fr

ABSTRACT

This study deals with interoperability formal definitions and test derivation avoiding the state-space explosion problem. First, the notion of *interoperability criteria* is introduced. An interoperability criterion formally describes the conditions that two implementations must verify in order to be considered interoperable. The second point studied in this paper is interoperability test derivation. Based on the equivalence of two interoperability criteria, we proposed a method to derive automatically interoperability test cases.

1. INTRODUCTION

Different types of tests exist to ensure that implementations will work correctly in a real operational environment. Among these tests, conformance testing is used to verify if an implementation behaves as described in its specification, generally a standard. Another type of test is the interoperability test. Goals of interoperability are multiple. First, one has to test if the considered implementations communicate correctly. Secondly, they must behave during their interaction as described in their respective specifications. Third, they must provide the expected services.

Conformance testing is precisely characterized. Testing architectures and conformance relations [1,2] were defined leading to automatic test generation [3,4] and execution. This is not the case for interoperability testing. However, some attempts to give definitions of interoperability or methods to derive interoperability tests exists in [5,6,7]. In this paper, we give formal definitions of interoperability with *interoperability criteria* (*iop criteria* for short in the following) that give conditions to be verified by implementations to be considered interoperable. These iop criteria manage quiescence. Indeed, implementations are allowed to be quiescent if it is foreseen in their specification. Based on these criteria, we describe a method to generate automatically interoperability test cases which avoids the well-known state-space explosion problem.

This paper is structured as follows. First, Section 2 describes possible interoperability testing architectures. Section 3 presents the formal definitions used in this paper. The interoperability criteria are defined in Section 4. In Section 5, the proposed method and associated algorithms for interoperability test case generation are described. Results obtained are illustrated with an example in Section 6. Conclusion and future work are in Section 7.

2. TESTING ARCHITECTURE

This study considers a one-to-one interoperability context. The interoperability system under test (SUT) is composed of two implementations (see figure 1). These two IUT (Implementation Under Test) are supposed to behave as described in their respective specification. They communicate with each other while providing the expected service.

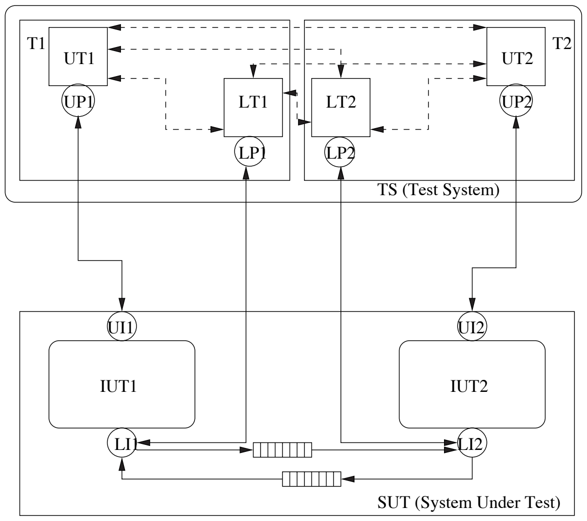


Figure 1. Test architecture for an asynchronous interaction

In this context, two kinds of interfaces can be differentiated. First, the interfaces LIi (lower interfaces) are used for the interaction between the two IUT (see figure 1). These interfaces are only observable but not controllable. Indeed, a test system connected to such interfaces can only observe the events, but it cannot send a stimulus to these interfaces. The lower tester LTi is in charge of the observation of LI_i via the lower PO (Point of Observation) LP_i.

The other interfaces are the interfaces UI_i (upper interfaces). These interfaces are not used for the interaction of the IUT but they are the interfaces through which the IUT communicate with its environment. These interfaces are observable and also controllable. The upper tester UTi is in charge of the control and observation of UIi via the upper PCO (Point of Control and Observation) UPi. Thus, the tester Ti, composed by UTi and LTi, is the part of the Test System (TS) in charge of the control and observation of IUTi.

Depending on the accessibility to the different interfaces, different interoperability testing architectures can be distinguished as described in [6,8]. The architecture is called lower (resp. upper) if only the lower (resp. the upper) interfaces are accessible, and total if both kind of interfaces are accessible. The interoperability testing architecture is called unilateral if only the interfaces of one of the two IUT are accessible. It is called bilateral if the interfaces of the two IUT are accessible but independently. The global architecture corresponds to the more usually considered case where all the interfaces of the two IUT are accessible with a global view.

Synchronous or asynchronous communication The interaction between the two IUT is asynchronous (cf. section 3.3). Notice also that the interaction between UP_i and the IUT can be either synchronous or asynchronous. It depends on the testing environment. We will consider that this latter is synchronous.

3. FORMAL BACKGROUND

In this study, we will use the well-known IOLTS (Input-Output Labelled Transition System) [9] to model specifications. As usual in the black-box testing context, we also need to model implementations, even though their behaviours are normally unknown. They will also be represented by an IOLTS.

3.1 IOLTS Model

Definition 1. *An IOLTS is a tuple* $M = (Q^M, \Sigma^M, \Delta^M, q_0^M)$ *where*
- Q^M *is the set of states of the system and* $q_0^M \in Q^M$ *is the initial state.*
- Σ^M *denotes the set of observable (input and/or output) events on the interaction points (with the environment) of the system.* $\Sigma^M \subseteq P^M \times \{?, !\} \times A^M$ *where* P^M *is the finite set of interaction points (ports) through which the system communicates with lower or upper layer, or other systems, "?" and "!" respectively denote an input and an output of message,* A^M *is the alphabet of input-output messages exchanged by the system through its ports.*
- $\Delta^M \subseteq Q^M \times (\Sigma^M \cup \{\tau\}) \times Q^M$ *is the transition relation, where* $\tau \notin A^M$ *denotes an internal event. We note* $q \xrightarrow{\alpha}_M q'$ *for* $(q, \alpha, q') \in \Delta^M$ *and* $q \not\xrightarrow{\alpha}$ *if there is no state* q' *such that* $(q, \alpha, q') \in \Delta^M$.

Σ^M can be decomposed as follow: $\Sigma^M = \Sigma^M_U \cup \Sigma^M_L$ (with $\Sigma^M_U \cap \Sigma^M_L = \emptyset$), where Σ^M_U (resp. Σ^M_L) is the set of messages exchanged on the upper (resp. lower) interface. Σ^M can also be decomposed in order to differentiate inputs from outputs: $\Sigma^M = \Sigma^M_O \cup \Sigma^M_I$ (with $\Sigma^M_O \cap \Sigma^M_I = \emptyset$), where Σ^M_O (resp. Σ^M_I) is the set of outputs (resp. inputs).

Let us consider an IOLTS M, and let $u_i \in \Sigma^M \cup \{\tau\}$, $\sigma \in (\Sigma^M)^*$, $q \in Q^M$, we have :
- $\Gamma(q)$ is the set of possible event from q, out(q) the set of outputs from q and in(q) the set of inputs from q.
- q *after* σ is the set of states which can be reached from q by the sequence of actions s . By extension, all the states reached from the initial state of the IOLTS M is (q_0^M *after* σ) and will be noted by (M *after* σ) . In the same manner, Out(M, σ) = out(M *after* σ) and In(M, σ) = in(M *after* σ) .
- Traces(q)= $\{\sigma \in (\Sigma^M)^* \mid q$ *after* $\sigma \neq \emptyset\}$ is the set of possible observable traces from q. And, Traces(M)= Traces(q_0^M) .
- $\bar{u}$=p!a if u=p?a and $\bar{u}$=p?a if u=p!a.

3.2 Quiescence, Suspensive IOLTS and Conformance Relation ioco

Three main situations lead to quiescence of a system : deadlocks, outputlocks and livelocks. A *deadlock* corresponds to a state after which no event is possible: $q \in$deadlock (M) = $\{\Gamma(q)= \emptyset\}$. An *outputlock* corresponds to a state after which only transitions labelled with input exist and none of these inputs is observed: $q \in$ outputlock(M) = $\{\Gamma(q) = In(q)\}$. A *livelock* corresponds to a loop of internal events: $q \in$livelock(M) = $\exists\ \tau, \ldots, \tau, (q,\{\tau, \ldots,\tau\},q) \in \Delta^M$. Thus, $q \in$quiescent(M) = $q \in$deadlock(M)$\cup q \in$outputlock(M) $\cup q \in$livelock(M) . A quiescence state $q \in$ quiescent(M) is modelled by (q, δ,q) where δ is treated as an observable output event. The obtained IOLTS is called suspensive IOLTS [2], is noted Δ(M) , and we have STraces(S)=Traces(Δ(S)) . Figure 2 gives an example of two specifications using the IOLTS model. Quiescence is modelled in the states 0 and 2 of S_1 , and in the state 0 of S_2.

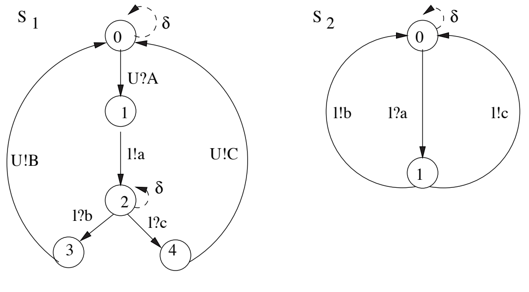


Figure 2. Specifications S_1 and S_2

Interoperability criteria defined in Section 4.2 are based on the ioco conformance relation [2]. This relation says that an implementation I is ioco-conformant with respect to its specification S if I can

never produce an output which could not be produced by S after the same suspension trace. Moreover, I may be quiescent only if S can do so. Formally:
I **ioco** S = $\forall \sigma \in STraces(S)$, $Out(\Delta(I), \sigma) \subseteq Out(\Delta(S), \sigma)$.

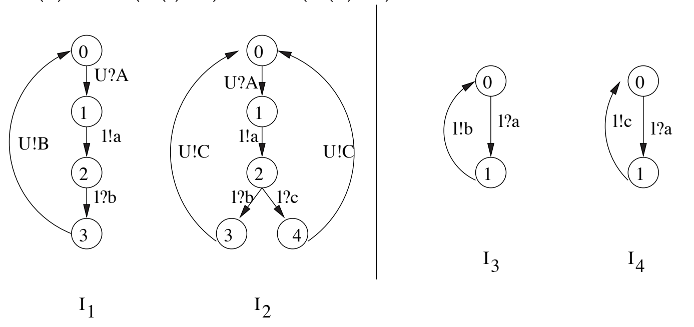


Figure 3. Implementations I_1 and I_2 of S_1, and I_3 and I_4 of S_2

Let us consider the implementation I_1 and I_2 of S_1 of figure 3: I_2 **ioco** S_1 (because of the output U! B after the reception of l?b) and I_1 **ioco** S_1 (because if the tester sends the message *c* on the lower interface of I_1, the implementation remains quiet, but no quiescence is foreseen in the state 4 of S_1). For the implementations I_3 and I_4 of S_2 of figure 3, we have I_3 **ioco** S_2 and I_4 **ioco** S_2.

3.3 Interaction

Interoperability testing generally deals with interactions of two or more implementations. To provide a formal definition of interoperability in a one-to-one context, we need to model the interaction of two IOLTS.

Definition 2 (Synchronous interaction $\|$). *The synchronous interaction of two IOLTS M_1 and M_2 is noted $M_1 \| M_2 = (Q^{M_1\|M_2}, \Sigma^{M_1\|M_2}, \Delta^{M_1\|M_2}, (q_0^{M_1}, q_0^{M_2}))$ with $Q^{M_1\|M_2} \subseteq Q^{M_1} \times Q^{M_2}$, $\Sigma^{M_1\|M_2} \subseteq \Sigma^{M_1} \cup \Sigma^{M_2}$. The transition relation $\Delta^{M_1\|M_2}$ is obtained as follows. $\forall (q_1, q_2) \in Q^{M_1} \times Q^{M_2}$,*

$$\frac{(q_1, a, q_1') \in \Delta^{M_1}, a \in \Sigma_U^{M_1} \cup \{\tau\}}{((q_1, q_2), a, (q_1', q_2)) \in \Delta^{M_1\|M_2}}, \frac{(q_2, a, q_2') \in \Delta^{M_2}, a \in \Sigma_U^{M_2} \cup \{\tau\}}{((q_1, q_2), a, (q_1, q_2')) \in \Delta^{M_1\|M_2}} \quad (1)$$

$$\frac{(q_1, a, q_1') \in \Delta^{M_1}, (q_2, \bar{a}, q_2') \in \Delta^{M_2}, a \in \Sigma_L^{M_1}, \bar{a} \in \Sigma_L^{M_2}}{((q_1, q_2), a, (q_1', q_2')) \in \Delta^{M_1\|M_2}} \quad (2)$$

There are different ways to obtain the model of the interaction of two IOLTS with quiescence management. The method chosen here is calculating first the suspensive IOLTS $\Delta(M_1)$ and $\Delta(M_2)$, as explained in section 3.2. This step is then followed by constructing the interaction of $\Delta(M_1)$ and $\Delta(M_2)$, using rules (1) and (2) of the definition 2. The main difficulty here is to preserve information that indicates the IUT in which quiescence is observed and to make appearing new quiescence introduced by the interaction. A quiescent state is noted: $((q_1,q_2),\delta(1),(q'_1,q'_2))$ if $(q_1, \delta,q'_1) \in \Delta(M_1)$, $(q_1,q_2),\delta(2),(q'_1, q'_2))$ if $(q_2,\delta,q'_2) \in \Delta(M_2)$, and $((q_1,q_2), \delta,(q'_1,q'_2))$ if $((q_1,q_2),\delta(1),(q'_1,q'_2))$ and $((q_1,q_2),\delta(2),(q'_1, q'_2))$. It is obtained by propagating δ of $\Delta(M_1)$ and $\Delta(M_2)$.

As, in the considered interoperability testing architecture, the interaction between the two implementations is asynchronous, we also need to model this asynchronous interaction. As in [10], we can model the asynchronous environment with FIFO queues. In [9], the asynchronous transformation $\mathcal{A}$ is defined. This transformation applied to a specification S gives as result the IOLTS $\mathcal{A}$(S) representing the behaviour of S in an asynchronous environment. As consequence, the asynchronous interaction of M_1 and M_2 corresponds to the synchronous interaction of $\mathcal{A}(M_1)$ and $\mathcal{A}(M_2)$, noted $M_1 \|_{\mathcal{A}} M_2$.

3.4 Projection

In interoperability testing, we usually need to observe some specific events of an IUT. The IUT, reduced to the expected messages, can be obtained by a projection of the IOLTS representing the whole behaviour of the implementation on a set (called X in the following). This latter is used to select the expected events. Quiescence d has to be seen in the projection as an observable event. For an IOLTS built from an interaction $M_1 \| _{\mathcal{A}} M_2$, quiescence $\delta(1)$ is an observable event of M_1 and $\delta(2)$ of M_2. The projection of an IOLTS M on the set of events X is noted by M/X and is obtained by hiding events (replacing by internal events) that do not belong to X, followed by determinization.

3.5 Modelling an Implementation for Interoperability Testing: The Iop-input Completion

As described in figure 1, the two IUT interact asynchronously and testers are connected to their interfaces. When an IUT sends a message m that cannot be treated by the other IUT, the problem is how to consider this message in the point of view of the receiver. Indeed, this message m is put in the input FIFO queue of the receiver that cannot effectively treat it. Thus, this receiving implementation may be quiescent. It can neither treat the message m in its input FIFO queue (l?m), nor it can do any other action because its input FIFO queue is not empty and no output is possible. To model this behaviour, we choose to complete any implementation with inputs corresponding to the output alphabet of the other IUT specification. These new transitions lead the IOLTS into an error state. It is a deadlock state. On the upper interfaces, the IUT interacts directly with the tester (like in a conformance testing context). Thus, for events on the upper interfaces, the input-completion of the IUT corresponds to the input completion made for conformance testing (see [10]).

Definition 3 (The iop-input completion).
Let us consider an IUT $I_1 = (Q, \Sigma, \Delta, q_0)$ *based on the specification* $S_1 = (Q_{S_1}, \Sigma_{S_1}, \Delta_{S_1}, q_0^{S_1})$ *interacting with an IUT based on the specification* $S_2 = (Q_{S_2}, \Sigma_{S_2}, \Delta_{S_2}, q_0^{S_2})$. *The iop-input completion of* I_1 *is* $C(I_1) = (Q_C, \Sigma_C, \Delta_C, q_0)$. $Q_C = Q \cup \{q_E, q_C\}$, q_E *represents the error trap state and* q_C *is the other input-completion state.* $\Sigma_C = \Sigma \cup \{\bar{a} | a \in \Sigma_O^{S_2} \cap \Sigma_L^{S_2}\}$. $\Delta_C = \Delta \cup \{(q, a, q_E) | q \in Q, \bar{a} \in \Sigma_O^{S_2} \cap \Sigma_L^{S_2}, q \overset{a}{\not\longrightarrow}\} \cup \{(q, a, q_C) | q \in Q, a \in \Sigma_I^{S_1} \cap \Sigma_U^{S_1}, q \overset{a}{\not\longrightarrow}\} \cup \{(q_C, x, q_C) | x \in \Sigma\}$.

Remark: The iop-input completion adds only transitions labelled with inputs to the original IOLTS representing the implementation. Thus, quiescence modelled in $C(I_1)$ or in I_1 is the same. To model the deadlock in the error state q_E, quiescence must be modelled in the iop-input completed implementation $C(I_1)$. Thus, $\Delta(C(I_1))$ is the model of the behaviour of I_1 in an asynchronous environment. In the following, the implementations are considered iop-input completed. Quiescence is also modeled on the considered implementations.

4. INTEROPERABILITY (IOP) CRITERIA

In this section, we define two iop criteria. These criteria formally describe conditions that have to be verified by two implementations to be considered interoperable. We prove their equivalence in terms of non-interoperability detection. These definitions only apply for compatible specifications. Indeed, two implementations cannot be interoperable if their specifications are not compatible.

4.1 Compatibility of the Considered Specifications

Two specifications are compatible if after any trace of the interaction, for each possible output on the interfaces used for the interaction, the corresponding input is foreseen in the other specification. In a formal way : $\forall \sigma \in Traces(S_1 \| _{\mathcal{A}} S_2)$, $\sigma/\Sigma^{S_1} = \sigma_1$, $\sigma/\Sigma^{S_2} = \sigma_2 \Rightarrow \{Out_{\Sigma L}(S_1, \sigma_1) \subseteq In_{\Sigma L}(S_2, \sigma_2)$ and $Out_{\Sigma L}(S_2, \sigma_2) \subseteq In_{\Sigma L}(S_1, \sigma_1)\}$. In the following, specifications are supposed to be compatible.

4.2 Definition of the Iop Criteria

In this section, we consider the global interoperability testing architecture (see Section 2). It is the most commonly used in practice for one-to-one interoperability testing. We define two iop criteria considering the events executed on the different interfaces of the implementations in two different ways.

The first iop criterion is the global iop criterion iop_G. It says that two implementations are considered interoperable if, after a suspensive trace of the asynchronous interaction of the specifications, all outputs and quiescence observed during the (asynchronous) interaction of the implementations are foreseen in the specifications.

Definition 4 (The global iop criterion iop_G). *Let I_1, $I_2 \in \mathcal{IOLTS}$ two IUT implementing respectively S_1, $S_2 \in \mathcal{IOLTS}$.*

$$I_1 \ iop_G \ I_2 =_\Delta \forall \sigma \in Traces(S_1 \|_A S_2), \ Out(I_1 \|_A I_2, \sigma) \subseteq Out(S_1 \|_A S_2, \sigma)$$

The other iop criterion defined in this section is the bilateral iop criterion iop_B. It says that after a suspensive trace of S_1 observed during the (asynchronous) interaction of the implementations, all outputs and quiescence observed in I_1 are foreseen in S_1, and the same in the point of view of I_2 implementing the specification S_2.

Definition 5 (The bilateral iop criterion iop_B). *Let I_1, $I_2 \in \mathcal{IOLTS}$ two IUT implementing respectively S_1, $S_2 \in \mathcal{IOLTS}$. $I_1 \ iop_B \ I_2 =_\Delta$*

$$\forall \sigma_1 \in Traces(\Delta(S_1)), \forall \sigma \in Traces(S_1 \|_A S_2), \ \sigma/\Sigma^{S_1} = \sigma_1 \Rightarrow$$
$$Out((I_1 \|_A I_2)/\Sigma^{S_1}, \sigma_1) \subseteq Out(\Delta(S_1), \sigma_1)$$
$$\text{and } \forall \sigma_2 \in Traces(\Delta(S_2)), \forall \sigma' \in Traces(S_2 \|_A S_1), \ \sigma/\Sigma^{S_2} = \sigma_2 \Rightarrow$$
$$Out((I_2 \|_A I_1)/\Sigma^{S_2}, \sigma_2) \subseteq Out(\Delta(S_2), \sigma_2).$$

As an example, with the implementations I1, I2, I3 and I4 of figure 3 (implementing S1 and S2 of figure 2), we have the results:

- I1 iop_B I3 and I1 iop_G I3 although *not*(I1 **ioco** S1).
- *not*(I2 iop_B I3) and *not*(I2 iop_G I3), but I2 iop_B I4 and I2 iop_G I4. Indeed, the output U!C is not allowed in S1 after l?b, but only I3 can send *b*, not I4.
- *not*(I1 iop_B I4) and *not*(I1 iop_G I4). Such a non-interoperability case would not have been detected without quiescence management. Indeed, the non-interoperability is due to the sending of message l2!c by I4 which is not expected by I1. Thus, this message is put in the input queue of I1 but not treated. The whole SUT is in a deadlock situation. This deadlock is not foreseen in the specification interaction. Thus the iop criteria are not verified due to non allowed quiescence.

4.3 Equivalence of the Two Interoperability Criteria

The most important result here is the following theorem 1. It says that the global iop criterion iop_G is equivalent to the the so-called bilateral iop criterion iop_B , in terms of non-interoperability detection. Its proof needs the lemmas defined in the following.

Theorem 1. *$I_1 \ iop_G \ I_2 \Leftrightarrow I_1 \ iop_B \ I_2$*

Lemma 1. *Let us consider $M_1, M_2 \in \mathcal{IOLTS}$, and let $\sigma \in Traces(M_1 \|_A M_2)$,*

$$Out(M_1 \|_A M_2, \sigma) = Out(\Delta(M_1), \sigma/\Sigma^{M_1}) \cup Out(\Delta(M_2), \sigma/\Sigma^{M_2}).$$

Proof. 1. Let $(q_1, q_2) \in [(M_1 \|_A M_2) after \sigma]$ and $a \in Out(M_1 \|_A M_2, \sigma)$. According to the interaction definition :
Either $a \in \Sigma^{M_1} \cup \{\delta, \delta(1)\}$, or $a \in \Sigma^{M_2} \cup \{\delta, \delta(2)\}$ ie. either $a \in Out(\Delta(M_1), \sigma/\Sigma^{M_1})$, or $a \in Out(\Delta(M_2), \sigma/\Sigma^{M_2})$.
$\Rightarrow Out(M_1 \|_A M_2, \sigma) \subseteq Out(\Delta(M_1), \sigma/\Sigma^{M_1}) \cup Out(\Delta(M_2), \sigma/\Sigma^{M_2})$.

2. In the other sense, it is easy to see that : $Out(M_1 \|_A M_2, \sigma) \subseteq Out(\Delta(M_1), \sigma/\Sigma^{M_1}) \cup Out(\Delta(M_2), \sigma/\Sigma^{M_2})$.

Lemma 2. *Let* $M_1, M_2 \in \mathcal{IOLTS}$.

$$((M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1})\|_{\mathcal{A}}((M_2\|_{\mathcal{A}}M_1)/\Sigma^{M_2}) = M_1\|_{\mathcal{A}}M_2$$

Proof. 1. Let $\sigma_1 \in Traces((M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1})$, $\sigma_2 \in Traces((M_2\|_{\mathcal{A}}M_1)/\Sigma^{M_2})$ and $\sigma = \sigma_1\|_{\mathcal{A}}\sigma_2 \in Traces(((M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1})\|_{\mathcal{A}}((M_2\|_{\mathcal{A}}M_1)/\Sigma^{M_2}))$.
We have : $\sigma_1 \in Traces(\Delta(M_1))$ and $\sigma_2 \in Traces(\Delta(M_2))$.
Thus, $\sigma = \sigma_1\|_{\mathcal{A}}\sigma_2 \in Traces(M_1\|_{\mathcal{A}}M_2)$.

2. Let $\sigma \in Traces(M_1\|_{\mathcal{A}}M_2)$ such that $\sigma = \sigma_1\|_{\mathcal{A}}\sigma_2$ with $\sigma_1 \in Traces(\Delta(M_1))$ and $\sigma_2 \in Traces(\Delta(M_2))$. We have $\sigma_1 = \sigma/\Sigma^{M_1}$ and $\sigma_2 = \sigma/\Sigma^{M_2}$.
Thus $\sigma_1 \in Traces((M_1\|_{\mathcal{A}}M_2/\Sigma^{M_1}))$, $\sigma_2 \in Traces((M_2\|_{\mathcal{A}}M_1/\Sigma^{M_2}))$ and $\sigma = \sigma_1\|_{\mathcal{A}}\sigma_2 \in Traces(((M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1})\|_{\mathcal{A}}((M_2\|_{\mathcal{A}}M_1)/\Sigma^{M_2}))$.

Lemma 3. *Let* $M_1, M_2 \in \mathcal{IOLTS}$, $\sigma_1 \in Traces(\Delta(M_1))$, $\sigma \in Traces(M_1\|_{\mathcal{A}}M_2)$ *and* $\sigma_1 = \sigma/\Sigma^{M_1}$. $Out((M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1}, \sigma_1) \subseteq Out(\Delta(M_1), \sigma_1)$.

Proof. $(M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1}$ is an IOLTS composed of events from $\Sigma^{(M_1\|_{\mathcal{A}}M_2)/\Sigma^{M_1}} \cup \{\delta\} \subseteq \Sigma^{M_1} \cup \{\delta\}$

Proof. of theorem 1. 1) Let us prove first that $I_1\, iop_B\, I_2 \Rightarrow I_1\, iop_G\, I_2$.
Let $\sigma \in Traces(S_1\|_{\mathcal{A}}S_2)$, $\sigma_1 \in Traces(\Delta(S_1))$ such that $\sigma_1 = \sigma/\Sigma^{S_1}$, $\sigma_2 \in Traces(\Delta(S_2))$ such that $\sigma_2 = \sigma/\Sigma^{S_2}$. Thus, $Out((I_1\|_{\mathcal{A}}I_2)/\Sigma^{S_1}, \sigma/\Sigma^{S_1}) \subseteq Out(\Delta(S_1), \sigma/\Sigma^{S_1})$ and $Out((I_2\|_{\mathcal{A}}I_1)/\Sigma^{S_2}, \sigma/\Sigma^{S_2}) \subseteq Out(\Delta(S_2), \sigma/\Sigma^{S_2})$.
Using the lemma 1, $Out[((I_1\|_{\mathcal{A}}I_2)/\Sigma^{S_1}\|_{\mathcal{A}}(I_2\|_{\mathcal{A}}I_1)/\Sigma^{S_2}), \sigma] \subseteq Out(S_1\|_{\mathcal{A}}S_2, \sigma)$.
With the lemma 2, $Out(I_1\|_{\mathcal{A}}I_2, \sigma) \subseteq Out(S_1\|_{\mathcal{A}}S_2, \sigma)$. That means $I_1\, iop_G\, I_2$.
2) Let us prove now that $I_1\, iop_G\, I_2 \Rightarrow I_1\, iop_B\, I_2$.
Let I_1, I_2, S_1, $S_2 \in \mathcal{IOLTS}$ such that $I_1\, iop_G\, I_2$. Let $\sigma_1 \in Traces(\Delta(S_1))$ such that $\sigma_1 = \sigma/\Sigma^{S_1}$ with $\sigma \in Traces(S_1\|_{\mathcal{A}}S_2)$. Using the definition of $I_1\, iop_G\, I_2$, we have : $Out(I_1\|_{\mathcal{A}}I_2, \sigma) \subseteq Out(S_1\|_{\mathcal{A}}S_2, \sigma)$. Projecting this inclusion on Σ^{S_1} gives $Out((I_1\|_{\mathcal{A}}I_2)/\Sigma^{S_1}, \sigma_1) \subseteq Out((S_1\|_{\mathcal{A}}S_2)/\Sigma^{S_1}, \sigma_1)$
Using the lemma 3, $Out((I_1\|_{\mathcal{A}}I_2)/\Sigma^{S_1}, \sigma_1) \subseteq Out(\Delta(S_1), \sigma_1)$. And using the fact that iop_G is symmetrical, we have also $I_1\, iop_G\, I_2 \Rightarrow Out((I_2\|_{\mathcal{A}}I_1)/\Sigma^{S_2}, \sigma_2) \subseteq Out(\Delta(S_2), \sigma_2)$. That means $I_1\, iop_G\, I_2 \Rightarrow I_1\, iop_B\, I_2$.

Based on the theorem 1, one may wonder how it can help interoperability test generation. This is the purpose of the study developed in the next section.

5. INTEROPERABILITY TEST GENERATION

In this section, we investigate the way to generate interoperability test using the equivalence between the bilateral and global criteria (cf. theorem 1).

5.1 General Principles for Interoperability Test Generation

The goal is to generate interoperability test cases (TC) that can be executable on the SUT. We consider a System Under Test (SUT) composed of two IUT interacting asynchronously (cf. figure 1 in Section2). These IUT are represented by a suspensive iop-input completed IOLTS. In practice, the inputs of a general interoperability test generation algorithm are the two specifications on which the implementations are based, and a test purpose (TP). A TP is a particular property (or behaviour in the interaction between the implementations) to be tested. In general, test purposes are incomplete sequences of actions. Let S be the set of specifications, P the set of test purposes and TC the set of interoperability test cases. The goal of a one-to-one interoperability test generation algorithm $\mathcal{G}$ is: $S{\times}S{\times}P{\rightarrow}TC$. Figure 4 (a) shows an example.
During conformance tests, a tester can send a stimulus to the implementation or receive an input. In the interoperability testing case, three kind of events are possible: sending of stimuli to the upper interfaces of the implementations, reception of inputs from these interfaces, but also observation of events (input and output) on the lower interfaces.

5.2 Interoperability Test Cases Modelling

A test case *TC* is represented by an extended version of IOLTS called *T-IOLTS* for Testing IOLTS. A T-IOLTS TC can be defined by TC =(Q^{TC}, Σ^{TC}, Δ^{TC}, q_0^{TC}). {PASS, FAIL, INC} $\subseteq Q^{TC}$ are trap states representing interoperability verdicts. q_0^{TC} is the initial state. $\Sigma^{TC} \subseteq \{u \mid \bar{u} \in \Sigma_U^{S1} \cup \Sigma_U^{S2}\} \cup \{?(u) \mid u \in \Sigma_L^{S1} \cup \Sigma_L^{S2}\}$. $?(u)$ denotes the observation of the message u on a lower interface. Δ^{TC} is the transition function.

In the following, any TC is supposed to be deterministic, and controllable (if a tester can do an output in a state, no other action is possible for the test case in this state). A TC must also be input and observation complete in the input and observation states : if an input or an observation is possible in a state, all other inputs and observations are possible in this state (generally denoted in test cases with ?otherwise label leading to FAIL). Moreover at least one of the verdict states (PASS, FAIL, or INC) is accessible from every state.

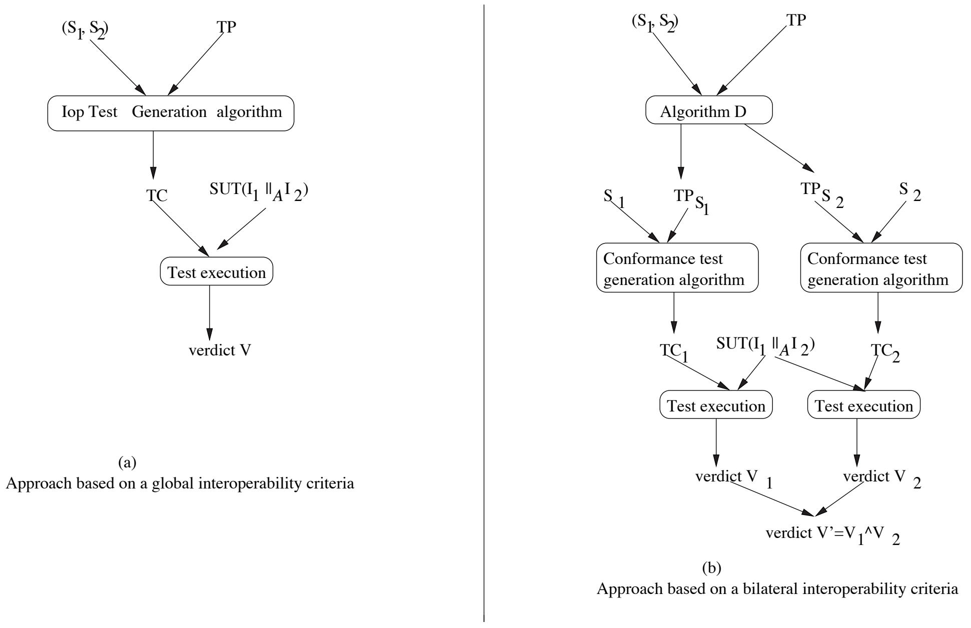

(S1, S2)
TP
Iop Test Generation algorithm
TC
SUT(I1 ||A I 2)
Test execution
verdict V
(a)
Approach based on a global interoperability criteria
(S1, S2)
TP
Algorithm D
S 1
TPS1
TPS 2
S 2
Conformance test generation algorithm
Conformance test generation algorithm
TC1
SUT(I1 ||A I 2)
TC2
Test execution
Test execution
verdict V 1
verdict V 2
verdict V'=V1^V 2
(b)
Approach based on a bilateral interoperability criteria

Figure 4. Interoperability Test Case Generation

The execution of the test case TC of the SUT (composed of the two considered IUT) gives a verdict: *verdict*(TC, SUT)∈{PASS, FAIL, INC}. The meanings of the possible interoperability verdicts are PASS: no error was detected during the tests, FAIL: the interoperability criterion is not verified and INC (for Inconclusive): the behaviour of the SUT seems valid but it is not the purpose of the test case.

5.3 Test Generation Based on the Global Iop Criteria

The construction of Test Cases based on the Global iop Criterion iop_G begins with the construction of the asynchronous interaction $S_1 \|_{\mathcal{A}} S_2$. Then $S_1 \|_{\mathcal{A}} S_2$ is composed with the test purpose TP. During this operation, two main results are calculated. First TP is validated. If the events composing TP are not found in the specifications (or not in the order described in TP), TP is not a valid Test Purpose. The composition is also used to keep (in the interaction of the two specifications) only the events concerned by the Test Purpose. It calculates the different ways to observe/execute TP on the SUT.

Problem: the construction of $S_1 \|_{\mathcal{A}} S_2$ can cause state-space explosion. Building $S_1 \|_{\mathcal{A}} S_2$ is exponential in the number of states of S1 and S2 and the FIFO queues size. Interoperability test derivation with this method may be impossible even for small specifications combined with "on-the-fly" techniques [3].

5.4 Using the Equivalence between Bilateral and Global Criteria

The theorem 1 of Section 4.3 proves that global and bilateral iop criteria are equivalent. We propose here a method to generate interoperability test cases that takes benefit from this result. This method uses conformance test tools.

Based on the bilateral iop criterion, the idea is to use a conformance test tool $\mathcal{F}$ such that $\mathcal{F}$: $(S_1, TP_{S1}) \rightarrow TC_1$ and $\mathcal{F}$: $(S_2, TP_{S2}) \rightarrow TC_2$. TP_{S1} and TP_{S2} are kind of "unilateral" test purposes derived from the test purpose TP. TP_{Si} is obtained from TP and contains only events of S_i.

In this context, the meaning of the theorem 1 is: *verdict* $(TC, I_1 ||_{\mathcal{A}} I_2)$= *verdict* $(TC_1, I_1 ||_{\mathcal{A}} I_2)$ *and verdict* $(TC_2, I_1 ||_{\mathcal{A}} I_2)$. The iop_G verdict *verdict* $(TC, I_1 ||_{\mathcal{A}} I_2)$ is an interoperability verdict with a global architecture. The two other verdicts are kinds of conformance verdicts. *verdict* $(TC_1, I_1 ||_{\mathcal{A}} I_2)$ (resp. *verdict*$(TC_2, I_1 ||_{\mathcal{A}} I_2)$) is the verdict obtained by executing TC_1 (resp. TC_2) unilaterally on interfaces of I_1 (resp. I_2) during its interaction with I_2 (resp. I_1). The rules for the combination of these two verdicts to obtain the final iop_B verdict are given by: PASS *and* PASS=PASS, PASS *and* INC=INC, PASS *and* FAIL=FAIL, INC *and* FAIL= FAIL, INC *and* INC=INC and FAIL *and* FAIL= FAIL .

Test generation based on the bilateral criterion iop_B

As described in figure 4 (b), the generation of TC_1 and TC_2 based on the bilateral criterion can be decomposed in two principal steps. First, step 1 (algorithm D) correspond to the derivation of TP_{S1} and TP_{S2} from TP . Then, step 2 is the calculation of TC_1 and TC_2. This step corresponds to the function $\mathcal{F}$ applied on (TP_{S1},S_1) and (TP_{S2}, S_2) and uses a conformance test tool.

For the execution of the test cases, TC_1 is applied on I_1 (during its interaction with I_2), and TC_2 on I_2 (during its interaction with I_1) leading to two verdict V1 and V2. The final interoperability verdict V'=V1 *and* V2 is obtained with the rules given above.

Step 1 : We will explain here how to obtain TP_{S1} from TP.

TP says that after the execution of some events $\mu_1 ... \mu_{n-1}$, the tester must observe another event μ_n, but does not explicit necessarily what may happen between μ_i and μ_{i+1}. In a formal context, TP is represented by an extended IOLTS. The most difficult problem to obtain TP_{S1} from TP is that $\mu_1 ... \mu_n$ may contain any events of both Σ^{S_1} and Σ^{S_2}. Thus, the algorithm to derive TP_{S1} from TP, S_1 and S_2 consists in separating events from S_1 (in TP_{S1}) and S_2 (in TP_{S2}) while keeping all information needed for the test generation.

If all the events described in TP are events on the lower interfaces, the algorithm to obtain TP_{S1} and TP_{S2} represented figure 5 is very simple. But if TP contains events on the upper interfaces, this algorithm needs to go through the IOLTS representing the specification S_2. It finds a path between μ_{i-1} (or μ_{i-1}) and μ_i. This operation is however simple and it costs less than calculating $S_1 ||_{\mathcal{A}} S_2$ with the method based on a global iop criterion (cf. figure 4 (a)).

This algorithm only verifies the existence of μ_i in the alphabet of the specifications. The verification of TP (thus the verification of TP_{S1} and TP_{S2} derived from TP) is done in step 2.

The algorithm of figure 5 uses some functions described hereafter. Let us consider a trace σ and an event a. The function remove_last_event is defined by: remove_last_event($\sigma.a$)= σ. And the function last_event by: last_event(σ)=ε if σ=ε (where $\varepsilon=\tau....\tau...\tau$) and last_event($\sigma$)=$a$ if $\sigma=\sigma_1.a$. The error function returns the cause of the error and exits the algorithm.

```
Input: $TP$: test purpose; Output: $\{TP_{S_l}\}_{l=1,2}$;
Invariant: $S_k = S_{3-l}$ (* $S_k$ is the other specification *); $TP = \mu_1 \ldots \mu_n$
Initialization: $\mu_0 = \epsilon$; $TP_{S_l} = \epsilon$;
for ($i = 0$; $i \leq n$; i++) do
    if ($\mu_i \in \Sigma^{S_l}$) then $TP_{S_l} = TP_{S_l}.\mu_i$ (* just add if it is an event of $S_l$ *)
    if ($\mu_i \in \Sigma_L^{S_k}$) then $TP_{S_l} = TP_{S_l}.\bar{\mu_i}$ (* just add the mirror if $\mu_i$ is
                                                        on the lower interface of $S_k$ *)
    if ($\mu_i \in \Sigma_U^{S_k} \cup \{\tau\}$)
        $\sigma_1 := TP_{S_l}$; $a_j$ =last_event($\sigma_1$)
        while $a_j \in \Sigma_U^{S_k} \cup \{\tau\}$ do
            $\sigma_1$=remove_last_event($\sigma_1$)
            $a_{j-1}$ =last_event($\sigma_1$) (* $a_{j-1}$ is the last event added to $TP_{S_l}$ and
                                              a mirror event $\bar{a}_{j-1}$ may exist in $S_k$ *)
        end
        $Ms_k = \{q \in Q^{S_k}$ such that q $\xrightarrow{\bar{a}_{j-1}}$ and $\sigma = \bar{a}_{j-1}.\omega.\mu_i \in Traces(q)\}$
        if ($\forall q \in Ms_k$, $q \overset{\sigma}{\not\longrightarrow}$) then error(TP not valid : no path in $S_k$ between the
        mirror event of last_event($TP_{S_l}$) and $\mu_i$)
        while (e=last_event($\omega$) $\notin \Sigma_L^{S_k} \cup \{\epsilon\}$) do $\omega$=remove_last_event($\omega$) end
        if (e $\in \Sigma_L^{S_k}$) then $TP_{S_l} = TP_{S_l}.\bar{e}$
    else error(TP not valid : $\mu_i \notin \Sigma^{S_1} \cup \Sigma^{S_2}$)
end
```

Figure 5. Algorithm to derive TP_{S_l} from TP

Step 2 : This step corresponds to the function $\mathcal{F}$ applied on (TP_{S1},S_1) and (TP_{S2}, S_2): (S_1, TP_{S1})$\rightarrow$$TC_1$ and (S_2, TP_{S2})$\rightarrow$$TC_2$. For the calculation of each test case (TC_1 and TC_2), the inputs are a specification (S_1) and a test purpose (TP_{S1}) based on this specification. We can use tools developed for conformance test generation like TGV [3] or TorX [4] for this step.
The most important difference consists in the access on the lower interfaces: these interfaces are observable and controllable in conformance testing but only observable in interoperability testing. Thus, the events on the lower interfaces described on the interoperability test cases obtained by $\mathcal{F}$ are only observed. The testers do not apply input on the lower interfaces. These inputs must come from the other implementation in interaction with the considered IUT. For example, if an event *l!m* exists in the test case obtained from conformance test generation tools (which means that the tester must send the message *m* to the lower interface of the IUT), this will correspond to *?(l?m)* in the interoperability test case. This means that the interoperability tester observes that a message *m* is received on the lower interface *l*. The events on the upper interfaces are controllable in both conformance and interoperability testing. Thus, no changes are made on the test cases for such events.

Few words about complexity
The first step of the method proposed here (cf. figure 4(b)) is linear in the maximum size of specifications. Indeed, it is a simple path search algorithm. The second step is also linear in complexity, at least when using TGV [3]. Thus, it costs less than calculating $S_1 \,||_{\mathcal{A}} S_2$ with the classical method based on a global iop criterion (cf. figure 4(a)). Moreover, if an iop test case can be obtained using the classical approach, the proposed method based on iop_B can also generate an equivalent bilateral iop test case.

6. APPLYING THE TEST GENERATION ALGORITHM TO AN EXAMPLE

Let us consider the two specifications S_1 and S_2 of figure 2 in Section 3.2. In the following, we show how the proposed algorithm can be used to derive interoperability tests. Two test purposes allow considering two significant situations that one may deal with.

First example: let us choose a test purpose *TP = l1!a.l2!b*
This example corresponds to the simplest case where all the events described in TP are events executed on the lower interfaces. When deriving TP_{S1} and TP_{S2} from TP ($\mu_1.\mu_2$=l1!a.l2!b in the algorithm), we obtain $TP_{S1}=\mu_1.\mu_2$=l1!a.l1?b and $TP_{S2}=\mu_1.\mu_2$=l2?a.l2!b. The obtained test cases TC_1 and TC_2 using TGV [3] are given in upper side of figure 6. (PASS) is a temporary verdict and PASS is the definitive verdict obtained after a postamble which returns to initial state, and the transitions labeled with *?otherwise* are not represented.

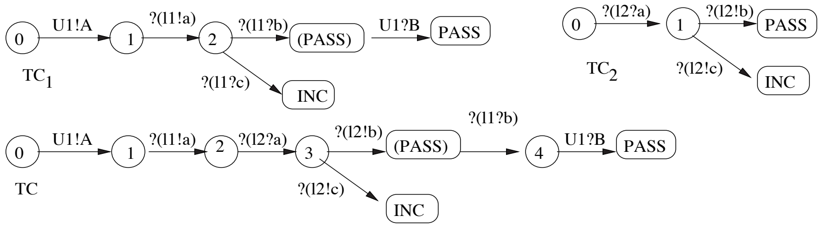


Figure 6. The obtained Test Cases from *TP=l1!a.l2!b*

For interoperability test case generation based on the global relation, the obtained TC (cf. figure 6) comes from the composition of $S_1 ||_{\mathcal{A}} S_2$ with TP. Thus, final interoperability verdicts obtained with TC_1 and TC_2, executed simultaneously or not on the SUT, must be the same as the verdict obtained with TC. The proof is not given here but a look at TC_1 and TC_2 shows that there are the same paths leading to the same verdicts as in TC.

Second example: let us now consider TP =U1? A. U1! B
This example is more complex than the previous one because TP contains only events on the upper interfaces of S_1. TP_{S1} is easy to derive from TP and TP_{S1}=TP. Deriving TP_{S2} from TP is more complex. Following the algorithm:

- μ_1=U1?A. Two possibilities, either ω=U1? A.l1!a.l1?b.U1!B or ω=U1?A.l1!a.l1?c.U1! C. Let us choose ω=U1?A.l1!a.l1?b.U1!B. So, last_event(ω)=U1!B $\notin \Sigma_L^{S_1}$ and ω= remove_last_event(ω)= U1?A.l1!a.l1?b. Next step, last_event(ω)=l1?b $\in \Sigma_L^{S_1}$, TP_{S2}=l2!b (if ω=U1?A.l1!a.l1?c.U1!C, TP_{S2}=l2!c).
- μ_2=U1! B: ω= l1!a.l1?b. Thus, last_event(ω)= l1?b $\in \Sigma_L^{S_1} \Rightarrow$ TP_{S2}=l2!b.l2!b.

Thus, we obtain $TP_{S1}=\mu_1.\mu_2$=U1?A. U1!B and TP_{S2}=l2!b.l2!b.

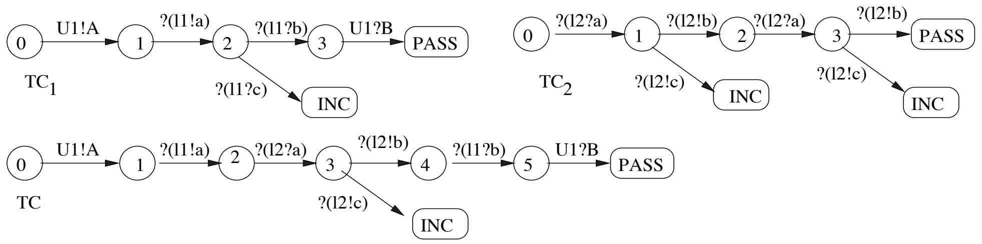


Figure 7.Test cases obtained for *TP=U1?A.U1!B*

The obtained test cases TC_1 and TC_2 are given in upper side of figure 7. The execution of TC_1 with TC_2 until state 2 of TC_2 corresponds to the same events as the execution of TC. The most difference is that TC_2 contains supplementary events to be executed: there is a loop that returns to initial state that comes from the search of the previous event of U1?A made to obtain TP_{S2}. Thus, verdicts obtained with TC_1 and TC_2 will be the same as the verdict that would be obtained with TC. But the calculation of TC needs the interaction of S_1 and S_2 whereas TC_1 and TC_2 are obtained using existing conformance test generation tools.

Some words on parallel test case execution. In the first example, TC_1 and TC_2 can be executed simultaneously because the derivation of TP_{S1} and TP_{S2} was simple. Indeed, the obtained test purposes contain only observations (no controllable events), TC_1 and TC_2 should be executed simultaneously with the tester T_1 observing and controlling IUT_1 and the lower tester LT_2 of T_2 observing IUT_2 (see figure 1).
In the second example, TC_1 and TC_2 can not be executed simultaneously. The most difference comes from the loop that returns to initial state in TC_2 (state 0 to state 2). There is no corresponding loop in TC_1. Thus, TC_2 is longer to execute than TC_1. TC_2 does not contain controllable events. Thus, the execution of this test case needs the application of a stimulus on I_1. I_1 can send a message on its lower interface to I_2. The observations are made on I_2 to verify TC_2.

7. CONCLUSION

In this paper, we propose formal interoperability definitions called iop criteria that give the conditions to be verified by two implementations in order to be considered interoperable. These two criteria (global iop criterion iop_G and bilateral iop criterion iop_B) are proved equivalent. This equivalence leads to a method to generate interoperability test cases which avoids the calculation of the specification interaction, and thus the state-space explosion problem.
Future work will study the generalization of these iop criteria to a context with more than two IUT. As it is suggested by the obtained test cases, we will also consider how a distributed approach can be applied for interoperability testing.

REFERENCES

[1] ISO. Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Parts 1-7. *International Standard ISO/IEC 9646/1-7*, 92.

[2] J. Tretmans. Testing concurrent systems: A formal approach. In J. C. M Baeten and S. Mauw, editors, *CONCUR'99– 10th Int. Conference on Concurrency Theory*, Lecture Notes in Computer Science, pages 46–65. Springer-Verlag, 99.

[3] J. C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming - Special Issue on Industrial Relevant Applications of Formal Analysis Techniques*, 29:123–146, 1997.

[4] J. Tretmans and E. Brinksma. Côte de Resyste–Automated Model Based Testing. In M. Schweizer, editor, *Progress 2002– 3rd Workshop on Embedded Systems*, pages 246–255, Utrecht, The Netherlands, Oct. 2002. STW Technology Foundation.

[5] Sébastien Barbin, Lénaïck Tanguy, and César Viho. Towards a formal framework for interoperability testing. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, pages 53–68, Cheju Island, Korea, August 2001.

[6] R. Castanet and O. Koné. Deriving coordinated testers for interoperability. In O. Rafiq, editor, *Protocol Test Systems*, volume VI C-19, pages 331–345, Pau-France, 94. IFIP, Elsevier Science B. V.

[7] Khaled El-Fakih, Vadim Trenkaev, Natalia Spitsyna, and Nina Yevtushenko. Fsm based interoperability testing methods for multi stimuli model. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2004.

[8] T. Walter, I. Schieferdecker, and J. Grabowski. Test architectures for distributed systems: state of the art and beyond. In Petrenko and Yevtushenko, editors, *Testing of Communicating Systems*, volume 11, pages 149–174. IFIP, Kap, Sep. 98.

[9] L. Verhaard, J. Tretmans, P. Kars, and E. Brinksma. On asynchronous testing. In G. V. Bochman, R. Dssouli, and A. Das, editors, *Fifth international workshop on protocol test systems*, pages 55–66, North-Holland, 93. IFIP Transactions.

[10] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. In J. Wu, S. Chanson, and Q. Gao, editors, *Formal methods for protocol engineering and distributed systems, FORTE XII/ PSTV XIX' 99, Beijing, China*, pages 25–40. Kluwer Academic Publishers, October 1999.

Encoding a process algebra using the Event B Method. Application to the validation of user interfaces

Yamine AIT-AMEUR, Mickael BARON and Nadjet KAMEL
LISI-ENSMA and University of Poitiers
BP 40109, 86961 Futuroscope Cedex, France
{yamine, baron, kamel}@ensma.fr
http://www.lisi.ensma.fr/ihm

ABSTRACT

This paper presents the use of the B technique in its event based definition. We show that it is possible to encode, using Event B, the models i.e. transition systems associated to a process algebra with asynchronous semantics. The Event B obtained encoding considers that the Event B model associated to the left hand side of a BNF rule defining the algebra expressions is refined by a model corresponding to the right hand side of the same rule. The translation rules of each operator of a basic process algebra are given. Then, an example illustrating each translation rule is given. This approach is based on a proof technique and therefore it does not suffer from the state number explosion problem occurring in classical model checking. The interest of this work is the capability to validate user tasks or scenarios when using a given system and particulary a critical system. Finally, we discuss the application of this approach for validating user interfaces tasks in the Human Computer Interaction (HCI) area.

Keywords. Event B method, events refinements, process algebra, application to HCI

INTRODUCTION

When performing syntactic analysis of formal languages, the classical approach consists in deriving abstract representations from a formal BNF grammar. Usually, these abstract representations are abstract syntax trees. The construction of an abstract syntax tree consists in applying the derivation rules of the BNF description. A hierarchical derivation tree is obtained. This tree may be built either ascending (bottom-up approach) or descending (top-down approach). The trees are augmented by other semantic information. Among this information, one can cite attributes, typing, or code generation routines and so on.

We claim that it is possible to associate to BNF grammars a hierarchy of Event B models in a top-down approach. The refinement relationship of Event B is used to encode the hierarchy provided by the abstract syntax tree. Each derivation rule is represented by a refinement and the obtained hierarchy describes a tree that is augmented by models. Moreover, these models are enriched by relevant properties (safety, reachability, robustness, ...). Obviously, the interest of such a transformation is to allow the possibility to perform proofs of these relevant properties associated to these models.

The particular case of a language of processes (a process algebra which looks like CCS) is shown in this paper. A BNF grammar defining the studied process algebra named CTT (ConcurTaskTrees [32]) is used as an example illustrating how our approach works. We give a translation rule for each BNF rule. This process algebra is used for the specification and validation of User Interfaces (UI). Indeed, validation of critical UI usually requires user scenarios which describe different usages of an UI. Nominal and non nominal scenarios are defined and checked on the designed UI. Moreover, these scenarios or tasks are checked at the specification and/or design level.

This paper is structured as follows. Next section recalls the basic definitions of the Event B method. Section 3 is the kernel of our proposal. It gives the principle of the translation of a BNF grammar to a hierarchy of Event B models. It also shows how this approach works on a BNF of the CTT process algebra and gives a practical example for each basic operator of this algebra. Section 4 describes the usage of this translation for validating UI. Section 5 discusses the interest of this approach in comparison with classical model checking.

THE EVENT B METHOD

Among the increasing number of formal methods that have been described, model oriented methods, such as VDM [15], Z [35] or B [2] [20], seem to have proved their applicability and efficiency. These methods are based on model description. They consist in defining a model by variable attributes which characterize the state of the described system, the invariants and other properties that must be satisfied and the different operations that alter these variables. Starting from this observation, Z method uses set theory notations and allows to encode the specifications in a structure named schema. Like VDM, it is based on preconditions and post-conditions [25, 27, 26]. Moreover, VDM allows the generation of a set of proof obligations. On the other hand, B is based on the weakest precondition technique of Dijkstra [21]. Starting from this method, J.R. Abrial [2] has defined a logical calculus, named the Generalized Substitution Calculus. Proof obligations are generated and need to be proved in order to ensure the correctness of developments and refinements.

In the recent years, J.R. Abrial has suggested a new definition of the B method: the Event B method [1]. This method is adapted to the development of interactive systems as well as sequential systems. Our choice of B is motivated by the fact that B Method is supported by tools which allow a complete formal development and is adapted to the description of interactive systems [17].

Event B models

The basic element of any development achieved with the Event B method is the *model*. A model is defined as a set of variables, defined in the **VARIABLES** clause that evolve thanks to events defined in the **EVENTS** clause. The notion of Event B model encode a state transition system where the variables represent the state and the events represent the transitions from one state to another. Moreover, the refinement capability offered by Event B allows to decompose a model (thus a transition system) into another transition system with more and more design decisions moving from an abstract level to a less abstract one. Refinement technique allows to preserve the proved properties and therefore it is not necessary to prove them again in the refined transition system (which is usually more complex). The structure of an Event B model is given by the following elements.

```
MODEL nameM
REFINES nameR
    ...
    VARIABLES ...
    INVARIANT ...
    ASSERTIONS ...
    INITIALISATION ...
    EVENTS ...
END
```

A model *nameM* is defined by a set of clauses. It may refine another model *nameR*. Briefly, the clauses mean:

- **VARIABLES** clause represents the variables of the model of the specification. Refinement may introduce new variables in order to enrich the described system.
- **INVARIANT** clause describes, thanks to first order logic expressions, the properties of the attributes defined in the clause VARIABLES. Typing information and safety properties are described in this clause. These properties shall remain true in the whole model and in further refinements. Invariants need to be preserved by the initialisation and events clauses.
- **ASSERTIONS** are logical expressions that can be proved from the invariants. They do not need to be proved for each event like for the invariant. Usually, they contain properties expressing that there is no deadlock nor livelock.
- **INITIALISATION** clause allows to give initial values to the variables of the corresponding clause. They define the initial states of the underlying transition system.
- **EVENTS** clause defines all the events that may occur in a given model. Each event is described by a body thanks to generalized substitutions defined below. Each event is characterized by its guard (i.e. a first order logic expression involving variables). An event is fired when its guard evaluates to true.

Semantics of generalized substitutions

The initialisation and the events occurring in a B model are described thanks to generalized substitutions. Generalized substitutions are based on the weakest precondition calculus of Dijkstra. Formally, several substitutions are defined in B. If we consider a substitution S and a predicate P representing a post-condition, then $[S]P$ represents the weakest precondition that establishes P after execution of S. The substitutions occurring in Event B models are inductively defined by the following expressions [1, 2, 29].

$$
\begin{aligned}
[\textbf{SKIP}]\,P &\iff P && (1)\\
[\textbf{S1} \mid\mid \textbf{S2}]\,P &\iff [S1]\,P \wedge [S2]\,P && (2)\\
[\textbf{ANY}v\textbf{WHERE}E\textbf{THEN}S\textbf{END}]\,P &\iff \forall\, v(P \Longrightarrow [S]\,P) && (3)\\
[\textbf{SELECT}E\textbf{THEN}S\textbf{END}]\,P &\iff E \Longrightarrow [S]P && (4)\\
[\textbf{BEGIN}S\textbf{END}]\,P &\iff [S]\,P && (5)\\
[\textbf{x:=E}]\,P &\iff P(x/E) && (6)
\end{aligned}
$$

$P(x/E)$ represents the predicate P where all the free occurrences of x are replaced by the expression E.

Substitutions 1, 2, 5 and 6 represent respectively the empty statement, the parallel substitution expressing that $S1$ and $S2$ are performed in parallel, the block substitution and the affectation. Substitutions 3 and 4 are the guarded substitutions where S is performed under the guard E.

In all the previous substitutions, the predicate E represents a guard. Each event guarded by a guard E is fired iff the guard is true and when it is fired, the post-condition P is established (feasibility of an event). The guards define the feasibility conditions given by the Fis predicate defined in [2].

Semantics of Event B models

The new aspect of the Event B method, in comparison with classical B, is related to the semantics. Indeed, the events of a model are atomic events. The associated semantics is an interleaving semantics.

Therefore, the semantics of an Event B model is trace based semantics with interleaving. A system is characterized by the set of licit traces corresponding to the fired events of the model which respects the described properties. The traces define a suite of states that may be observed by properties. All the properties will be expressed on these traces.

This approach has proved to be able to represent event based systems like interactive systems. Moreover, decomposition (thanks to refinement) allows building of complex systems gradually in an incremental manner by preserving the initial properties thanks to the gluing invariant preservation.

Refinement of Event B models

Each Event B model can be refined. A refined model is defined by adding new events, new variables and a gluing invariant. Each event of the abstract model is refined in the concrete model by adding new information by expressing how the new set of variables and the new events evolve. All the new events appearing in the refinement refine the $skip$ event of the refined model. Each new event corresponds to an $\epsilon-$transition in the abstract model.

The gluing invariant ensures that the properties expressed and proved at the abstract level (in the **ASSERTIONS** and **INVARIANTS** clauses) are preserved in the concrete level. Moreover, **INVARIANT**, **ASSERTIONS** and **VARIANT** clauses allow to express deadlock and livelock freeness.

1. They shall express that the new events of the concrete model are not fired infinitely (no livelock). *A decreasing variant is introduced for this purpose.*
2. They shall express that at any time an event can be fired (no deadlock). *This property is ensured by asserting (in the* **ASSERTIONS** *clause) that the disjunction of all the abstract events guards implies the disjunction of all the concrete events guards.*

Moreover, in the refinement, it is not needed to re-prove these properties again while the model complexity increases. Notice that this advantage is important if we compare this approach to classical model checking where the transition system describing the model is refined and enriched.

A full simple example [16]

Let us consider below, the specifications of the clock example[16]. The abstract specification $Clock$ uses one variable h describing the hours of the clock. Two events are described. The first ($incr$ event) allows to increment the hour variable. The second event is the $zero$ event. It is fired when $h = 23$ to initialize the hour variable.

```
MODEL
  $Clock$
VARIABLES
  $h$
INVARIANT
  $h \in 0..23$
ASSERTIONS
  $h < 100$
INITIALISATION
  $h := 13$
EVENTS
  $incr =$ SELECT $h \neq 23$ THEN $h := h + 1$ END;
  $zero =$ SELECT $h = 23$ THEN $h := 0$ END.
```

In the refinement specification ($ClockWMinute$) we introduce a new variable m and a new event $ticTac$. We enhance the guards of the $incr$ and $zero$ events in introducing the description of minutes. **ASSERTIONS** clause allows to ensure that the new events of the description system can be fired.

```
REFINEMENT
  $ClockWMinute$
REFINES
  $Clock$
VARIABLES
  $h, m$
INVARIANT
  $m \in 0..59$
ASSERTIONS
  $(h \neq 23) \vee (h = 23) \Rightarrow (h \neq 23 \wedge m = 59) \vee (h = 23 \wedge m = 59) \vee (m \neq 59)$
VARIANT
  $59 - m$
INITIALISATION
  $h := 13 \parallel m := 14$
EVENTS
  $incr =$ SELECT $h \neq 23 \wedge m = 59$ THEN $h := h + 1 \parallel m := 0$END;
  $zero =$ SELECT $h = 23 \wedge m = 59$ THEN $h := 0 \parallel m := 0$END;
  $ticTac =$ SELECT $m \neq 59$ THEN $m := m + 1$ END.
```

ENCODING CTT ALGEBRA IN EVENT B MODELS

This section is the kernel of our proposal. It presents the informal BNF translation rule and its application on a particular language describing a process algebra.

BNF rules translation principle

Our claim is that it is possible to parse BNF grammars into Event B models. The translation principle is defined as follows.

Each BNF rule of the form T ::= E OP F is translated into two Event B models. The first one is associated with the left hand side of the rule and contains only one event $eventT$ associated with the non terminal T. The second model is a refinement of the first one and corresponds to the right hand side of the BNF rule. Two new events $eventE$ and $eventF$ associated with the non terminals E and F are added in the refinement. These events carry the semantics of the op terminal and of the right hand side of the BNF rule. The new events are fired and when they are completed, the refined event $eventT$ is fired.

The firing order of the events is determined by introducing a decreasing variant. A variant is a natural number which decreases to zero. When the variant is zero, the events of the described refined model can no longer be fired again. Events of the abstract model can be fired, this corresponds to a return at the previous level in the decomposition tree. This possibility to return to the events of the abstract level is offered by the refinement relationship. In practice, this variant corresponds to a decreasing enumeration of states in a trace thanks to logical expressions. Remember that the events allow to go from an initial state to a target state defining a trace in the underlying described transition system.

In order to illustrate our approach, let us consider the CTT (ConcurTaskTrees) language which defines a classical process algebra. This language is widely used by the user interface community for specifying and/or validating user interfaces. User interfaces tasks are described thanks to this language. We have used this language to validate user tasks on user interfaces designs expressed with Event B. This point is discussed later in the paper in next section.

The task modelling language CTT

CTT [32] is defined by its authors as a notation for task model specifications to overcome limitations of notations used to design interactive applications. Its main purpose is to provide with an easy-to-use notation, which permits to describe tasks expressions combining CTT temporal operators and atomic tasks (atomic events). A CTT task model is based on a hierarchical structure of tasks represented by a tree-like structure. It requires identification of temporal relationships between other subtasks of the same tree level.

Below, a potential grammar describing the syntax of the CTT language is given. It presents temporal operators (from classical process algebra) and task characteristics of CTT.

T ::=	$T >> T$	- - Enabling
|	$T[]T$	- - Choice
|	$T\|\|T$	- - Concurrent
|	$T \models\| T$	- - Order independency
|	$[T]$	- - Optional process
|	$T[> T$	- - Disabling
|	$T\| > T$	- - Interruption
|	$T^{*}[> T$	- - Disabling infinite process
|	T^{N}	- - Finite process iteration
|	$T_{A}t$	- - Atomic process

Next sections, we show how the semantics of CTT can be formally described in Event B allowing to translate, in a generic manner, with generic translation rules, *every* CTT construction (interruption and disabling included) in Event B. This approach uses the refinement capability offered by Event B. For all the translation rules presented below, we will note var_i for the state variables, T_i for processes, G_i for event guards and S_i for any Event B generalized substitution. S_i corresponds to actions executed by a process T_i. It represents the captured semantics.

Generic rules for the translation of the basic CTT constructions

The rules for translating basic operators (*enabling*, *choice*, *iteration*, *concurrency* and *atomic process*) into Event B models are given below. They will be used to translate the remaining operators.

```
MODEL $T_0$
INVARIANT
  $I(var_i)$
INITIALISATION
  $Init(var_i)$
EVENTS
$Evt_0 =$
SELECT
  $G_0$
THEN
  $S_0$
END;
```

For the description of the transformation rules, we will use a process T_0 as the root process to be decomposed into another process expression corresponding to the CTT BNF given in above section. The set var_i describes all the useful state variables that characterize the process T_0. Other variables may be added after refinement if it is needed observe new elements while decomposing T_0 in the process tree. S_0 is the substitution that expresses, under the guard G_0, the state variables changes due to the process T_0. These elements are semantic features and are not represented in the syntax.

Basic illustrating example

The sum Sum of two natural numbers aa and bb will be used to illustrate how these rules work. We will give several different refinement possibilities to compute the sum of two natural numbers. The first Event B model Sum_{T0} corresponds to the instantiation of the previous generic model T_0. It contains the event Evt_0 whose corresponding guard (G_0) is equivalent to true (**BEGIN** . . . **END** generalized substitution).

```
MODEL $Sum_{T0}$
INVARIANT
  $Sum \in NAT \wedge aa \in NAT \wedge bb \in NAT$
INITIALISATION
  $Sum :\in NAT \parallel aa :\in NAT \parallel bb :\in NAT$
EVENTS
$Evt_0 =$
BEGIN
  $Sum := aa + bb$
END;
```

Here $aa :\in NAT$ means that aa becomes any natural number.

For the refinement of this example, we will use $RSum$, AA BB as the new variables of the refinement. They correspond to refinement variables of the abstract variables Sum, aa and bb respectively. These variables are linked by the same gluing invariant $RSum + AA + BB = aa + bb$ which guarantees the correctness of the refinement and therefore of the CTT operators encoding. The variable initialization $aa, AA :\in (aa \in NAT \wedge AA = aa)$ of the refinement ensures that the variables aa and AA are arbitrarily chosen natural numbers and are equal. The same applies for bb and BB.

Notice that the kernel of the *semantic part* of the translation consists in finding such an invariant. The other part, is related to the firing order of events.

Enabling ">>"

Let us consider $T_0 ::= T_1 >> T_2$ for the activation of T_1 followed by T_2 (sequence). The translation to Event B is given by a model with two events $EvtT_1$ and $EvtT_2$ corresponding to T_1 and T_2.

The translation uses a decreasing variant $StateEna$ initialized to 2. $EvtT_1$ is fired if its guard G_1 is true and so its substitution S_1 is performed, the variant decreases. $EvtT_2$ is fired in sequence if its guard is true and if the variant value is set to 1 by Evt_1.

```
REFINEMENT $RefEnabling_{T0}$
REFINES $T_0$
INVARIANT
  $J(var_i, var_j) \wedge StateEna \in \{0,1,2\}$
ASSERTIONS
  $G_0 \Rightarrow ((StateEna = 2 \wedge G_1) \vee (StateEna = 1 \wedge G_2) \vee \ldots$
VARIANT
  $StateEna$
INITIALISATION
  $StateEna := 2 \parallel \ldots$
EVENTS
$EvtT_1 =$                  $EvtT_2 =$                  $EvtT_0 =$
SELECT                      SELECT                      SELECT
  $StateEna = 2 \wedge G_1$     $StateEna = 1 \wedge G_2$     $StateEna = 0 \wedge G'_0$
THEN                        THEN                        THEN
  $StateEna := 1 \parallel S_1$   $StateEna := 0 \parallel S_2$   $S'_0$
END;                        END;                        END;
```

The disjunction of guards is given in the **ASSERTIONS** clause. The set var_j defines the state variables of the refinement. They are linked to the abstract state variables of the abstract model thanks to the gluing invariant $J(var_i, var_j)$. This gluing invariant is defined in all the refinements given below. Notice that the event $EvtT_0$ ends the enabling of the two processes, it gives the refinement of the event corresponding to process T_0.

Example of the use of the enabling operator translation - Let us consider that the sum of aa and bb is performed in a sequential manner. First the variable aa is added to $RSum$ by event $EvtT_1$ and then the variable bb is added to $RSum$ by event $EvtT_2$. These two events of the refinement work for $EvtT_0$ which collects the results and refines the event Evt_0 of the abstraction.

```
REFINEMENT $RefEnabling_{T0}$
REFINES $Sum_{T0}$
INVARIANT
  $(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
  $(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
  $AA + BB$
INITIALISATION
  $RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$
EVENTS
$Evt_1 =$                         $Evt_2 =$                         $Evt_0 =$
SELECT                            SELECT                            SELECT
  $AA \neq 0 \wedge BB \neq 0 \wedge \ldots$     $AA = 0 \wedge BB \neq 0 \wedge \ldots$     $AA = 0 \wedge BB = 0 \wedge \ldots$
THEN                              THEN                              THEN
  $RSum := RSum + AA \parallel$          $RSum := RSum + BB \parallel$          $Sum := RSum$
  $AA := 0$                         $BB := 0$                       END;
END;                              END;
```

Variables AA and BB are used to implicitly represent the variant $StateEna$. The **ASSERTIONS** clause ensures that the new events are fired and the variant guarantees the sequential ordering.

Choice ”[]”

Let us consider $T_0 ::= T_1 [] T_2$ defining a non deterministic choice between processes T_1 and T_2 i.e. either T_1 or T_2 is fired. The translation to Event B is given by a model with three guarded events $EvtT_1$, $EvtT_2$ and $Evt_{InitChoice}$.

The variant $StateCho$ is initialized to 3. According to the guard value of each event, one of Evt_1 or Evt_2 is fired. Each event decreases immediately the variant to value 0 forbidding the other events to be fired. The first event to be fired is arbitrarily chosen by the **ANY WHERE THEN** substitution. The refined event $EvtT_0$ ends the process T_0, it allows to fire again the event Evt_0 of the abstract model T_0.

```
REFINEMENT $RefChoice_{T0}$
REFINES $T_0$
INVARIANT
  $J(var_i, var_j) \wedge StateCho \in \{0, 1, 2, 3\}$
ASSERTIONS
  $G_0 \Rightarrow ((\exists(p).(p \in \{1, 2\} \wedge$
  $StateCho = 3)) \vee (G_1 \wedge StateCho = 1) \vee$
  $(G_2 \wedge StateCho = 2) \vee (StateCho = 0 \wedge G_0))$
VARIANT
  $StateCho$
INITIALISATION
  $StateCho :\in \{1, 2\} \parallel \ldots$
EVENTS
$EvtChoiceT_1 =$              $EvtChoiceT_2 =$              $EvtT_0 =$
SELECT                        SELECT                        SELECT
  $StateCho = 1 \wedge G_1$         $StateCho = 2 \wedge G_2$         $StateCho = 0 \wedge G'_0$
THEN                          THEN                          THEN
  $StateCho := 0 \parallel S_1$        $StateCho := 0 \parallel S_2$        $S'_0$
END;                          END;                          END;
```

Event Evt_0 ends the firing of the choice between two processes, it gives the refinement of the event corresponding to process T_0.

Example of the use of the choice operator translation - Let us consider that the sum of aa and bb is performed using a non deterministic choice. This possibility is offered by the semantics of Event B which allows a non deterministic event firing.

Two events are defined. One $EvtChoice_1$ computes the result $RSum = AA + BB$ and the second $EvtChoice_2$ computes the result $RSum = BB + AA$. These two events of the refinement are working for the event Evt_0 which collects the results and refines the event Evt_0 of the abstraction.

Here the variant is the natural number $AA + BB$. Notice that the non deterministic choice is performed thanks to the presence of the variant expression $AA \neq 0 \wedge BB \neq 0$ in the two events $EvtChoice_1$ and $EvtChoice_2$.

```
REFINEMENT $RefChoice_{T0}$
REFINES $Sum_{T0}$
INVARIANT
  $(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
  $(AA = 0 \wedge (BB = 0 \vee BB \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
  $AA + BB$
INITIALISATION
  $RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$
EVENTS
$Evt_0 =$                       $EvtChoice_1 =$                      $EvtChoice_2 =$
SELECT                          SELECT                               SELECT
  $AA = 0 \wedge BB = 0 \wedge \ldots$     $AA \neq 0 \wedge BB \neq 0 \wedge \ldots$     $AA \neq 0 \wedge BB \neq 0 \wedge \ldots$
THEN                            THEN                                 THEN
  $Sum := RSum$                   $RSum := AA + BB \parallel$            $RSum := BB + AA \parallel$
END;                              $AA := 0 \parallel BB := 0$            $BB := 0 \parallel AA := 0$
                                END;                                 END;
```

The $InitChoice$ event of the choice translation rule is not needed in this example. It is directly encoded in the **INITIALISATION** clause of the refinement.

Iterative process "T^*"

Let us consider a loop process $T_0 ::= T_1^N$. The principle of encoding a loop in Event B consists making possible to fire, N times, the events associated to the process T_1. A decreasing variant initialized to N is used. The translation into a loop encoded in Event B requires three events : a first one $Evt_{InitLoop}$ for initializing the variant $StateLoop$, a second one Evt_{Loop1} for the body of the loop and decreasing of the variant, and a third one Evt_0 for ending the loop and returning to the event of the abstract level.

```
REFINEMENT $RefIterative_{T0}$
REFINES $T_0$
INVARIANT
  $J(var_i, var_j) \wedge StateLoop \in NAT \wedge Start \in \{0, 1\}$
ASSERTIONS
  $G_0 \Rightarrow Start = 0 \vee (StateLoop > 0 \wedge G_1 \wedge Start = 1) \vee (StateLoop = 0 \wedge G_0' \wedge Start = 1)$
VARIANT
  $StateLoop + Start$
INITIALISATION
  $Start := 1 \parallel \ldots$
EVENTS
$Evt_{InitLoop} =$            $Evt_{Loop1} =$                                   $EvtT_0 =$
SELECT                        SELECT                                            SELECT
  $Start = 1$                   $G_1 \wedge StateLoop > 0 \wedge Start = 0$       $G_0' \wedge StateLoop = 0 \wedge Start = 0$
THEN                          THEN                                              THEN
  $StateLoop :\in NAT \parallel$  $StateLoop := StateLoop - 1 \parallel$          $S_0'$
  $Start := 0$                  $S_{Loop}$                                      END;
END;                          END;
```

The variant corresponding to the number of iteration is initialized by the $Evt_{InitLoop}$ event and then the Evt_{Loop1} event is fired $StateLoop$ times. When the loop terminates, $EvtT_0$ is fired. The variant $StateLoop$ decreases from its arbitrary initial value to 0. The **ASSERTIONS** clause states that one of the events guards is always true.

The initial value of the variant is arbitrary fixed thanks to the $:\in$ operator. The advantage of such an approach is the possibility to encode an arbitrary number of loop steps without increasing the complexity of the proof process. Compared to model checking techniques, increasing the number of loop steps may lead to the combinatorial explosion problem.

Example of the use of the iterative process translation - Let us consider the sum of two numbers obtained by performing the sum of aa and bb using a loop operator. The idea consists in first performing $RSum := AA$ and then adding, BB times, the value 1 to $RSum$. In this case, the variant will be the variable BB. Therefore, this example does not use an explicit variable $StateLoop$ to represent the variant.

```
REFINEMENT $RefIterative_{T0}$
REFINES $Sum_{T0}$
INVARIANT
  $(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
  $(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
  $AA + BB$
INITIALISATION
  $RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$
EVENTS
$EvtT_0 =$                     $Evt_{InitLoop} =$               $Evt_{Loop1} =$
SELECT                         SELECT                           SELECT
  $AA = 0 \wedge BB = 0 \wedge \ldots$     $AA \neq 0 \wedge BB \neq 0 \wedge \ldots$     $AA = 0 \wedge BB \neq 0 \wedge \ldots$
THEN                           THEN                             THEN
  $Sum := RSum$                  $RSum := RSum + AA \parallel$        $RSum := RSum + 1 \parallel$
END;                             $AA := 0$                        $BB := BB - 1$
                               END;                             END;
```

The previous Event B model uses three events. The event Evt_{Loop_1} decreases the variable BB until its value becomes 0. When the variant equals to 0, $EvtT_0$ is fired to return to the events of the abstraction.

Concurrency "||"

Let us consider $T_0 ::= T_1 || T_2$. The semantics of concurrency in interleaving semantics imposes to describe all the possible behaviors. Therefore, this process is described by all the possible traces. It uses the interleaving underlying Event B semantics i.e. if two events have their guard to true, they are fired in parallel, in an interleaving manner.

Two events $EvtT_1$ and $EvtT_2$ corresponding to processes T_1 and T_2 are defined. They can be fired at any time.

```
REFINEMENT $RefConcurrency_{T0}$
REFINES $T_0$
INVARIANT
  $J(var_i, var_j) \wedge StateConc_1 \in \{0,1\} \wedge StateConc_2 \in \{0,1\}$
ASSERTIONS
  $G_0 \Rightarrow ((G_1 \wedge StateConc_1 = 1) \vee (G_2 \wedge StateConc_2 = 1) \vee \ldots$
VARIANT
  $StateConc_1 + StateConc_2$
INITIALISATION
  $StateConc_1 :\in NAT \parallel StateConc_2 :\in NAT1 \parallel \ldots$
EVENTS
$Evt_1 =$                      $Evt_2 =$                      $Evt_0 =$
SELECT                         SELECT                         SELECT
  $G_1 \wedge StateConc_1 = 1$      $G_2 \wedge StateConc_2 = 1$      $G'_0 \wedge StateConc_1 = 0 \wedge StateConc_2 = 0$
THEN                           THEN                           THEN
  $StateConc_1 = 0 \parallel S_1$     $StateConc_2 = 0 \parallel S_2$     $S'_0$
END;                           END;                           END;
```

Once the events $EvtT_1$ and $EvtT_2$ are fired, they cannot be fired again. Their variant ($StateConc_1$ and $StateConc_2$ respectively) is decreased from any natural number ($:\in$) to 0 and the abstract event of the refined model can be fired again (Evt_0).

Example of the use of the concurrency operator translation - For the same example, we have imagined a refinement with a concurrent operator. Indeed, two concurrent events are defined. $EvtT_1$ adds the value of AA to $RSum$ when the event $EvtT_2$ adds the value BB to $RSum$. When these two events are fired, the last event $EvtT_0$ is fired to compute the final result for the abstraction.

```
REFINEMENT $RefConcurrency_{T0}$
REFINES $Sum_{T0}$
INVARIANT
  $(RSum + AA + BB) = (aa + bb) \wedge \ldots$
ASSERTIONS
  $(AA = 0 \wedge BB = 0 \wedge aa + bb \in NAT) \vee \ldots$
VARIANT
  $AA + BB$
INITIALISATION
  $RSum := 0 \parallel Sum :\in NAT \parallel aa, AA :\in (aa \in NAT \wedge AA = aa) \parallel \ldots$

EVENTS
$EvtT_0 =$
SELECT
  $AA = 0 \wedge BB = 0 \wedge \ldots$
THEN
  $Sum := RSum$
END;

$EvtT_1 =$
SELECT
  $AA \neq 0 \wedge \ldots$
THEN
  $RSum := RSum + AA \parallel$
  $AA := 0$
END;

$EvtT_2 =$
SELECT
  $BB \neq 0 \wedge \ldots$
THEN
  $RSum := RSum + BB \parallel$
  $BB := 0$
END;
```

The variant is the sum of variables AA and BB. However, if the model requires not to change AA nor BB then, the developer could have used explicit variants.

Translation of other CTT constructions

Up to now, $>>$, $||$, $[\,]$ and $*$ operators have been described in Event B models. All the other CTT constructions (*order independency*, *optional process*, *disabling*, *interruption*) are described below. Thanks to the interleaving semantics of the Event B method, the translation of these operators uses the basic operators defined previously.

Order independency "$|{=}|$"

Let us consider $T_0 ::= T_1 \;|{=}|\; T_2$. In trace based semantics, order independency is interpreted by:

$$T_0 ::= T_1 \;|{=}|\; T_2 \; is\, translated\, to\, T_0 ::= (T_1 >> T_2)\, [\,]\, (T_2 >> T_1) \tag{7}$$

The enabling and choice basic operators are used for this translation. They define the interleaved traces encoding the order independency operator.

Optionality "$[Task]$"

Let us consider $T_0 ::= [T_1]$. The optional process indicates that process T_1 may be accomplished. In this case, we introduce the atomic empty process, namely T_{Skip}. Then, optionality is translated to a choice between the process T_1 or the empty process. We get:

$$T_0 ::= [T_1] \;\; is\, translated\, to\, T_0 ::= T_1\, [\,]\, T_{Skip} \tag{8}$$

The generalized substitution of the process T_{Skip} is simply $skip$.

Disabling ”[>”

Let us consider $T_0 ::= T_1[> T_2$ where T_1 is disabled by T_2. Disabling requires a case based processing: either T_1 is atomic (i.e. cannot be refined or decomposed) or not.

Indeed, if T_1 is an atomic process, then it means that either T_1 is performed or T_2 is performed. We get a translation by a choice operator:

$$T_0 ::= T_1[> T_2 \; is\; translated\; to\; T_0 ::= T_1 \,[]\, T_2 \tag{9}$$

When T_1 is not an atomic process (i.e. it involves CTT operators defining observable states) disabling can occur in the trace defined by T_1. If T_1 is decomposed into $T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$, then the disabling translation is defined by the choice of all the possible traces resulting from the disabling i.e. disabling can occur at each observable state of the T_1 decomposition and shall be propagated in the further decompositions. We get:

$$\begin{aligned} T_0 ::= T_{1,1}\ op_1\ T_{1,2}\ op_2 \quad & \cdots\ op_n\ T_{1,n+1}[> T_2 \; is\; translated\; to \\ & T_0 ::= T_2 \\ & []\, T_{1,1} >> T_2 \\ & []\, T_{1,1} op_1 T_{1,2} >> T_2 \\ & []\, T_{1,1} op_1 T_{1,2} \cdots op_i T_{1,i+1} >> T_2 \\ & []\, \cdots >> T_2 \\ & []\, T_{1,1} op_1 T_{1,2} \cdots op_n T_{1,n+1} \end{aligned} \tag{10}$$

Remark. When disabling occurs, T_1 is stopped. The system may be in a corrupted state which may correspond to a bad state of the system.

When we use the Event B method to encode the disabling operator, it may happen that the proof obligations cannot be proved. Indeed, since a state is corrupted it does not correspond to a correct process decomposition and the gluing invariant is not preserved by the refinement which encodes the disabling. The use of Event B allows to generate proof obligations that cannot be proved. In this case, a repairing event can be added in the disabling decomposition. The repairing event will artificially repair the system in order to return to a non corrupted state. This event enforces the preservation of the gluing invariant and provides for a formal debugging.

Let us consider a small decomposition of $T_0 ::= T_1[> T_2$ where T_1 is not an atomic process. We give below a partial refinement specification of the T_0 process.

```
REFINEMENT
  $DisablingRef$
INITIALISATION
  $DisablingState := 3 \parallel \ldots$

EVENTS
$EvtT_{11} =$                          $EvtT_{12} =$                              $Evt_{Disabling} =$
SELECT                                 SELECT                                     ANY $pp$
  $G_{11} \wedge DisablingState = 2$     $G_{12} \wedge DisablingState = 1$         WHERE
THEN                                   THEN                                         $pp \in 1, 2 \wedge$
  $S_{11}$                               $S_{12} \parallel DisablingState := 0$       $\neg DisablingState = 0$
END;                                   END;                                       THEN
                                                                                    $DisablingState := pp$
                                                                                  END;
$EvtT_1 =$
SELECT
  $G_1 \wedge DisablingState = 0$
THEN
  $S_1$
END;
```

Disabling infinite loop process " $^*[>$"

Let us consider $T_0 ::= T_1^*[> T_2$. In this case, the translation uses the same reasoning as the previous classical disabling. We use the previous translation to define intermediate disabling in each iteration. We get:

$$\begin{gathered} T_0 ::= T_1^*[> T_2 \; is \; translated \; to \; T_0 ::= (T_1)^N [> T_2 \\ where \; N \; is \; any \; arbitrary \; natural \; number \end{gathered} \tag{11}$$

Notice that the Event B method allows to define such an arbitrary natural number and to perform proofs on this basis. This capability represents one advantage on model checking techniques where such a reasoning is not possible.

Finally, if the process T_1 is not atomic, then disabling can occur in each observable state of the T_1 decomposition as defined in disabling operator subsection.

Interruption "|>"

Let us consider $T_0 ::= T_1| > T_2$. The process T_1 is interrupted by the process T_2. As for the disabling operator, interruption translation requires a case based reasoning.

If T_1 is an atomic process (not involving CTT operators), then it means that T_2 can be performed an arbitrary number of times (may be 0) and then T_1 is performed. In this case, we get:

$$T_0 ::= T_1| > T_2 \; is \; translated \; to \; T_0 ::= T_2^N >> T_1 \tag{12}$$

When T_1 is not an atomic process involving CTT operators the interruption can occur many times in each observable state of the trace defined by T_1. If T_1 is written as $T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}$, then the interruption translation is defined by the choice off all the possible traces resulting from the interruption. We get:

$$\begin{gathered} T_0 ::= T_{1,1}\ op_1\ T_{1,2}\ op_2 \cdots\ op_n\ T_{1,n+1}| > T_2 \\ is \; translated \; to \\ T_0 :: T_2^{N_1} >> T_{1,1} >> T_2^{N_2} op_1 T_{1,2} \cdots >> T_2^{N_{n+1}} op_n T_{1,n+1} \end{gathered} \tag{13}$$

Here N_i are arbitrary natural numbers showing that the process T_2 associated to interruption can occur zero or several times. Notice that we have chosen to interpret the interruption operator using this approach. One could have chosen to activate interruption exactly once ($N_i = 1$).

APPLICATION FOR THE SPECIFICATION AND VALIDATION OF UI

One of the major aspects, in the User Interface (UI) development activity, is the capability to take into account usability of the UI. In general, usability is captured a posteriori through experimentation and/or a priori through the description of a set of tasks representing scenarios of use. The validation of UI in critical systems like plane cockpits or machine factory interfaces use such scenarios.

The work we have performed with the previously described approach deals with the latter. We have represented tasks by decomposable processes. We consider that a set of tasks is described using a user task notation and the designed UI shall meet the requirements expressed within these tasks.

Notations and models for design and validation of HCI

In general, the development of UI is concerned by two important interleaving phases.

1. A *design phase* which allows to produce the code implementing the suited UI and its link with the functional core (heart of the application). In this phase, architectural notations, verification, validation, specification, refinement and programming techniques are used by UI developers. Among the design notations and techniques we can cite the Seeheim model [33], PAC[18], ARCH[14, 13], hybrid models [24, 22] and so on. Usually this phase allows to establish *robustness properties*.

2. A *task validation phase* which consists in validating user needs. This phase is not well mastered by UI designers since most of these validations are issued from non computer scientists like psychologists and ergonomists. A set of tasks, defining scenarios, is described at the requirements level and shall be supported by the final UI product. Among the description oriented techniques and notations one can cite MAD [34], XUAN [23] and CTT [31]. This list is not exhaustive and may be completed. Usually this phase allows to establish *validation properties*.

Unfortunately, the different models and notations we outlined above do not give enough formal representation to allow the complete validation and verification, at the specification and design levels, of the UI design with respect to given properties and user tasks. In general, verification and validation are reported at the testing phase when all the software development is completed.

Therefore, representing formally both design and tasks, at early stages of the development (abstract levels) will permit such verification and validation. However, thanks to the possibility of describing formal abstract models provided by formal techniques, it becomes possible to handle a large amount of validation and verification efforts early at the specification and design phases.

The proof based techniques we are using help to increase the quality of UI software developments. Indeed, our approach uses the Event B formal technique for representing, verifying and refining specifications [10, 5] [3] and [4]. In [5, 6], we presented our approach, based on Event B, handling the design phase.

Applications to Human Computer Interaction area

User requirements validation is performed by way of the validation of a set of user tasks. A task can be seen as a scenario of use of the interface. It can be validated using several techniques: running a prototype, simulating or animating a model or model checking or proof techniques.

For validating user interfaces tasks, we have used the CTT (ConcurTaskTrees) process algebra and its representation in Event B previously presented. Two main applications have been developed : one consists in validating WIMP interfaces (Windows, Icons, Mouse and Pointers) and the second one deals with multi-modal interfaces. Before describing these two applications, we overview the way we have represented the dialog controller of an UI which represents the kernel of any UI.

Encoding the dialog controller with Event B

A set of B models is described. They allow to encode all the parts of an UI software from the functional core of the application to the toolkit, presentation and dialog controller.

The dialog controller contains all the events that can be fired by an user while using the interface. These events are *atomic*. In their turn, these events fire other events from the presentation, toolkit or from the functional core. The guards of these events define their firing order and how these events interleave.

We do not give the details of this approach for the description of the UI in this paper but more details can be found in [10, 5, 4, 3] and [8].

Application to the WIMP user interfaces[7, 9, 12]

This first application consists in describing the whole CTT operators using Event B models for the WIMP user interfaces. Each decomposition of an upper task in the CTT tree corresponds to an Event B refinement. A CTT task is described by an initial state and a final state. It is refined into a sequence of atomic events which lead from the initial state to the final state. The refinement preserves all the properties of the initial task. This process is repeated until atomic events, of the *dialog controller* are reached in the leaves. When the atomic events of the dialog controller are reached by the refinement, the validation process is completed.

The previous encoding of CTT in Event B has been experimented on several task models for WIMP applications. Complete examples may be found in [8]. Moreover, the developed examples have shown that it is possible to have a generic translation of CTT task trees allowing task validation also for plastic interfaces.

Let us consider our approach on a small WIMP case study which allows to convert euros to dollars and vice-versa. The user enters, in a textfield component, the value he/she wants to convert then he/she makes the conversion by pushing either the €$>>$ \$ button component to convert euros to dollars, or \$ $>>$ € button component to convert dollars to euros. The converted value is displayed thanks to a textfield component.

The dialog controller - A list of atomic events defining the transition system of the dialog controller of the exchange currency application is given below.

Event Name	Description
Evt_{Exit}	Click on exit button
$Evt_{ExitApplication}$	Close exchange application
$Evt_{InputValue}$	Input any convert value
$Evt_{ReadOutputValue}$	Updating all views after a conversion
$Evt_{ConvertInEuro}$	Click on dollar to euro button
$Evt_{ConvertInDollar}$	Click on euro to dollar button

An user task model - A potential user task model, giben below, of the exchange currency application has been experimented from our encoding CTT. Notice that the leaves of this user task model correspond of the atomic avents of the dialog controller.

$$\begin{array}{ll} ExchangeApplication & = ExchangeTask^{*} [> Evt_{Exit} >> Evt_{ExitApplication} \\ ExchangeTask & = Evt_{InputValue} >> ChoiceOfConversion >> Evt_{ReadOutputValue} \\ ChoiceOfConversion & = Evt_{ConvertInEuro} [] Evt_{ConvertInDollar} \end{array}$$

Application to the multi-modal user interfaces [28, 11]

In this kind of application, several modalities (i.e. Speech, Gesture or Direct Manipulation) are available. They may be used in order to build an interaction with the system. The building process associated to the interaction is based on a composition of sub-interactions which are themselves compositions of other sub-interactions and so on, until basic events (from dialog controller) are reached. Composition operators are needed in order to build such multi-modal interactions. So, the application to the multi-modal user interfaces consists in applying the whole CTT operators translation of previously introduced to compose multi-modal interactions. Regarding the WIMP applications, multi-modal applications need to express and check by Event B method another kind of properties, the CARE (Complementary, Assignment, Redundancy and Equivalence) properties [19].

As a case study, we have specified and validated the multimodal MATIS (Mulitmodal Airline Travel Information System) application defined in [30]. This application allows an user to retrieve information about flights schedules using speech and/or direct manipulation with keyboard and mouse, or a combination of these modalities. This interaction mode supports individual and synergistic use of multiple input modalities. The information about flights schedules correspond to the parameters of the request, such as the departure and arrival city names and the min and max departure hours.

The dialog controller - Below the list of the atomic events of the dialog controller MATIS case study is given. It corresponds to the transition system defining the multi-modal interactive application.

Event Name	Description
$Evt_{CityDepartDM}$	Input the departure city thanks to the direct manipulation modality
$Evt_{CityDepartSpeech}$	Input the departure city thanks to the voice modality
$Evt_{CityDepartGesture}$	Input the departure city thanks to the gesture modality
$Evt_{CityDepartDMSG}$	Input the departure city thanks to all the three modalities
$Evt_{CityArrivalDM}$	Input the arrival city thanks to the direct manipulation modality
$Evt_{CityArrivalSpeech}$	Input the arrival city thanks to the voice modality
$Evt_{CityArrivalGesture}$	Input the arrival city thanks to the gesture modality
$Evt_{CityArrivalDMSG}$	Input the arrival city thanks to all the three modalities
$Evt_{ResultOfRequest}$	Display the result of request
...	...

An user task model - We have experimented our encoding CTT user tasks model on the MATIS application. A potential user task model is shown below. This user task model expresses the capability to modify the different parameters before processing the request (iterative process) and the capability to input parameter values in different orders using a concurrent interaction mode involving all the three modalities (Speech, Direct Manipulation and Gesture).

$$\begin{array}{ll}
SearchFly & = InputData^{*}[> Evt_{ResultOfRequest} \\
InputData & = CityDeparture||CityArrival||MinHourDepart||MaxHourDepart \\
CityDeparture & = Evt_{CityDepartDM}[]Evt_{CityDepartSpeech}[] \\
 & \qquad Evt_{CityDepartGesture}[]Evt_{CityDepartDMSG} \\
CityArrival & = Evt_{CityArrivalDM}[]Evt_{CityArrivalSpeech}[] \\
 & \qquad Evt_{CityArrivalGesture}[]Evt_{CityArrivalDMSG} \\
MinHourDepart & = \ldots \\
MaxHourDepart & = \ldots
\end{array}$$

USE OF THE EVENT B METHOD. PROOF TECHNIQUE VERSUS MODEL CHECKING TECHNIQUE

The previous translation rules cover the whole CTT language for user tasks modelling and description. These translation rules give not only a syntactical translation, but also give a formal semantics using the Event B method semantics for the CTT language. All these rules are implemented and are tool supported. The Atelier B [17] tool is used for this purpose. We have developped several examples of CTT tasks using this approach.

When a CTT task is defined, the corresponding decomposition tree corresponds to a set of models and refinements designed using event B. These models contain **ASSERTIONS** clauses expressing the soundness of events occurrence thanks to the variant behavior and to the guard disjunction property. In addition to the task model validation provided by the event B technique, these clauses allow to express other properties and to prove them thanks to the used technique. It means that it is possible to validate or invalidate properties on the CTT tasks descriptions.

Finally, the use of arbitrary natural numbers in the interruption and disabling operators is possible using the **ANY** . . . **WHERE** . . . **THEN**. . . **END** or :∈ event B operators. The possibility of using arbitrary natural numbers allow to deal with all the possible cases for tasks descriptions and modelling. Moreover, the proving system supported by event B method allows to prove all the properties expressed in these models. Notice that this is almost impossible in model checking techniques, where a fixed value for the natural numbers is required. Usually the state number explosion problem arises when these natural numbers increase.

Nevertheless, we claim that model checking techniques and proof based techniques shall be used in conjunction in order to get the benefits of both: automatic proving for model checking techniques and state number explosion avoidance for proof based techniques. Proof based techniques have been used to validate user needs in WIMP (Windows Icons Menus and Pointers) applications[8] and to express and to check multi-modal properties[11].

CONCLUSION

This paper has presented an approach allowing to translate a BNF grammar to a set of Event B models. The translation rule we defined consists in associating a first event B model to the left hand side of a BNF rule and a second one to the right hand side of this rule. The second B model refines the first one. Variants are defined to ensure the correct firing order of events in these models. The abstract syntax tree describes the hierarchy of refined models. When these models are built, it is possible to enrich them by defining state variables, transitions and invariants expressing relevant properties. The interest of such models is to establish properties and to check them a priori allowing early validation/verification. Moreover, we have applied this translation rule to the BNF grammar defining process algebra expressions. These expressions were used to describe tasks and scenarios used to validate user interfaces.

Although the proposed approach produced satisfactory results, principally for user interfaces validation, it needs a deep study. Indeed, there is a need to address the following points :

1. study of other BNF examples and other languages for which useful proofs could be performed. These examples will improve the translation rule definition;
2. study the theoretical aspects. What are the necessary and/or sufficient conditions that shall be fulfilled by a BNF grammar to be translatable in Event B models;
3. and finally study and/or develop an automatic (or semi-automatic) tool that allows parsing of BNF grammars to a hierarchy of Event B models corresponding to an abstract syntax tree.

REFERENCES

[1] J-R Abrial. Extending b without changing it (for developing distributed systems). In H Habrias, editor, *First B Conference, Putting Into Pratice Methods and Tools for Information System Design*, page 21, Nantes, France, 1996.

[2] J.R. Abrial. *The B Book. Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] Y. Aït-Ameur, B. Bréholée, L. Guittet, F. Jambon, and P. Girard. Formal Verification and Validation of Interactive Systems Specifications. Technical report, LISI/ENSMA, March 2000.

[4] Y. Aït-Ameur and P. Girard. Specification, Design, Refinement and Implementation of Interactive Systems: the B Method. Technical report, LISI/ENSMA, March 2001.

[5] Y. Aït-Ameur, P. Girard, and F. Jambon. Using the B Formal Approach for Incremental Specification Design of Interactive Systems. In S. Chatty and P. Dewan, editors, *IFIP TC2/WG2.7 Engineering for Human-Computer Interaction*, pages 91–110. Kluwer Academic Publishers, 1998.

[6] Yamine Aït-Ameur. Cooperation of formal methods in an enngeneering based software development process. In Spring Verlag LNCS 1945, editor, *Second International conference on Integrated Formal Methods, IFM*, pages 136–155, Schloss Dagstuhl, 2000.

[7] Yamine Aït-Ameur and Mickaël Baron. Bridging the gap between formal and experimental validation approaches in hci systems design : use of the event b proof based technique. In Department of Computer Science University of Cyprus, editor, *ISOLA 2004 - 1st International Symposium on Leveraging Applications of Formal Methods*, pages 74–81, Paphos, Cyprus, 2004.

[8] Yamine Aït-Ameur and Mickaël Baron. Bridging the gap between formal and experimental validation approaches in hci systems design : use of the event b proof based technique (revue). *Accepted in International Journal on Software Tools for Technology Transfer Special Section*, 2005.

[9] Yamine Aït-Ameur, Mickaël Baron, and Patrick Girard. Formal validation of hci user tasks. In Al-Ani Ban, Arabnia H.R, and Mum Youngsong, editors, *The 2003 International Conference on Software Engineering Research and Practice - SERP 2003*, volume 2, pages 732–738, Las Vegas, Nevada USA, 2003. CSREA Press.

[10] Yamine Aït-Ameur, Patrick Girard, and Francis Jambon. A uniform approach for the specification and design of interactive systems: the b method. In Panos Markopoulos and Peter Johnson, editors, *Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'98)*, volume Proceedings, pages 333–352, Abingdon, UK, 1998.

[11] Yamine Aït-Ameur and Nadjet Kamel. A generic formal specification of fusion of modalities in a multimodal hci. In Ren Jacquart, editor, *IFIP World Computer Science*, pages 415–420, Toulouse, France, 2004. Kluwer Academic Publishers.

[12] Mickaël Baron. *Vers une approche sûre du développement des Interfaces Homme-Machine (Thesis)*. Thèse de doctorat, Université de Poitiers, 2003.

[13] L. Bass, E. Hardy, K. Hoyt, R. Little, and R. Seacord. The arch model : Seeheim revisited, the serpent run time architecture and dialog model. Technical Report CMU/SEI-88-TR-6, Carnegie Melon University, 1988.

[14] l. Bass, R. Pellegrino, S. Reed, S. Sheppard, and M. Szezur. The arch model : Seeheim revisited. In *User Interface Developper's Workshop*, 1991.

[15] D. Bjorner. VDM a Formal Method at Work. In Springer-Verlag. LNCS, editor, *Proc. of VDM Europe Symposium'87*, 1987.

[16] Dominique Cansell. *Assistance au dveloppement incrmental et sa preuve*. Habilitation diriger les recherches, Universit Henri Poincar, 2003.

[17] ClearSy. Atelier b - version 3.5, 1997.

[18] J. Coutaz. Pac, an implementation model for the user interface. In *IFIP TC13 Human-Computer Interaction (INTERACT'87)*, pages 431–436, Stuttgart, 1987. North-Holland.

[19] J. Coutaz, L. Nigay, D. Salber, A. Blandford, J. May, and R.M. Young. Four easy pieces for assessing the usability of multimodal interaction: the CARE properties. In *Proceedings of Human Computer Interaction - Interact'95*, pages 115–120. Chapman and Hall, 1995.

[20] E.W. Dijkstra. In *A Discipline of Programming*. Prentice-Hall Englewood Cliffs, 1976.

[21] E.W. Dijkstra. In *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[22] Jean-Daniel Fekete. *Un modèle multicouche pour la construction d'applications graphiques interactives*. Doctorat d'université (phd thesis), Université Paris-Sud, 1996.

[23] Phil Gray, David England, and Steve McGowan. Xuan: Enhancing the uan to capture temporal relation among actions. Department research report IS-94-02, Department of Computing Science, University of Glasgow, February 1994. Modifications par rapport UAN : - Aspect symtrique USER/SYSTEM - Contraintes temporelles - Paramtrisation des tches - Pr et Post-conditions.

[24] Laurent Guittet. *Contribution l'Ingénierie des Interfaces Homme-Machine - Théorie des Interacteurs et Architecture H4 dans le système NODAOO*. Doctorat d'université (phd thesis), Université de Poitiers, 1995.

[25] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *CACM*, 12(10):576–583, 1969.

[26] C.A.R. Hoare, I.J. Hayes, He Jifeng, C.C. Morgan, A.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of Programming. *CACM*, 30(8), 1987.

[27] C.A.R. Hoare and N. Wirth. An Axiomatic Definition of the Programming Language Pascal. *Acta-Informatica*, 23(2):335–355, 1973.

[28] Nadjet Kamel. Utilisation de smv pour la vrification de proprits d'ihm multimodales. In *16 Confrence Francophone sur l'Interaction Homme-Machine (IHM'2004)*, volume 1, pages 219–222, Namur, Belgique, 2004. ACM Press.

[29] K. Lano. *The B Language Method: A Guide to Practical Formal Development*. Springer Verlag, 1996.

[30] Laurence Nigay. *Conception et Modlisation Logicielle des Systmes Interactifs : Application aux Interfaces Multimodales*. Doctorat d'universit (phd thesis), Universit Joseph Fourier, 1994.

[31] F Patern, G Mori, and R Galimberti. Ctte: An environment for analysis and development of task models of cooperative applications. In *ACM CHI 2001*, volume 2, Seattle, 2001. ACM/SIGCHI.

[32] Fabio Patern. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2001.

[33] Gnther E Pfaff, editor. *User Interface Management Systems, Proceedings of the Workshop on User Interface Management Systems held in Seeheim*. Eurographic Seminars. Springer-Verlag, Berlin, 1985.

[34] D L Scapin and C Pierret-Golbreich. Mad : Une méthode analytique de description des tâches. In *Colloque sur l'Ingénierie des Interfaces Homme-Machine (IHM'89)*, pages 131–148, Sophia-Antipolis, France, 1989.

[35] J M. Spivey. *The Z notation: A Reference Manual*. Prentice–Hall Int., 1988.

AN AUTOMATED TESTING EXPERIMENT FOR MULTI-LAYERED EMBEDDED C CODE

Boutheina Chetali, Marie-Noelle Lepareux and Quang-Huy Nguyen
Axalto, Smart Cards Research,
34-36 rue de la Princesse, 78431 Louveciennes Cedex, France
{bchetali,qnguyen}@axalto.com

ABSTRACT

This paper describes an experiment using an automated tool for testing smart cards embedded software developed in C. Conventional testing of smart cards uses low-level commands and writing the use cases and test scripts is a error-prone and tedious task. Our goal was to show how one can use a formal tool to improve the testing process in order to concentrate better on the efficiency of the test. The approach consists in modeling each layer of the system independently while abstracting the services provided by the lower layers. The model is then verified, simulated and test cases are automatically generated using test criteria such as branch coverage or statement coverage. To use those generated test scripts, we developed a translator to execute the scripts on the C implementation of the system. We show the results obtained and the lessons learned from the application of this approach to the validation phase of a smart cards file system manager.

1 INTRODUCTION

Smart cards provide a high level of security in a large variety of domains such as banking, mobile communication, public transport and e-government. Strictly bounded resource (in terms of memory and CPU) is the first major constraint to consider while building software for these tiny computers. A typical card has no more than 128 bytes of RAM, 64K of persistent memory (E^2PROM or flash memory) and is powered by a 8-bit microprocessor with clock speeds up to 5Mhz [1]. The second major requirement for smart cards software is correctness and robustness. Correctness is usually verified using test, but conventional testing of smart cards uses low-level commands and the writing of use cases and test scripts is tedious and error-prone. A way to improve the testing phase is to use an automated tool which can also improve the test coverage. Our goal is then to show how one can use a formal tool to improve the testing process in order to better concentrate on the efficiency of the software testing.

Another requirement for smart cards software is portability. There are numerous hardware providers in the market and their chip-sets have different resource access services. In order to optimize the use of resource and to ease portability into different hardware platforms, smart cards software, in particular the Operating System (OS), is often built using a multi-layered architecture. For example, the file system manager, which is a core component of a smart cards OS, can be implemented in 3 layers: FSM (File System Module), HIM (Hierarchical Management), and DMM (Dynamic Memory Management) layers. Each layer corresponds to a particular abstract level of memory usage. In the FSM layer, the memory is viewed as an usual directory tree *i.e.,* a set of files and subdirectories. The primitive memory operations are provided by the DMM layer where memory is divided into blocks and fragments *i.e.,* series of non-adjacent blocks. The HIM layer manages the hierarchical structure of these fragments and classifies them for being used in the FSM layer.

Manually testing this type of software is tedious and error-prone. No real independent unit test can be performed in the upper layers. Because the tester also needs the correct responses from the lower layers, it is not possible to test one single layer independently of the other one. As the previously tested lower layers can never be considered totally error free, error locating (when the test case fails) in the layer under test is very time consuming. The whole manual process is therefore very long and does not include any reusable methodology and its quality is difficult to be assess. In this context, our objective is to provide a more reliable and efficient validation method (test and/or verification) for this kind of software. The main idea is to formally model the system and then to generate test cases from the model in a systematic (and as automatic as possible) manner. Taking into account this multi-layered nature of the software, our approach consists in building in an incremental way the integration tests starting from the unit tests. This could be done using a tool which proposes a modular design of the formal model.

Inside a multi-layered software, a layer can be seen as a *reactive subsystem*, which interacts during its lifetime with the upper and lower adjacent layers. This feature advocates the use of a *synchronous language* for modeling purpose[1]. The choice of Esterel Studio is based on the criteria of reliability, automation and user-friendliness. This experiment is made in an industrial context where one of the main issues we address is the acceptance of the tool by the engineers without changing the culture and the existing development process. Consequently, our main results could be valid for other formal tools providing that they have solid theoretical bases and the sufficient industrial maturity.

In this paper, we provide a modular method for unit testing multi-layered C programs. The tester first identifies the interfaces between the modules of the same or different layers. Then, in the layer under test, each control-related module, *i.e.,* module that contains at least one control structure, is identified and modeled. Note that for the modules that do not include any control structure (*e.g.,* purely mathematical operations) the test case generation is more trivial (no constraint solving is needed to compute the inputs while the outputs can be find out by the simulator). The obtained models will be used to generate test cases *w.r.t.* a coverage criterion.

Obviously, additional work has been done to allow the modeling of C function using a synchronous language. As their execution models are quite different, we use the constraint facilities to simulate the C execution (see Section 2.4). Furthermore, we need to provide an appropriate solution for some typical problems of automated testing:

Incompletion of test case generator in terms of test coverage: Esterel Studio generates the test cases using a SAT-based solver which may fail to fulfill the coverage criteria. As in [2], we need to manually complete the set of test cases. This manual intervention is simplified using Esterel Studio and is described in Section 2.6.

Execution of test cases on the implementation: in order to effectively use the generated test cases, we need to translate them into a test plan in C syntax and link them with the C implementation. This process is realized by an automatic tool developed in Section 3.

Correctness of the unit testing: the functions under test can call some external services. To ensure the correctness of the unit testing, the behavior of those external services is mimicked independently from their semantics while executing the test cases.

The rest of this paper is organized as follows. Section 2 presents a method for modeling C functions and for generating test cases in Esterel Studio. In Section 3, we describe the translation and execution of test cases on the C implementation of these modules. Section 4 provides a case study where our method is applied to a smart card file system module, in particular, the obtained results and the learned lessons. We discuss the related work in Section 5 before giving some concluding remarks and future directions in Section 6.

[1]Synchronous languages such as Esterel and Lustre have been shown to be very appropriate for modeling reactive systems thanks to their precise semantics and operational model [2].

2 MODELING C FUNCTIONS AND GENERATING TEST CASES

2.1 The Esterel Studio tool-set

Esterel [3] is a modeling language based on the synchronous approach where the time is discrete and the reaction time is not taken into account (the execution time is hence said to be zero). This approach is shown to be particularly appropriate for modeling reactive systems *i.e.,* the systems that are continually dependent on the environment.

Esterel Studio is a graphical tool-set developed by Esterel Technologies that allows the user to specify a system with the following assurance: the graphical representation (the specification) of the system is executable and can be checked in terms of determinism, causality, and formal properties. The graphical language SyncCharts [4] is used to build the specification which is then translated into Esterel syntax. The user can also write some or all modules of the model directly in Esterel syntax. The tool will link all these modules together to build an executable model (by simulating). Moreover, Esterel Studio also provides a tool for verifying properties of the model. An automatic tool is also integrated which allows symbolic execution paths to be generated from the model *w.r.t.* a coverage criteria.

In Esterel, the interface between a system and the environment is provided by the *signals*. A signal may be attached with a value (*valued signal*) or not (*pure signal*). Arriving signals are input while broadcasting signals are outputs of the system. Signals can be hence used to interface different modules of a system. SyncCharts also provides the notion of *port* which is a group of different signals. The evolution of the whole system is the evolution of the underlying finite state machine: a transition of this machine is the reaction of the system to a given configuration of input signals.

2.2 An outline of the method

Starting from the informal specification of the function under test, the user develops a SyncCharts model describing its behavior. Thanks to the simulator and the verifier of Esterel Studio, the developer gains more confidence in the validity of the model and can use it as an oracle in the testing process of the C implementation. Then the user uses several tools of Esterel Studio (execution paths generator, constraints solver and simulator) for elaborating a series of test cases which fill a given test criteria. Two available test criteria are reachable state coverage and transition coverage.

Our translator transforms the test cases into a test plan that calls the function under test with appropriate arguments, and eventually provides returns from the related modules, *i.e.,* the modules that contain external called functions. The test verdict (pass/fail) is obtained by checking execution result of the function under test against the output generated by model simulating. This process allows us to perform a real unit test of the C implementation. In fact, in the test plan, returns from external called functions are replaced by values proposed by the test case generator in order to perform the test cases.

2.3 Modeling C functions

Interface with the environment

We call *interface variables* the following elements of a C function:

- the arguments
- the returns (values and exceptions)
- the global variables used by the function
- the calls to external functions
- the arguments transferred to external functions
- the returns of external functions

All these interface variables are modeled by signals which are attached with a value if it is necessary. The function arguments and the returns of the external functions are modeled by input valued signals. The returns are modeled by output valued signals while the calls to external functions are modeled by pure signals. The global variables and the arguments transferred to external functions are modeled by one input (for their old value) and one output signal (for their new value if any) because their value can be modified during the lifetime of the function[2].

In order to keep the correspondence between the model and the code, the interface variables must appear in the same order in Esterel Studio and in C. In Esterel, the appearing order of signals does not have any influence on the model execution. However, in C code, an order must be fixed between interface variables to ensure the determinism of the program.

Datatypes

All C primitive data types can be modeled in Esterel except the pointers. Since Esterel does not support pointers, we must transform them into arrays and hence, fix the size of data they point to. In other words, the flexibility of the pointers in C cannot be totally reproduced. A data structure is modeled using a SyncCharts port whose each signal models an element of the structure.

Control structures

All generally used control structures can be modeled in SyncCharts. The `if-then-else` structure is modeled through paths between simple states, modules. The `case-of` structure is modeled in the form of a conditional connector that automatically includes a default path (for the default case). The loop structures `while` and `for` are modeled like in any other flowchart language using a `if-then-else`. The pure computing parts of the program should be directly written in Esterel syntax which is sufficiently expressive for this task. A function call is modeled by a pure signal while the transferred arguments are modeled by valued signals.

Example **1** *Consider the function `max3(int a,int b,int c)` that returns the maximum of* 3 *given integers. This function calls another function (`max2(x,y)`) which is contained in a separate module (max.c) and returns the maximum of* 2 *given integers.*

```
int max2(int x,int y); /* external called function from max.c */
int max3(int a, int b, int c) {
  if (a > b) ret = max2(a,c)
  else ret = max2(b,c);
  return ret;
}
```

The SyncCharts *model is shown in Figure 1. An `if-then-else` control structure based on the condition (`?max3.A>?max3.B`) divides the model into two branches corresponding to two different calls of `max2`. Such a call is modeled by the pure signal `max2.Call`. The two arguments transferred to `max2` are modeled by two valued signals `max2.X` and `max2.Y`. The return of `max2` is modeled by the valued signal `R_max2` and is used to build the return of `max3` (i.e., the valued signal `R_max3`).*

2.4 Model of constraints

The model of constraints is used to enforce the semantics of a model in order to bridge the gap between the Esterel model and the C code. The constraints restrict the possible sequence of input signals into the main model. The model of constraints has as input signals, all input and output signals of the main model, and as unique output, an "Error" signal. The four typical constraints are:

1. the arguments of the modeled function must arrive before the first instant of the model simulation lifetime.

[2]The modification of global variables specifies the side-effect of the function under test.

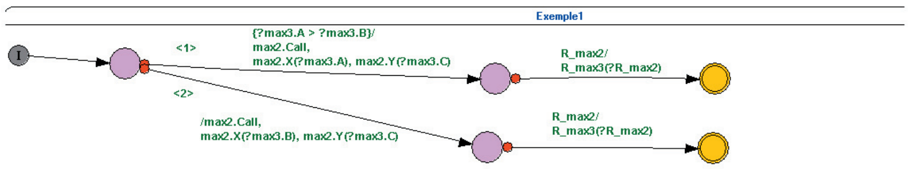


Figure 1: SyncCharts model of `max3`

2. no response from a called function can arrive before the corresponding call: this constraint ensures chronological order between any function call and its response.

3. all external called functions must return after being called in the same test case.

4. the last output signal must be a signal containing a real output, *i.e.,* an exception or a returned value of an expected type or a global variable modified by the function under test.

2.5 Model verification

The first step of the validation consists in using the models for simulating the behavior of the system. The tool implicitly verifies the soundness of the formal model. Indeed, a successful execution of the model ensures that it is sound in terms of determinism and causality. Furthermore, the simulation allows the user to gain more confidence in the model *w.r.t.* its expected behavior.

The second step consists in verifying more complex properties like the broadcast of an output, or the validity of an assertion. Each property is represented by a synchronous observer, *i.e.,* a SyncCharts module running in parallel with the model and having as input, the input and the output of the model itself. If the property is satisfied, a specific signal is emitted by this observer.

2.6 Test case generation

The test case generation is done in two steps. The first step consists in generating the *symbolic execution paths* of the model *w.r.t.* to a test criterion. This step also takes into account a user-defined model of constraints described in Subsection 2.4, that restricts the sequence of input signals. An *execution path* is an ordered set of transitions from the entry point to an exit point of the model. The execution paths generated in this first step are said to be symbolic because they do not consist of real transitions but only of a set of of constraints on the input signals (*i.e.,* on their presence and value) that triggers those transitions.

Example **2** *The set of constraints S $(?S < 2)$ $(?T > 4)$ R requires the arrival of the input signals S and R . Moreover, the value of S must be less than 2. The signal T is not obligated to arrive in this step but its last broadcast must have been done with a value greater than 4.*

The most useful test criterion used in this step is the coverage (on the model), either in terms of reachable states or of transitions between them. This property requires every reachable state of the model (respectively every transition) to be passed through by an execution path.

In the second step, the constraints solver (Esverify developed by Prover Technology) is used for computing a solution for each symbolic execution path, *i.e.,* a set of values for its input signals that allow the transitions to be produced. In Example 2, one solution for the constraints is $T = 5$; $S = 1$; $R = 2$. Because the input signals correspond to the arguments and global variables in the C code, their values are actually the input test data of the corresponding test case. The output data (*i.e.,* test oracle) is computed by the simulator by replaying the execution path. Note that some test cases may not correspond to a real execution path in C. For example, the test cases ending in a state of (infinitely) waiting for response from an external called function have no correspondence in C because a C function is assumed to always give a response. These test cases will be rejected by the translator described in Section 3.

Esverify is a SAT-based proof engine whose performance is sensitive to the size of the constraints. If Esverify is not able to solve the constraints for fulfilling the coverage criterion, then a user interaction is required. In Esterel Studio, this interaction is first done by visualizing the missing states or transitions. After having observed missing elements, the user can manually complete the execution paths with help of the simulator, by entering at each step the valued signals to cover the missing states or transitions. In other words, the user needs to interactively solve the constraints on the signals to compute the input data of the test cases.

3 TEST CASE TRANSLATION AND EXECUTION

In order to effectively test a C function, these test cases must be translated into a test plan in C syntax. This translation can be seen as a refinement of the execution scripts from the abstract model to a concrete implementation. This refinement can reject some test cases that do not respect the C execution semantics. Furthermore, some specific powerful features of the Esterel models must be treated with care, *i.e.,* the parallel parameter checks must be serialized.

For each test case, the translated C code prepares the context (affectation of global variables) and simulates a function call using the arguments. During its execution, if the function under test calls another function, then the return of this function is also simulated. Note that all the necessary information (values of global variables, values of arguments, returned values from external called functions) is contained in the input data of the test case. The return from the function call (returned value or exception) and its side-effect (the new values of global variables) are then checked against the output data of the test case in order to determine the test verdict.

Figure 2 gives an outline of the translation and the execution of the test cases. The tester, the modeler and the programmer are not necessarily three different people. The translation process is done in three steps: (1) parsing and checking the test cases, (2) translating Esterel signals into C symbols and (3) generating test plan in C syntax. The main input of the first phase is the set of test cases generated by Esterel Studio (*.esi* file). Because the initialized values of the signals are not provided in the test cases, the translator needs to look for those values in the model in Esterel syntax (*.strl* file). The name dictionary is needed to translate Esterel names into C names. The test result can be `Pass` or `Fail` or an unexpected exception. The latter is added in order to more easily locate errors in the C code.

3.1 Parsing and checking the test cases

A test case consists of:

- a set of input data (*i.e.,* arguments, global variables and returns of external called functions),
- a set of function calls, and
- an expected output data (*i.e.,* returned value or raised exception and/or new values of global variables).

Example **3** *A test case generated from the model of* `max3` *is as follows (excerpt from max3.esi):*

```
1. max3.A = 127 max3.B = 1
2. % Output:;
3. % Output: max2.Call max2.X = 127 max2.Y = 0;
4. R_max2 = 0
5. % Output: R_max3 = 0;
```

Two arguments `a` *and* `b` *of* `max3` *are valued in the line* 1. *The value of* `c` *is its initial value defined in max3.strl:*

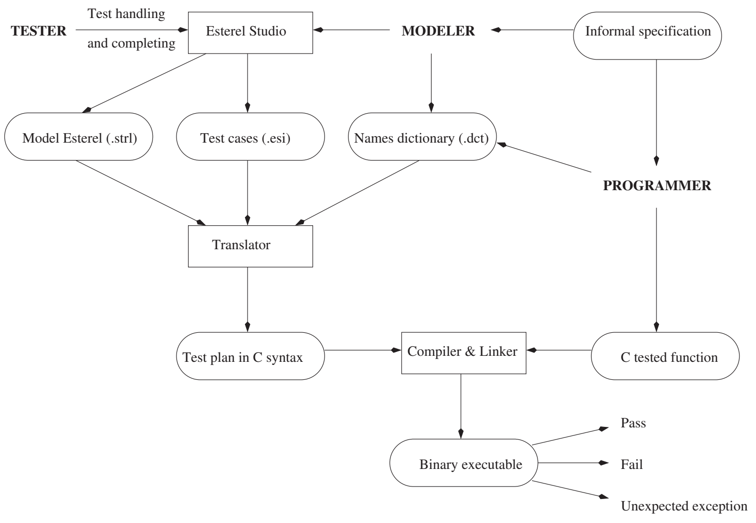


Figure 2: Test cases translation and execution

```
refine max3.A:init 0;
refine max3.B:init 0;
refine max3.C:init 0;
```

In this case, `a>b` *and* `max3` *calls* `max2(a,c)`*. This call is defined in line* 3 *(*`x=127` *and* `y=0`*). The mimicked returned value of* `max2` *is* 0 *(line* 4*). Note that no semantics of* `max2` *is assumed and any integer can be accepted as its return. Finally, the expected returned value of this test case is also* 0 *(line* 5*).*

Inside the translator, the parser not only checks the integrity of each test case but also the model constraints described in Section 2.4. If these constraints are not satisfied, the test case does not respect the C execution semantics and shall be ignored.

3.2 Translating Esterel signals into C symbols

In order to link the generated test plan with the C code, the signals used in the test cases must share the names with the symbols in this function. For that, the translator uses a names dictionary (the *.dct* file in Figure 2) where the tester defines the mapping between signals in the model and symbols in the C code. In this dictionary, a signal is assigned to an external name which contains sufficient information to generate the corresponding C symbol. The syntax of the external names is described as follows where the symbol — is used to separate different components of a name:

$$
\begin{array}{lcl}
signal\_external\_name & ::= & simple\_signal \\
 & & \mid composed\_signal \\
composed\_signal & ::= & simple\_signal \\
 & & \mid simple\_signal.composed\_signal \\
simple\_signal & ::= & no\_argument\_function \mid void\_return\_function \\
 & & \mid global\_variable \mid argument \mid return \mid exception \\
 & & \mid call\_argument \mid function\_call \mid call\_return \\
 & & \mid modified\_argument \\
no\_argument\_function & ::= & DO\text{—}module\text{—}function \\
void\_return\_function & ::= & DONE\text{—}module\text{—}function \\
exception & ::= & module\text{—}name \\
argument & ::= & A\text{—}arg\_ordinal\text{—}arg\_name\_type
\end{array}
$$

$call\_argument$	::=	$\mid module—function—A—arg\_ordinal—arg\_name\_type$
$global\_variable$	::=	$G—arg\_name\_type$
		$\mid module—function—G—arg\_name\_type$
$return$	::=	$R—simple\_name\_type$
$call\_return$	::=	$module—function—R—simple\_name\_type$
$modified\_argument$	::=	$module—function—R—A—arg\_ordinal—arg\_name\_type$
arg_name_type	::=	$simple\_name\_type \mid array \mid pointer$
$simple\_name\_type$	::=	$type\_c——name$
$pointer$	::=	$type\_c—P—name—size$
$array$	::=	$type\_c——name—size$
$function\_call$	::=	$module—function$
$module \mid function \mid name$	::=	$[a \ldots zA \ldots Z]^+[a \ldots zA \ldots Z0 \ldots 9]^*$
$size \mid arg\_ordinal$	::=	$[1 \ldots 9]^+[0 \ldots 9]^*$

Most of these above identifiers are self-explained. A signal can be simple or composed. A composed signal denotes an element of a C structure. The identifiers $argument$, $return$, $global\_variable$, $call\_return$, $call\_argument$ and $modified\_argument$ respectively represent the function arguments, the function returns, the global variables, the returns from external functions, the arguments transferred to external functions and the modified arguments by external functions. The identifier $type\_c$ can be any of the primitive or defined data types used in the C code (except the structures). Pointers are represented by arrays with pre-fixed size. An external function call ($function\_call$) is identified by its name and by the module it belongs to.

Example **4** *The names dictionary for Example 1 is as follows (—is replaced with* `__` *for parsing purpose):*

```
max3.A -> A__1__max3.int____a  max2.X -> max__max2__A__1__int____x
max3.B -> A__2__max3.int____b  max2.Y -> max__max2__A__2__int____y
max3.C -> A__3__max3.int____c  R_max2 -> max__max2__R__int____ret
max2.Call -> max__max2         R_max3 -> R__int____ret
```

3.3 Generating C test plan

The translator generates the test plan (*i.e.,* set of all test cases) in a Microsoft Visual C project which includes a `main.c` file, a link to the C module containing the function under test and a C module containing external called functions. For each test case, the `main.c` first declares and initializes the global variables, assigns the function arguments using the input data. Then, a call to the function under test is simulated. During its execution, the function under test may call other external functions whose we mimic the effects (*i.e.,* the returned value, the raised exception, the modification of the global variables and referenced arguments) in the external module. Finally, `main.c` uses an exception handler[3] to check the return (value or exception) of the function under test against the output data of the test case.

4 CASE-STUDY: A SMART CARDS FILE SYSTEM SERVICE

We describe the application of our process to the `CreateFile` service provided by the Axalto smart cards file system module. The considered file hierarchical structure is composed of 4 categories of files: RF (root file), DF (directory file), EF (elementary file) and LF (linked file, *i.e.,* shortcut to another file). A file system instance (FSI) can have only one RF but the file system module can manage several FSI stored in the persistent memory. Furthermore, some specific files can be shared between different FSIs using linked files. An elementary file can be simply a raw sequence of bytes (*transparent files*) but can also be structured as a sequence of records of variable size (*linear variable files*) or of fixed size (*linear fixed files*). In both of those files, the records (or the bytes) are ordered in a linear manner. However, the records can also be ordered in a cyclic manner (*cyclic fixed files*).

[3]By default, C has no exception but we use here an exception handling library developed in Axalto.

The file system module is made of three layers[4] (FSM/HIM/DMM) in order to optimize the memory in use and to ease the portability to different chip-sets. The file system maintains a global structure called the *file system context* which points to the currently selected file. `CreateFile` is one of the main functions of the FSM layer that creates a new file and correspondingly updates the file system context. This function has 14 arguments (*e.g.,* directory identifier, default access control rights, file size, type of file, file identifier, file status, number of records, length of records, path to the pointed file if the new file is a shortcut) and uses 9 services of the lower HIM layer. These services can return `void`, an exception or a value through a pointer.

4.1 Model construction and verification

The `CreateFile` function consists of two main parts, described in two separate modules. The first one consists in checking the validity of the request (*w.r.t.* the current context) while the second consists in realizing the command (file creation, file initialization and context update). In Esterel Studio, the `CreateFile` function is modeled by two main modules "Is the request admissible?" and "The request is admissible".

The module "Is the request admissible?" can be modeled in several styles, according to the order and the parallelism of the checks performed on the parameters. Parallel check is more efficient because it stops as soon as the first check fails. However, performing the checks in parallel must be done with care because in the corresponding C code, all checks are serial and each failed check may raise a different exception. The model on which our test experiment is based describes effectively the checks of the module "Is the request admissible?" in the same order as in the C code. It also calls the functions of the HIM layer in the same order as the C code does.

This model has 66 reachable states. Amongst them, 23 states correspond to the "stop states" with an exception raised by the function because the file creation is not possible in the current context. The other states are either initial state (1), final states (3) or intermediate state (39). The intermediate states correspond to the points of control from which one amongst several transitions can be produced according to the value of the signals at that stage. The whole finite state machine is driven by 33 input signals which are either pure or valued by a 16-bits integer.

The model is then checked by Esverify. In this stage, the properties to be ensured on the model are typically "one cannot access a file that has not been created before", "one cannot create a file with an existing identifier", "one cannot create a file if there is no more available memory", etc.

4.2 Test coverage

The symbolic execution paths have been generated *w.r.t.* the *transitions coverage* criterion. The solver did not succeed to solve all symbolic execution paths. Actually, we obtained only a coverage of 84%. Amongst the generated test cases, one has to be removed because it has no corresponding C execution. Using the Esterel Studio simulator, we could manually locate and complete the missing test cases.

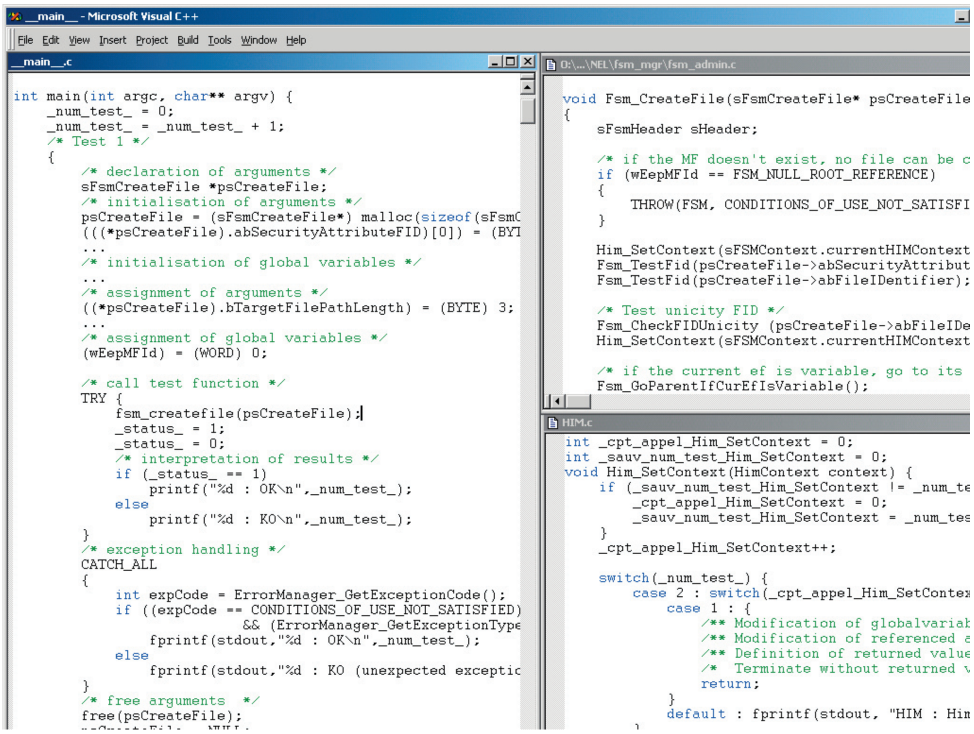
4.3 Generated C test plan

Figure 3 shows a test plan generated for the function `Fsm_CreateFile`. Only an excerpt from the declaration, initialization and assignment of the arguments and global variables is seen due to the space limit. The function call is put inside a `TRY...CATCH_ALL` structure to get and check its return against the expected output (exception `CONDITIONS_OF_USE_NOT_SATISFIED`). The C function `fsm_createfile` is encoded in `fsm_admin.c` using the function `Him_SetContext` which is mimicked in `HIM.c`.

4.4 Results and learned lessons

During the development of the formal model in Esterel Studio, we faced the incompleteness and impreciseness of the informal specification. We needed a lot of interaction with the development team in order to check the

[4]See Section 1 for an explanation.


__main__ - Microsoft Visual C++
File Edit View Insert Project Build Tools Window Help
__main__.c
O:\...\NEL\fsm_mgr\fsm_admin.c
HIM.c

```
int main(int argc, char** argv) {
    _num_test_ = 0;
    _num_test_ = _num_test_ + 1;
    /* Test 1 */
    {
        /* declaration of arguments */
        sFsmCreateFile *psCreateFile;
        /* initialisation of arguments */
        psCreateFile = (sFsmCreateFile*) malloc(sizeof(sFsmC
        (((*psCreateFile).abSecurityAttributeFID)[0]) = (BYT
        ...
        /* initialisation of global variables */
        ...
        /* assignment of arguments */
        ((*psCreateFile).bTargetFilePathLength) = (BYTE) 3;
        ...
        /* assignment of global variables */
        (wEepMFId) = (WORD) 0;

        /* call test function */
        TRY {
            fsm_createfile(psCreateFile);
            _status_ = 1;
            _status_ = 0;
            /* interpretation of results */
            if (_status_ == 1)
                printf("%d : OK\n",_num_test_);
            else
                printf("%d : KO\n",_num_test_);
        }
        /* exception handling */
        CATCH_ALL
        {
            int expCode = ErrorManager_GetExceptionCode();
            if ((expCode == CONDITIONS_OF_USE_NOT_SATISFIED)
                            && (ErrorManager_GetExceptionType
                fprintf(stdout,"%d : OK\n",_num_test_);
            else
                fprintf(stdout,"%d : KO (unexpected exceptic
        }
        /* free arguments  */
        free(psCreateFile);
```

```
void Fsm_CreateFile(sFsmCreateFile* psCreateFile
{
    sFsmHeader sHeader;

    /* if the MF doesn't exist, no file can be c
    if (wEepMFId == FSM_NULL_ROOT_REFERENCE)
    {
        THROW(FSM, CONDITIONS_OF_USE_NOT_SATISFI
    }

    Him_SetContext(sFSMContext.currentHIMContext
    Fsm_TestFid(psCreateFile->abSecurityAttribut
    Fsm_TestFid(psCreateFile->abFileIDentifier);

    /* Test unicity FID */
    Fsm_CheckFIDUnicity (psCreateFile->abFileIDe
    Him_SetContext(sFSMContext.currentHIMContext

    /* if the current ef is variable, go to its
    Fsm_GoParentIfCurEfIsVariable();
```

```
int _cpt_appel_Him_SetContext = 0;
int _sauv_num_test_Him_SetContext = 0;
void Him_SetContext(HimContext context) {
    if (_sauv_num_test_Him_SetContext != _num_te
        _cpt_appel_Him_SetContext = 0;
        _sauv_num_test_Him_SetContext = _num_tes
    }
    _cpt_appel_Him_SetContext++;

    switch(_num_test_) {
        case 2 : switch(_cpt_appel_Him_SetContex
            case 1 : {
                /** Modification of globalvariak
                /** Modification of referenced a
                /** Definition of returned value
                /*  Terminate without returned v
                return;
            }
            default : fprintf(stdout, "HIM : Him
```

Figure 3: An example of test plan in C syntax

meaning and the purpose of described functions. However, this cooperation turned out to be very beneficial because both the model and the informal specification have been since significantly improved. In particular, the order of the tasks that can be performed in parallel in the model, must be defined in the informal specification.

The generated test plan enabled us to find out 1 flaw in the C code which has not been detected by the manually-built test plan. This flaw is caused by a comparison of two variables of different sizes. In fact, before creating a new file, CreateFile checks if the file size does not overcome a limit. However, the check defined in the C code is always successful (and hence meaningless) because it compares a Byte representing the file size to a Word representing the size limit. The corresponding test case has showed a difference between the behaviors of the C code and of the Esterel model.

In comparison with the conventional development in C, SyncCharts is more adapted for describing the control structures of an algorithm. In contrast, the linear parts of the algorithm, which do not contain control structures, need to be expressed in Esterel using a C-like syntax. In other words, building a SyncCharts model is less error-prone than developing the C code only if control structures are present. Furthermore, the quality of the test cases produced for fulfilling coverage criterion also depends on the quantity of control structures. The case study is actually chosen according to this feature.

Before this case study, a test plan of the file system module has been manually produced by Axalto test engineers. This is actually not a unit test because the HIM services are also involved. Furthermore, it is not easy for the test engineers to set the file system in a given state before each test case. The test criterion is output coverage, that is, coverage *w.r.t.* the function output (exceptions or file system context modifications). This test criterion is less rigorous than the criterion used in our method (transitions coverage) and hence, we generate more test cases. Indeed, there may be several execution paths which produce a given output. In our method, the coverage measurement and the ability to test each layer of the system independently are two critical advantages. Furthermore, our method is also more beneficial in terms of maintenance: if the specification of the system is changed, the adaptation of the test plan is much more costly in the conventional approach.

In an industrial context, the main concern is to reduce the global time of the software production, *i.e.,* of both development and validation phases. If the time for assimilating Esterel Studio tools is not included, then the cost for modeling and automatically unit testing is not longer than for a conventional testing approach. Actually, all the cost is dedicated to modeling and eventually to completing the missing test cases. The test case generation, translation and execution are fully automatic and require no extra-cost. In our case study, 3 man-months were needed to build the Esterel model and to validate the general methodology; the test case completion required 1 man-month; the (build-once but use-many-times) translator was encoded by 2 man-months. Note however that, a tester getting familiar with the specification of the system will need less time for modeling a second function. The test case completion is more tedious task because it requires the tester to follow carefully the test simulation (by Esterel Studio) in order to locate the missing cases. Fortunately, the test case generator already provides a good coverage rate. In summary, a complete test of all functions of the same layer will clearly improve the global time needed for the validation phase.

5 RELATED WORK

Model-based testing [5, 2, 6] has been used by numerous researchers in the validation of embedded code, in particular, in the smart cards industry. In mobile communication domain, B-Method has been used for testing several GSM applications [7, 8]. A WAP identity module embedded in the SIM cards has been tested using formal models developed in AutoFocus [9, 2]. In banking domain, an electronic purse application has been validated using TGV tool [10].

Both of those works are *black-box* testing [11], that is, no information of the code is used. They concentrate on integration test, that is, the whole system is tested by sending it the command APDUs (application protocol data units) and checking the response APDUs. Their main coverage criteria is hence *w.r.t.* to specification requirements. Our testing method can also perform unit test and like [12], we use a *gray-box* approach because we are actually testing an existing system. That is to say the model is built using the information from both the specification and the implementation. This practice is not really unusual in the industry where formal methods specialists often need to look at both of those elements in order to precisely understand the system. The gray-box approach keeps our model relatively close to the tested implementation and hence, the coverage results on the model and on the implementation are also strongly related. In [12], the authors use the Alloy tool-set for modeling and generating test cases for a reliability-analyzing tool. They also use a different coverage criterion (coverage of inputs data up to certain sizes). The case study is modeled in Z and hence, a translation to Alloy is needed. As stated by the authors, this translation is an additional risk of comprising test completeness.

Java Card [1] applications can be specified in JML (Java Modeling Language) and then be verified using dedicated tools. Unit test cases can also be generated from JML specifications [13] but this process is still semi-automatic because the user still needs to provide the input data (only output data, *i.e.,* test oracle is automatically generated). On the contrary, in [14], only input data for test cases are generated using the Java PathFinder, a model-checker for Java programs. An interesting point of that work is that the test generation can be directly applied to complex Java code.

The recently-born Spec# modeling language [15] can be seen as a C# counterpart of JML. Test cases can be generated from a Spec# specification using Spec Explorer [16]. This tool seems to be more powerful than Esterel Studio because it also enables the execution of test cases on .NET assemblies.

6 CONCLUDING REMARKS

We have described an effective method for using Esterel-based tools for unit testing multi-layered embedded C code. In this method, we first built a SyncCharts model of the C module under test and then, use the Esterel Studio tools to generate test cases from this model. A translator has also been built in order to translate and execute these test cases on the corresponding C code. Because the translator is independent of the Esterel Studio tools, it can be used to compare the behavior of the model and of the C code, using any suite of global variable, arguments and returns of the related modules (in the right order and with the right syntax). This feature allows the testers to manually complete any test cases series or even to build their own tests.

Model-based testing implies an extra-cost due to the construction of the model. However, the counterpart is a more precise representation of the user-requirements and the security policy and a possibility of checking the model *w.r.t.* those. These features are highly valuable in a sensitive domain such as smart cards where security is one of the main criteria for any product assessment. Recently, Common Criteria [17] has become a well-known standard that allows an independent organization to assess information systems in terms of security. Automated testing can be used in the evaluation of the smart cards products in a high assurance level (EAL5 to EAL7) of the Common Criteria ladder. For those levels, Common Criteria require a rigorous (correct and complete) correspondence between formal models and the test cases used for testing the implementation.

We are now extending this method to build a complete test environment for reducing the validation cost of cards embedded software. Future work consists in modeling completely the file system manager and then, a large part of the OS using a step-by-step methodology. Another interesting point consists of a more sophisticated use of the Esterel Studio verification tools to ensure the security properties of the developed model before using it to check against the C code (by test case generation). An approach for expressing the security properties of cards C embedded code is studied in [18] and can be adapted to this context. In many embedded system applications, Esterel Studio has been used to automatically generate C code from a model. At the time being, the generated C codes still do not fit in a smart card due to its restricted memory. However, this precise feature of Esterel Studio can be used in the development of the test cases translator described in Section 3, which is a critical component of our testing method. Finally, because a product is usually specified in UML by the programmers, we are looking (with Esterel Technologies team) at reusing (in an automatic manner) the UML specification while building an Esterel model.

References

[1] Z. Chen. *Java Card technology for Smart Cards: Architecture and Programmer's guide*. The Java Series. Addison-Wesley, 2000.

[2] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model based testing for real–the inhouse card case study. *J. Software Tools for Technology Transfer*, 5(2-3):140–157, March 2004.

[3] G. Berry. The foundation of Esterel. In G Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction, Essays in Honour of Robin Milner*. The MIT press, 2000.

[4] C. André. SyncCharts: a visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996.

[5] I. K. El-Far and J. A. Whittaker. Model-based software testing. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 825–837. John Wiley & Sons, 2001.

[6] Model-based software testing hompage. `http://www.geocities.com/model_based_testing/`.

[7] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11-11 standard case study. *International Journal of Software Practice and Experience*, 34(10):915–948, 2004.

[8] F. Bouquet, B. Legeard, F. Peureux, and E. Torreborre. Mastering Test Generation from Smart Card Software Formal Models. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 70–85. Springer-Verlag, March 2004.

[9] J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. *Electr. Notes Theor. Comput. Sci.*, 80, 2003.

[10] D. Clarke, T. Jron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *Proceedings of International Conference on Research in Smart Cards (e-Smart'01)*, volume 2140 of *Lecture Notes in Computer Science*, pages 58–70. Springer-Verlag, 2001.

[11] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[12] D. Coppit, J. Yang, S. Khurshid, W. Le, and K. J. Sullivan. Software assurance by bounded exhaustive testing. *IEEE Trans. Software Eng.*, 31(4):328–339, 2005.

[13] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *Proceedings of ECOOP 2002 – Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, June 2002.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[15] M. Barnett, K. Rustan, M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, and M. Huisman, editors, *Procs. of the Int. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2004.

[16] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with Spec Explorer (extended abstract). In *[19]*, pages 542–547.

[17] Common criteria for information technology security evaluation, August 1999. Version 2.1. Available at `http://www.commoncriteriaportal.org/`.

[18] B. Chetali, C. Paulin-Mohring, and J. Andronick. Formal verification of security properties of smart card embedded source code. In *[19]*, pages 302–317.

[19] J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors. *FM 2005: Formal Methods.* Lecture Notes in Computer Science. Springer-Verlag, 2005.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 17-09-2005	**2. REPORT TYPE** Conference Proceedings	**3. DATES COVERED** *(From - To)* 23–24 September 20005

4. TITLE AND SUBTITLE IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Editors: Tiziana Margaria, Bernhard Steffen, and Michael G. Hinchey	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Goddard Space Flight Center Greenbelt, MD 20771	**8. PERFORMING ORGANIZATION REPORT NUMBER** 2005-02349-0
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING REPORT NUMBER CP–2005–212788

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited, Subject Category: 59, 62
Report available from the NASA Center for Aerospace Information, 7121 Standard Drive, Hanover, MD 21076. (301)621-0390

13. SUPPLEMENTARY NOTES

T. Margaria: University of Göttingen, Göttingen, Germany; B. Steffen: University of Dortmund, Dortmund, Germany

14. ABSTRACT

This volume contains the Preliminary Proceedings of the 2005 IEEE ISoLA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation, with a special track on the theme of Formal Methods in Human and Robotic Space Exploration. The workshop was held on 23–24 September 2005 at the Loyola College Graduate Center, Columbia, MD, USA. The idea behind the Workshop arose from the experience and feedback of ISoLA 2004, the 1st International Symposium on Leveraging Applications of Formal Methods held in Paphos (Cyprus) last October–November. ISoLA 2004 served the need of providing a forum for developers, users, and researchers to discuss issues related to the adoption and use of rigorous tools and methods for the specification, analysis, verification, certification, construction, test, and maintenance of systems from the point of view of their different application domains.

15. SUBJECT TERMS

Leveraging Applications, Formal Methods, Systems, Verification, Validation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19b. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Michael Hinchey
Unclassified	Unclassified	Unclassified	Unclassified	139	**19b. TELEPHONE NUMBER** ***(Include area code)*** (301) 286-9057

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18