Programming Research Group

PROCEEDINGS OF THE WORKSHOP ON AUTOMATED VERIFICATION OF CRITICAL SYSTEMS (AVOCS'01)

David Nowak

PRG-RR-01-07

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

Preface

This research report contains the proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01) held at Oxford University Computing Laboratory on 19-20 April 2001. The aim of AVoCS is to foster a research community in verification in United Kingdom through encouraging communication among researchers. Specific objectives include concrete efforts at integration as well as the transfer of methods between different groups. Topics include but are not limited to:

- Abstraction methods (symmetry, ...)
- Applications to security
- Applications to hardware
- Case studies
- Connections with abstract interpretation
- Data independence
- Integrating theorem proving and model checking
- Model checking
- Model checking of infinite-state systems
- Model checking of real-time systems
- Performance and dependability evaluation
- Refinement checking
- Software verification
- Temporal and modal logic
- Theorem proving

This year, a special session featuring security was held. Invited presentations with accepted submissions were combined.

This technical report include 17 of the 19 abstracts which were accepted for presentation. Abstracts of talks by invited speakers (Andrew D. Gordon and Pierre Wolper) are also included.

I would like to thank:

- the members of the Organising Committee, the invited speakers, the authors of submitted papers and all the attendees for making this workshop a success;
- the Office of Naval Research (ONR), the Defence Evaluation and Research Agency (DERA) and Formal Systems for sponsoring the event; and

- the Oxford University Computing Laboratory (OUCL) for providing many facilities.

David Nowak April 2000

Organising Committee

Marta Kwiatkowska (University of Birmingham)
Michael Leuschel (University of Southampton)
David Nowak (Oxford University)
Joy Reed (Oxford Brookes University)
Mike Reed (Oxford University)
Bill Roscoe (Oxford University)
Ulrich Ultes-Nitsche (University of Southampton)
Irfan Zakiuddin (DERA)

Table of Contents

On the Use of Weak Automata for Deciding Linear Arithmetic with Integer and Real Variables*

Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper

Université de Liège,
Institut Montefiore, B28,
4000 Liège, Belgium
{boigelot,jodogne,pw}@montefiore.ulg.ac.be,
http://www.montefiore.ulg.ac.be/~{boigelot,jodogne,pw}/

Abstract. This paper considers finite-automata based algorithms for handling linear arithmetic with both real and integer variables. Previous work has shown that this theory can be dealt with by using finite automata on infinite words, but this involves some difficult and delicate to implement algorithms. The contribution of this paper is to show, using topological arguments, that only a restricted class of automata on infinite words are necessary for handling real and integer linear arithmetic. This allows the use of substantially simpler algorithms and opens the path to the implementation of a usable system for handling this combined theory.

1 Introduction

Among the techniques used to develop algorithms for deciding or checking logical formulas, finite automata have played an important role in a variety of cases. Classical examples are the use of infinite-word finite automata by Büchi [Büc62] for obtaining decision procedures for the first and second order monadic theories of one successor as well as the use of tree automata by Rabin [Rab69] for deciding the second-order monadic theory of n successors. More recent examples are the use of automata for obtaining decision and model checking procedures for temporal and modal logics [VW86a,VW86b,VW94,KVW00]. In this last setting, automata-based procedures have the advantage of moving the combinatorial aspects of the procedures to the context of automata, which are simple graph-like structures well adapted to algorithmic development. This separation of concerns between the logical and the algorithmic has been quite fruitful for instance in the implementation of model checkers for linear-time temporal logic [CVWY90,Hol97].

* This work was partially funded by a grant of the "Communauté française de Belgique - Direction de la recherche scientifique - Actions de recherche concertées".
To appear in proceedings IJCAR 2001, Springer-Verlag LNCS.

As already noticed by Büchi [Büc60,Büc62], automata-based approaches are not limited to sequential and modal logics, but can also be used for Presburger arithmetic. To achieve this, one adopts the usual encoding of integers in a base $r \geq 2$, thus representing an integer as a word over the alphabet $\{0, \ldots, r-1\}$. By extension, n-component integer vectors are represented by words over the alphabet $\{0, \ldots, r-1\}^n$ and a finite automaton operating over this alphabet represents a set of integer vectors. Given that addition and order are easily represented by finite automata and that these automata are closed under Boolean operations as well as projection, one easily obtains a decision procedure for Presburger arithmetic. This idea was first explored at the theoretical level, yielding for instance the very nice result that base-independent finite-automaton representable sets are exactly the Presburger sets [Cob69,Sem77,BHMV94]. Later, it has been proposed as a practical means of deciding and manipulating Presburger formulas [BC96,Boi98,SKR98,WB00]. The intuition behind this applied use of automata for Presburger arithmetic is that finite automata play with respect to Presburger arithmetic a role similar to the one of Binary Decision Diagrams (BDDs) with respect to Boolean logic. These ideas have been implemented in the LASH tool [LASH], which has been used successfully in the context of verifying systems with unbounded integer variables.

It almost immediately comes to mind that if a finite word over the alphabet $\{0, \ldots, r-1\}$ can represent an integer, an infinite word over the same alphabet extended with a fractional part separator (the usual dot) can represent a real number. Finite automata on infinite words can thus represent sets of real vectors, and serve as a means of obtaining a decision procedure for real additive arithmetic. Furthermore, since numbers with empty fractional parts can easily be recognized by automata, the same technique can be used to obtain a decision procedure for a theory combining the integers and the reals. This is not presently handled by any tool, but can be of practical use, for instance in the verification of timed systems using integer variables [BBR97]. However, turning this into an effective implemented system is not as easy as it might first seem. Indeed, projecting and complementing finite automata on infinite words is significantly more difficult than for automata on finite words. Projection yields nondeterministic automata and complementing or determinizing infinite-word automata is a notoriously difficult problem. A number of algorithms have been proposed for this [Büc62,SVW87,Saf88,KV97], but even though their theoretical complexity remains simply exponential as in the finite word case, it moves up from $2^{O(n)}$ to $2^{O(n \log n)}$ and none of the proposed algorithms are as easy to implement and fine-tune as the simple Rabin-Scott subset construction used in the finite-word case.

However, it is intuitively surprising that handling reals is so much more difficult than handling integers, especially in light of the fact that the usual polyhedra-based approach to handling arithmetic is both of lower complexity and easier to implement for the reals than for the integers [FR79]. One would expect that handling reals with automata should be no more difficult than handling integers[1]. The conclusion that comes out of these observations is that infinite-word automata constructed from linear arithmetic formulas must have a special structure that makes them easier to manipulate than general automata on infinite words. That this special structure exists and that it can exploited to obtain simpler algorithms is precisely the subject of this paper.

As a starting point, let us look at the topological characterization of the sets definable by linear arithmetic formulas. Let us first consider a formula involving solely real variables. If the formula is quantifier free, it is a Boolean combination of linear constraints and thus defines a set which is a finite Boolean combination of open and closed sets. Now, since real linear arithmetic admits quantifier elimination, the same property also holds for quantified formulas. Then, looking at classes of automata on infinite words, one notices that the most restricted one that can accept Boolean combinations of open and closed sets is the class of deterministic weak automata [SW74,Sta83]. These accept all ω-regular sets in the Borel class $F_\sigma \cap G_\delta$ and hence also finite Boolean combinations of open and closed sets. So, with some care about moving from the topology on vectors to the topology on their encoding as words, one can conclude that the sets representable by arithmetic formulas involving only real variables can always be accepted by deterministic weak automata on infinite words. If integers are also involved in the formula, there is no established quantifier elimination result for the combined theory and one cannot readily conclude the same. A first result in this paper closes this loophole. It establishes that sets definable by quantified linear arithmetic formulas involving both real and integer variables are within $F_\sigma \cap G_\delta$ and thus are representable by deterministic weak automata. Rather than using a quantifier elimination type argument to establish this, our proof relies on separating the integer and fractional parts of variables and on topological properties of $F_\sigma \cap G_\delta$.

The problematic part of the operations on automata needed to decide a first-order theory is the sequence of projections and complementations needed to eliminate a string of quantifiers alternating between existential and universal ones. The second result of this paper shows that for sets defined in linear arithmetic this can be done with constructions that are simple adaptations of the ones used for automata on finite words. Indeed, deterministic weak automata can be viewed as either Büchi or co-Büchi automata. The interesting fact is that co-Büchi automata can be determinized by the "breakpoint" construction [MH84,KV97], which basically amounts to a product of subset constructions. Thus, one has a simple construction to project and determinize a weak automaton, yielding a deterministic co-Büchi automaton, which is easily complemented into a deterministic Büchi automaton. In the general case, another round of projection will lead to a nondeterministic Büchi automaton, for which a general determinization procedure has to be used. However, we have the result that for automata obtained from linear arithmetic formulas, the represented sets stay

[1] Note that one cannot expect reals to be easier to handle with automata than integers since, by nature, this representation includes explicit information about the existence of integer values satisfying the represented formula.

within those accepted by deterministic weak automata. We prove that this implies that the automata obtained after determinization will always be weak.

Note that this cannot be directly concluded from the fact that the represented sets stay within those representable by deterministic weak automata. Indeed, even though the represented sets can be accepted by deterministic weak automata, the automata that are obtained by the determinization procedure might not have this form. Fortunately, we can prove that this is impossible. For this, we go back to the link between automata and the topology of the sets of infinite words they accept. The argument is that ω-regular sets in $F_\sigma \cap G_\delta$ have a topological property that forces the automata accepting them to be inherently weak, i.e. not to have strongly connected components containing both accepting and non accepting cycles.

As a consequence of our results, we obtain a decision procedure for the theory combining integer and real linear arithmetic that is suitable for implementation. The fact that this theory is decidable was known [BBR97], but the results of this paper move us much closer to an implemented tool that can handle it effectively.

2 Automata-Theoretic and Topological Background

In this section we recall some automata-theoretic and topological concepts that are used in the paper.

2.1 Automata on Infinite Words

An infinite word (or ω-word) w over an alphabet Σ is a mapping $w : \mathbb{N} \to \Sigma$ from the natural numbers to Σ. A Büchi automaton on infinite words is a five-tuple $A = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states;
- Σ is the input alphabet;
- δ is the transition function and is of the form $\delta : Q \times \Sigma \to 2^Q$ if the automaton is nondeterministic and of the form $\delta : Q \times \Sigma \to Q$ if the automaton is deterministic;
- q_0 is the initial state;
- F is a set of accepting states.

A run π of a Büchi automaton $A = (Q, \Sigma, \delta, q_0, F)$ on an ω-word w is a mapping $\pi : \mathbb{N} \to Q$ that satisfies the following conditions:

- $\pi(0) = q_0$, i.e. the run starts in the initial state;
- For all $i \geq 0$, $\pi(i + 1) \in \delta(\pi(i), w(i))$ (nondeterministic automata) or $\pi(i + 1) = \delta(\pi(i), w(i))$ (deterministic automata), i.e. the run respects the transition function.

Let $\mathit{inf}(\pi)$ be the set of states that occur infinitely often in a run π. A run π is said to be accepting if $\mathit{inf}(\pi) \cap F \neq \emptyset$. An ω-word w is accepted by a Büchi automaton if that automaton has some accepting run on w. The language $L_\omega(A)$ of infinite words defined by a Büchi automaton A is the set of ω-words it accepts.

A co-Büchi automaton is defined exactly as a Büchi automaton except that its accepting runs are those for which $\mathit{inf}(\pi) \cap F = \emptyset$.

We will also use the notion of *weak* automata [MSS86]. For a Büchi automaton $A = (Q, \Sigma, \delta, q_0, F)$ to be *weak*, there has to be a partition of its state set Q into disjoint subsets $Q_1, \ldots, Q_m$ such that

- for each of the Q_i either $Q_i \subseteq F$ or $Q_i \cap F = \emptyset$; and
- there is a partial order $\leq$ on the sets $Q_1, \ldots, Q_m$ such that for every $q \in Q_i$ and $q' \in Q_j$ for which, for some $a \in \Sigma$, $q' \in \delta(q, a)$ ($q' = \delta(q, a)$ in the deterministic case), $Q_j \leq Q_i$.

For more details, a survey of automata on infinite words can be found in [Tho90].

2.2 Topology

Given a set S, a distance $d(x, y)$ defined on this set induces a topology on subsets of S. A neighborhood $N_\varepsilon(x)$ of a point $x \in S$ is the set $N_\varepsilon(x) = \{y \mid d(x, y) < \varepsilon\}$. A set $C \subseteq S$ is said to be open if for all $x \in C$, there exists $\varepsilon > 0$ such that the neighborhood $N_\varepsilon(x)$ is contained in C. A closed set is a set whose complement with respect to S is open. We will be referring to the first few levels of the Borel hierarchy which are shown in Figure 1. The notations used are the following:

- F are the closed sets,
- G are the open sets,
- F_σ is the class of countable unions of closed sets,
- G_δ is the class of countable intersections of open sets,
- $F_{\sigma\delta}$ is the class of countable intersections of F_σ sets,
- $G_{\delta\sigma}$ is the class of countable unions of G_δ sets,
- $\mathcal{B}(X)$ represents the finite Boolean combinations of sets in X.

An arrow between classes indicates proper inclusion.

3 Topological Characterization of Arithmetic Sets

We consider the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$, where $+$ represents the predicate $x + y = z$. Since any linear equality or order constraint can be encoded into this theory, we refer to it as additive or linear arithmetic over the reals and integers. It is the extension of Presburger arithmetic that includes both real and integer variables. In this section, we prove that the sets representable in this theory belong to the topological class $F_\sigma \cap G_\delta$ defined relatively to the Euclidean distance between vectors. This result is formalized by the following theorem.

$$
\begin{tikzcd}[column sep=small]
 & \vdots & \\
F_{\sigma\delta} & & G_{\delta\sigma} \\
 & F_{\sigma\delta} \cap G_{\delta\sigma} \arrow[ul] \arrow[ur] & \\
 & \mathcal{B}(F_\sigma) = \mathcal{B}(G_\delta) \arrow[u] & \\
F_\sigma \arrow[ur] & & G_\delta \arrow[ul] \\
 & F_\sigma \cap G_\delta \arrow[ul] \arrow[ur] & \\
 & \mathcal{B}(F) = \mathcal{B}(G) \arrow[u] & \\
F \arrow[ur] & & G \arrow[ul] \\
 & F \cap G \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

Fig. 1. The first few levels of the Borel hierarchy.

Theorem 1. *Let $S \subseteq \mathbb{R}^n$, with $n > 0$, be a set defined in the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. This set belongs to the topological class $F_\sigma \cap G_\delta$ induced by the distance*

$$d(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^2 \right)^{1/2} .$$

Proof. Since $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ is closed under negation, it is actually sufficient to show that each formula of this theory defines a set that belongs to F_σ, i.e., a set that can be expressed as a countable union of closed sets.

Let φ be a formula of $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. To simplify our argument, we will assume that all free variables of φ are reals. This can be done without loss of generality since quantified variables can range over both $\mathbb{R}$ and $\mathbb{Z}$. We introduce $u < v$ as a shorthand for $u \leq v \wedge \neg(u = v)$.

The first step of our proof consists of modifying φ in the following way. We replace each variable x that appears in φ by two variables x_I and x_F representing respectively the integer and the fractional part of x. Formally, this operation replaces each occurrence in φ of a free variable x by the sum $x_I + x_F$ while adding to φ the constraints $0 \leq x_F$ and $x_F < 1$, and transforms the quantified variables of φ according to the following rules:

$$\begin{aligned}
(\exists x \in \mathbb{R})\phi &\longrightarrow (\exists x_I \in \mathbb{Z})(\exists x_F \in \mathbb{R})(0 \leq x_F \wedge x_F < 1 \wedge \phi[x/x_I + x_F]) \\
(\forall x \in \mathbb{R})\phi &\longrightarrow (\forall x_I \in \mathbb{Z})(\forall x_F \in \mathbb{R})(x_F < 0 \vee 1 \leq x_F \vee \phi[x/x_I + x_F]) \\
(Qx \in \mathbb{Z})\phi &\longrightarrow (Qx_I \in \mathbb{Z})\phi[x/x_I],
\end{aligned}$$

where $Q \in \{\exists, \forall\}$, ϕ is a subformula, and $\phi[x/y]$ denotes the result of replacing by y each occurrence of x in ϕ. The transformation has no influence on the set represented by φ, except that the integer and fractional part of each value are now represented by two distinct variables.

Now, the atomic formulas of φ are of the form $p = q + r$, $p = q$ or $p \leq q$, where p, q and r are either integer variables, sums of an integer and of a fractional variable, or integer constants. The second step consists of expanding these atomic formulas so as to send into distinct atoms the occurrences of the integer and of the fractional variables. This is easily done with the help of simple arithmetic rules, for the truth value of the atomic formulas that involve both types of variables has only to be preserved for values of the fractional variables that belong to the interval $[0, 1[$. The less trivial expansion rules[2] are given below:

$$\begin{aligned}
(x_I + x_F) = (y_I + y_F) &\longrightarrow x_I = y_I \wedge x_F = y_F \\
(x_I + x_F) \leq (y_I + y_F) &\longrightarrow x_I < y_I \vee (x_I = y_I \wedge x_F \leq y_F) \\
(x_I + x_F) = (y_I + y_F) + (z_I + z_F) &\longrightarrow (x_I = y_I + z_I \wedge x_F = y_F + z_F) \\
&\quad\ \vee (x_I = y_I + z_I + 1 \wedge x_F = y_F + z_F - 1) \\
(x_I + x_F) = (y_I + y_F) + z_I &\longrightarrow x_I = y_I + z_I \wedge x_F = y_F \\
x_I = (y_I + y_F) + (z_I + z_F) &\longrightarrow (x_I = y_I + z_I \wedge y_F + z_F = 0) \\
&\quad\ \vee (x_I = y_I + z_I + 1 \wedge y_F + z_F = 1)
\end{aligned}$$

After the transformation, each atomic formula of φ is either a formula ϕ_I involving only integer variables or a formula ϕ_F over fractional variables. We now distribute existential (resp. universal) quantifiers over disjunctions (resp. conjunctions), after rewriting their argument into disjunctive (resp. conjunctive) normal form, and then apply the simplification rules

$$\begin{aligned}
(Qx_I \in \mathbb{Z})(\phi_I \,\alpha\, \phi_F) &\longrightarrow (Qx_I \in \mathbb{Z})(\phi_I)\ \alpha\ \phi_F \\
(Qx_F \in \mathbb{R})(\phi_I \,\alpha\, \phi_F) &\longrightarrow \phi_I\ \alpha\ (Qx_F \in \mathbb{R})(\phi_F),
\end{aligned}$$

where $Q \in \{\exists, \forall\}$ and $\alpha \in \{\vee, \wedge\}$.

Repeating this operation, we eventually get a formula φ that takes the form of a finite Boolean combination $\mathcal{B}(\phi_I^{(1)}, \phi_I^{(2)}, \ldots, \phi_I^{(m)}, \phi_F^{(1)}, \phi_F^{(2)}, \ldots, \phi_F^{(m')})$ of subformulas $\phi_I^{(i)}$ and $\phi_F^{(i)}$ that involve respectively only integer and fractional variables.

[2] In these rules, the expression $p = q + r + s$ is introduced as a shorthand for $(\exists u)(u = q + r \wedge p = u + s)$, where the quantifier is defined over the appropriate domain.

Let $x_I^{(1)}, x_I^{(2)}, \ldots, x_I^{(k)}$ be the free integer variables of φ. For each assignment of values to these variables, the subformulas $\phi_I^{(i)}$ are each identically true or false, hence we have

$$\varphi \equiv \bigvee_{(a_1,\ldots,a_k)\in\mathbb{Z}^k} \Big((x_I^{(1)}, \ldots, x_I^{(k)}) = (a_1, \ldots, a_k) \wedge \mathcal{B}_{(a_1,\ldots,a_k)}(\phi_F^{(1)}, \ldots, \phi_F^{(m')}) \Big) .$$

Each subformula $\phi_F^{(i)}$ belongs to the theory $\langle \mathbb{R}, +, \leq, 1 \rangle$, which admits the elimination of quantifiers [FR79]. The sets of reals vectors satisfying these formulas are thus finite Boolean combinations of linear constraints with open or closed boundaries. It follows that, for each $(a_1, \ldots, a_k) \in \mathbb{Z}^k$, the set described by $\mathcal{B}_{(a_1,\ldots,a_k)}(\phi_F^{(1)}, \ldots, \phi_F^{(m')})$ is a finite Boolean combination of open and closed sets and, since any open set is a countable union of closed sets, is within F_σ. Therefore, the set described by φ is a countable union of F_σ sets and is also within F_σ.

4 Representing Sets of Integers and Reals with Finite Automata

In this section, we recall the finite-state representation of sets of real vectors as introduced in [BBR97].

In order to make a finite automaton recognize numbers, one needs to establish a mapping between these and words. Our encoding scheme corresponds to the usual notation for reals and relies on an arbitrary integer base $r > 1$. We encode a number x in base r, most significant digit first, by words of the form $w_I \star w_F$, where w_I encodes the integer part x_I of x as a finite word over $\{0, \ldots, r-1\}$, the special symbol "$\star$" is a separator, and w_F encodes the fractional part x_F of x as an infinite word over $\{0, \ldots, r-1\}$. Negative numbers are represented by their r's complement. The length p of $|w_I|$, which we refer to as the *integer-part length* of w, is not fixed but must be large enough for $-r^{p-1} \leq x_I < r^{p-1}$ to hold.

According to this scheme, each number has an infinite number of encodings, since their integer-part length can be increased unboundedly. In addition, the rational numbers whose denominator has only prime factors that are also factors of r have two distinct encodings with the same integer-part length. For example, in base 10, the number $11/2$ has the encodings $005 \star 5(0)^\omega$ and $005 \star 4(9)^\omega$, "ω" denoting infinite repetition.

To encode a vector of real numbers, we represent each of its components by words of identical integer-part length. This length can be chosen arbitrarily, provided that it is sufficient for encoding the vector component with the highest magnitude. An encoding of a vector $\boldsymbol{x} \in \mathbb{R}^n$ can indifferently be viewed either as a n-tuple of words of identical integer-part length over the alphabet $\{0, \ldots, r-1, \star\}$, or as a single word w over the alphabet $\{0, \ldots, r-1\}^n \cup \{\star\}$.

Since a real vector has an infinite number of possible encodings, we have to choose which of these the automata will recognize. A natural choice is to accept all encodings. This leads to the following definition.

Definition 1. *Let $n > 0$ and $r > 1$ be integers. A* Real Vector Automaton (RVA) *A in base r for vectors in $\mathbb{R}^n$ is a Büchi automaton over the alphabet $\{0, \ldots, r-1\}^n \cup \{\star\}$, such that*

- *Every word accepted by A is an encoding in base r of a vector in $\mathbb{R}^n$, and*
- *For every vector $\boldsymbol{x} \in \mathbb{R}^n$, A accepts either all the encodings of $\boldsymbol{x}$ in base r, or none of them.*

An RVA is said to *represent* the set of vectors encoded by the words that belong to its accepted language. Efficient algorithms have been developed for constructing RVA representing the sets of solutions of systems of linear equations and inequations [BRW98]. Since it is immediate to constrain a number to be an integer with an RVA and since, using existing algorithms for infinite-word automata, one can apply Boolean operations as well as projection to RVA, it follows that one can construct an RVA for any formula of the arithmetic theory we are considering.

5 Weak Automata and their Properties

If one examines the constructions given in [BRW98] to build RVA for linear equations and inequations, one notices that they have the property that all states within the same strongly connected component are either accepting or nonaccepting. This implies that these automata are *weak* in the sense of [MSS86] (see Section 2).

Weak automata have a number of interesting properties. A first one is that they can be represented both as Büchi and co-Büchi. Indeed, a weak Büchi automaton $A = (Q, \Sigma, \delta, q_0, F)$ is equivalent to the co-Büchi automaton $A = (Q, \Sigma, \delta, q_0, Q \setminus F)$, since a computation eventually remains within a single component Q_i in which all states have the same status with respect to being accepting. A consequence of this is that weak automata can be determinized by the fairly simple "breakpoint" construction [MH84,KV97] that can be used for co-Büchi automata. This construction is the following.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a nondeterministic co-Büchi automaton. The deterministic co-Büchi automaton $A' = (Q', \Sigma, \delta', q_0', F')$ defined as follows accepts the same ω-language.

- $Q' = 2^Q \times 2^Q$, the states of A' are pairs of sets of states of A.
- $q_0' = (\{q_0\}, \emptyset)$.
- For $(S, R) \in Q'$ and $a \in \Sigma$, the transition function is defined by
  - if $R = \emptyset$, then $\delta((S, R), a) = (T, T \setminus F)$ where $T = \{q \mid \exists p \in S \text{ and } q \in \delta(p, a)\}$, T is obtained from S as in the classical subset construction, and the second component of the pair of sets of states is obtained from T by eliminating states in F;
  - if $R \neq \emptyset$, then $\delta((S, R), a) = (T, U \setminus F)$ where $T = \{q \mid \exists p \in S \text{ and } q \in \delta(p, a)\}$, and $U = \{q \mid \exists p \in R \text{ and } q \in \delta(p, a)\}$, the subset construction set is now applied to both S and R and states in F are removed from U.

– $F' = 2^Q \times \{\emptyset\}$.

When the automaton A' is in a state (S, R), R represents the states of A that can be reached by a computation that has not gone through a state in F since that last "breakpoint", i.e. state of the form $(S, \emptyset)$. So, for a given word, A has a computation that does not go infinitely often through a state in F iff A' has a computation that does not go infinitely often through a state in F'. Notice that the difficulty that exists for determinizing Büchi automata, which is to make sure that the *same* computation repeatedly reaches an accepting state disappears since, for co-Büchi automata, we are just looking for a computation that eventually avoids accepting states.

It is interesting to notice that the construction implies that all reachable states (S, R) of A' satisfy $R \subseteq S$. The breakpoint construction can thus be implemented as a subset construction in which the states in R are simply tagged. One can thus expect it to behave in practice very similarly to the traditional subset construction for finite word automata.

Another property of weak automata that will be of particular interest to us is the topological characterization of the sets of words that they can accept. Consider the topology on the set of ω-words induced by the distance

$$d(w, w') = \begin{cases} \frac{1}{|common(w,w')|+1} & \text{if } w \neq w' \\ 0 & \text{if } w = w', \end{cases}$$

where $|common(w, w')|$ denotes the length of the longest common prefix of w and w'. In this topology, weak deterministic automata accept exactly the ω-regular languages that are in $F_\sigma \cap G_\delta$. This follows from the results on the Staiger-Wagner class of automata [SW74,Sta83], which coincides with the class of deterministic weak automata, as can be inferred from [SW74] and is shown explicitly in [MS97]. Given the result proved in Section 3, it is tempting to conclude that the encodings of sets definable in the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ can always be accepted by weak deterministic automata. This conclusion is correct, but requires shifting the result from the topology on numbers to the topology on words, which we will do in the next section. In the meantime, we need one more result in order to be able to benefit algorithmically from the fact that we are dealing with $F_\sigma \cap G_\delta$ sets, i.e. that any deterministic automaton accepting a $F_\sigma \cap G_\delta$ set is essentially a weak automaton.

Consider the following definition.

Definition 2. *A Büchi automaton is* inherently weak *if none of the reachable strongly connected components of its transition graph contains both accepting (including at least one accepting state) and non accepting (not including any accepting state) cycles.*

Clearly, if an automaton is inherently weak, it can directly be transformed into a weak automaton. The partition of the state set is its partition into strongly connected components and all the states of a component are made accepting or not, depending on whether the cycles in that component are accepting or not.

We will now prove the following.

Theorem 2. *Any deterministic Büchi automaton that accepts a language in $F_\sigma \cap G_\delta$ is inherently weak.*

To prove this, we use the fact that the language accepted by an automaton that is not inherently weak, must have the following *dense oscillating sequence* property.

Definition 3. *A language $L \subseteq \Sigma^\omega$ has the* dense oscillating sequence *property if, $w_1, w_2, w_3, \ldots$ being words and $\varepsilon_1, \varepsilon_2, \varepsilon_3, \ldots$ being distances, one has that $\exists w_1 \forall \varepsilon_1 \exists w_2 \forall \varepsilon_2 \ldots$ such that $d(w_i, w_{i+1}) \leq \varepsilon_i$ for all $i \geq 1$, $w_i \in L$ for all odd i, and $w_i \notin L$ for all even i.*

The fact that the language accepted by an automaton that is not inherently weak has the dense oscillating sequence property is an immediate consequence of the fact that such an automaton has a reachable strongly connected component containing both accepting and non accepting cycles. Given this, it is sufficient to prove the following lemma in order to establish Theorem 2.

Lemma 1. *An ω-regular language that has the dense oscillating sequence property cannot be accepted by a weak deterministic automaton and hence is not in $F_\sigma \cap G_\delta$.*

Proof. We proceed by contradiction. Assume that a language L having the dense oscillating sequence property is accepted by a weak deterministic automaton A. Consider the first word w_1 in a dense oscillating sequence for L. This word eventually reaches an accepting component Q_{i_1} of the partition of the state set of A and will stay within this component. Since ε_1 can be chosen freely, it can be taken small enough for the computation of A on w_2 to also reach the component Q_{i_1} before it starts to differ from w_1. Since w_2 is not in L, the computation of A on w_2 has to eventually leave the component Q_{i_1} and will eventually reach and stay within a non accepting component $Q_{i_2} < Q_{i_1}$. Repeating a similar argument, one can conclude that the computation of A on w_3 eventually reaches and stays within an accepting component $Q_{i_3} < Q_{i_2}$. Carrying on with this line of reasoning, one concludes that the state set of A must contain an infinite decreasing sequence of distinct components, which is impossible given that it is finite.

6 Deciding Linear Arithmetic with Real and Integer Variables

We first show that the result of Section 3 also applies to the sets of words encoding sets defined in $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$. In order to do so, we need to establish that the topological class $F_\sigma \cap G_\delta$ defined over sets of reals is mapped to its ω-word counterpart by the encoding relation described in Section 4.

Theorem 3. *Let $n > 0$ and $r > 1$ be integers, and let $L(S) \subseteq (\{0, \ldots, r-1\}^n \cup \{\star\})^\omega$ be the set of all the encodings in base r of the vectors belonging to the set $S \subseteq \mathbb{R}^n$. If the set S belongs to $F_\sigma \cap G_\delta$ (with respect to Euclidean distance), then the language $L(S)$ belongs to $F_\sigma \cap G_\delta$ (with respect to ω-word distance).*

Proof. Not all words over the alphabet $\{0,\ldots,r-1\}^n \cup \{\star\}$ encode a real vector. Let V be the set of all the valid encodings of vectors in base r. Its complement $\overline{V}$ can be partitioned into a set $\overline{V}_0$ containing only words in which the separator "$\star$" does not appear, and a set $\overline{V}_+$ containing words in which "$\star$" occurs at least once.

The set $\overline{V}_0 \cup V$ is closed. Indeed, each element of its complement is a word that does not encode validly a vector and that contains at least one separator. Such a word admits a neighborhood entirely composed of words satisfying the same property, which entails that the complement of $\overline{V}_0 \cup V$ if open. In the same way, one obtains that the set $\overline{V}_+ \cup V$ is open.

Let now consider an open set $S \subseteq \mathbb{R}^n$. The language $L' = L(S) \cup \overline{V}_+$ is open. Indeed, each word $w \in L(S)$ has a neighborhood entirely composed of words in $L(S)$ (formed by the encodings of vectors that belong to a neighborhood of the vector encoded by w), and of words that do not encode vectors but contain at least one separator. Moreover, each word $w \in \overline{V}_+$ admits a neighborhood fully composed of words in $\overline{V}_+$. Since $L(S) = L' \cap (\overline{V}_0 \cup V)$, we have that $L(S)$ is the intersection of an open and of a closed set.

The same line of reasoning can be followed with a closed set $S \subseteq \mathbb{R}^n$. The language $L'' = L(S) \cup \overline{V}_0$ is easily shown to be closed, which, since $L(S) = L'' \cap (\overline{V}_+ \cup V)$, implies that $L(S)$ is the intersection of a closed and of an open set.

We are now ready to address the case of a set $S \subseteq \mathbb{R}^n$ that belongs to $F_\sigma \cap G_\delta$. Since S is in F_σ, it can be expressed as a countable union of closed sets S_1, S_2, ... The languages $L(S_1)$, $L(S_2)$, ... are Boolean combinations of open and of closed sets, and thus belong to the topological class F_σ. Therefore, $L(S) = L(S_1) \cup L(S_2) \cup \cdots$ is a countable union of sets in F_σ, and thus belongs itself to F_σ. Now, since S is in G_δ, it can also be expressed as a countable intersection of open sets S'_1, S'_2, ... The languages $L(S'_1)$, $L(S'_2)$, ... belong to the topological class G_δ. Hence, $L(S) = L(S'_1) \cap L(S'_2) \cap \cdots$ is a countable intersection of sets in G_δ, and thus belongs itself to G_δ. This concludes our proof of the theorem.

Knowing that the encodings of sets definable in the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ are in $F_\sigma \cap G_\delta$, we use the results of Section 5 to conclude the following.

Theorem 4. *Every deterministic RVA representing a set definable in* $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ *is inherently weak.*

This property has the important consequence that the construction and the manipulation of RVA obtained from arithmetic formulas can be performed effectively by algorithms operating on weak automata. Precisely, to obtain an RVA for an arithmetic formula one can proceed as follows.

For equations and inequations, one uses the constructions given in [BRW98] to build weak RVA. Computing the intersection, union, and Cartesian product of sets represented by RVA simply reduces to performing similar operations with the languages accepted by the underlying automata, which can be done by simple product constructions. These operations preserve the weak nature of the automata. To complement a weak RVA, one determinizes it using the breakpoint construction, which is guaranteed to yield an inherently weak automaton (Theorem 4) that is easily converted to a weak one. This deterministic weak RVA is then complemented by inverting the accepting or non-accepting status of each of its components, and then removing from its accepted language the words that do not encode validly a vector (which is done by means of an intersection operation).

Applying an existential quantifier to a weak RVA is first done by removing from each transition label the symbol corresponding to the vector component that is projected out. This produces a non-deterministic weak automaton that may only accept some encodings of each vector in the quantified set, but generally not all of them. The second step thus consists of modifying the automaton so as to make it accept every encoding of each vector that it recognizes. Since different encodings of a same vector differ only in the number of times that their leading symbol is repeated, this operation can be carried out by the same procedure as the one used with finite-word number automata [Boi98]. This operation does not affect the weak nature of the automaton, which can then be determinized by the breakpoint construction, which has to produce an inherently weak RVA easily converted to a weak automaton.

Thus, in order to decide whether a formula of $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$ is satisfiable, one simply builds an RVA representing its set of solutions, and then check whether this automaton accepts a nonempty language. This also makes it possible to check the inclusion or the equivalence of sets represented by RVA. The main result of this paper is that, at every point, the constructed automaton remains weak and thus only the simple breakpoint construction is needed as a determinization procedure.

7 Conclusions

A probably unusual aspect of this paper is that it does not introduce new algorithms, but rather shows that existing algorithms can be used in a situation where *a priori* they could not be expected to operate correctly. To put it in other words, the contribution is not the algorithm but the proof of its correctness.

The critical reader might be wondering if all this is really necessary. After all, algorithms for complementing Büchi automata exist, either through determinization [Saf88] or directly [Büc62,SVW87,Kla91,KV97] and the more recent of these are even fairly simple and potentially implementable. There are no perfectly objective grounds on which to evaluate "simplicity" and "ease of implementation", but it is not difficult to convince oneself that the breakpoint construction for determinizing weak automata is simpler than anything proposed for determinizing or complementing Büchi automata. Indeed, it is but one step of the probably simplest complementation procedure proposed so far, that of [KV97]. Furthermore, there is a complexity improvement from $2^{O(n \log n)}$ to $2^{O(n)}$; experience with the subset construction as used for instance in the LASH tool [LASH] indicates that the breakpoint construction is likely to operate very well in practice; and being able to work with deterministic automata allows minimization [Löd01], which leads to a normal form.

An implementation and some experiments would of course substantiate the claims to simplicity and ease of implementation. It is planned in the context of the LASH tool and will be made available [LASH]. However, this paper is not about an implementation, but about the fact that, with the help of what might appear to be pure theory, one can obtain very interesting conclusions about algorithms for handling the theory $\langle \mathbb{R}, \mathbb{Z}, +, \leq \rangle$.

References

[BBR97] B. Boigelot, L. Bronne, and S. Rassart. An improved reachability analysis method for strongly linear hybrid systems. In *Proc. 9th Int. Conf.on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 167–178, Haifa, June 1997. Springer-Verlag.

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of CAAP'96*, number 1059 in Lecture Notes in Computer Science, pages 30–43. Springer-Verlag, 1996.

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bulletin of the Belgian Mathematical Society*, 1(2):191–238, March 1994.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.

[BRW98] Bernard Boigelot, Stéphane Rassart, and Pierre Wolper. On the expressiveness of real and integer arithmetic automata. In *Proc. 25th Colloq. on Automata, Programming, and Languages (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*, pages 152–163. Springer-Verlag, July 1998.

[Büc60] J. R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift Math. Logik und Grundlagen der Mathematik*, 6:66–92, 1960.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Cob69] A. Cobham. On the base-dependence of sets of numbers recognizable by finite automata. *Mathematical Systems Theory*, 3:186–192, 1969.

[CVWY90] Constantin Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242, Rutgers, June 1990. Springer-Verlag.

[FR79] J. Ferrante and C. W. Rackoff. *The Computational Complexity of Logical Theories*, volume 718 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin-Heidelberg-New York, 1979.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[Kla91] N. Klarlund. Progress measures for complementation of ω-automata with applications to temporal logic. In *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, San Juan, October 1991.

[KV97] O. Kupferman and M. Vardi. Weak alternating automata are not that weak. In *Proc. 5th Israeli Symposium on Theory of Computing and Systems*, pages 147–158. IEEE Computer Society Press, 1997.

[KVW00] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.

[LASH] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[Löd01] C. Löding. Efficient minimization of deterministic weak ω-automata, 2001. Submitted for publication.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[MS97] O. Maler and L. Staiger. On syntactic congruences for ω-languages. *Theoretical Computer Science*, 183(1):93–112, 1997.

[MSS86] D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th Int. Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1986.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, White Plains, October 1988.

[Sem77] A. L. Semenov. Presburgerness of predicates regular in two number systems. *Siberian Mathematical Journal*, 18:289–299, 1977.

[SKR98] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *Proceedings of the 10th Intl. Conf. on Computer-Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, Vancouver, June/July 1998. Springer-Verlag.

[Sta83] L. Staiger. Finite-state ω-languages. *Journal of Computer and System Sciences*, 27(3):434–448, 1983.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[SW74] L. Staiger and K. Wagner. Automatentheoretische und automatenfreie Charakterisierungen topologischer Klassen regulärer Folgenmengen. *Elektron. Informationsverarbeitung und Kybernetik EIK*, 10:379–392, 1974.

[Tho90] Wolfgang Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science – Volume B: Formal Models and Semantics*, chapter 4, pages 133–191. Elsevier, Amsterdam, 1990.

[VW86a] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW86b] Moshe Y. Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *Journal of Computer and System Science*, 32(2):183–221, April 1986.

[VW94] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[WB00] Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19, Berlin, March 2000. Springer-Verlag.

Authenticity by Typing for Security Protocols

Andrew D. Gordon
Microsoft Research

Alan Jeffrey
DePaul University

We propose a new method to check authenticity properties of cryptographic protocols. First, code up the protocol in the spi-calculus of Abadi and Gordon. Second, specify authenticity properties by annotating the code with correspondence assertions in the style of Woo and Lam. Third, figure out types for the keys, nonces, and messages of the protocol. Fourth, check that the spi-calculus code is well-typed according to a novel type and effect system presented in this paper. Our main theorem guarantees that any well-typed protocol is robustly safe, that is, its correspondence assertions are true in the presence of any opponent expressible in spi. It is feasible to apply this method by hand to several well-known cryptographic protocols. It requires little human effort per protocol, puts no bound on the size of the opponent, and requires no state space enumeration. Moreover, the types for protocol data provide some intuitive explanation of how the protocol works. Our method has led us to the independent rediscovery of flaws in existing protocols and to the design of improved protocols.

The talk describes our method and gives some simple examples. A paper describing the method will appear in the proceedings of the *14th IEEE Computer Security Foundations Workshop, June 11–13, 2001*, to be published by the IEEE Computer Society.

Reactive System Specification and Verification in RSDS

K. Androutsopoulos, K. Lano*

March 22, 2001

Abstract

In this paper we describe the RSDS method for automated control system specification, and the techniques used for modular decomposition and verification of reactive systems, involving a structure-preserving translation to SMV.

Keywords: Reactive system specification and design; control synthesis; SMV; B Method; RSDS

Specification in RSDS

RSDS (Reactive System Design Support) supports the specification of reactive systems using state machines and invariants. Its philosophy is to express such specifications in the simplest terms possible: no formal notation apart from these two notations are used, and automated support for analysis, design and code generation is built into the RSDS tool.

The use of invariants as the central means of specifying discrete control systems was suggested by the corresponding use of "transfer functions" in continuous control theory to express the required behaviour of a continuous controller [3]. In a discrete system such invariants are principally of three kinds:

1. Static (or 'safety') invariants: $\mathbf{P} \;\Rightarrow\; \mathbf{Q}$ where both $\mathbf{P}$ and $\mathbf{Q}$ consist only of constraints on the current states of sensors and actuators in the system. No temporal operators or event names are used. For example:

   ```
   sw = On & s2 = Off  =>  bm = On
   ```

   for the feed belt of the simple production cell [7], where **s2** is the blank sensor at the end of the belt and **bm** the belt motor (Figure 1).

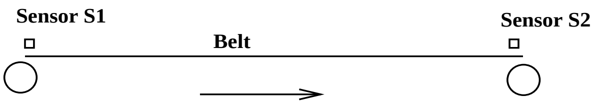


Figure 1: Feedbelt of Fault-tolerant Production Cell

2. Action (or 'operational') invariants α & $\mathbf{P} \;\Rightarrow\; \mathbf{AX}(\mathbf{Q})$ where $\mathbf{P}$ and $\mathbf{Q}$ are single-state constraints as in 1, and α an event of some sensor. **AX** is the branching temporal logic operator "in all next states". For example

   ```
   swon & sw = Off & s2 = Off  =>  AX(bm = On)
   ```

3. Temporal invariants $\mathbf{P} \;\Rightarrow\; \mathbf{AF}(\mathbf{Q})$ or $\mathbf{P} \;\Rightarrow\; \mathbf{AG}(\mathbf{Q})$. **AF** is "in some future state" and **AG** is "in all future states". For example

*Department of Computer Science, King's College London, Strand, London WC2R 2LS. `kcl@dcs.kcl.ac.uk`

```
s2 = Off & s1 = On & bm = On  =>  AF(s2 = On)
```

which expresses that blanks will eventually reach **s2** from **s1** if the belt motor is on.

Other temporal operators which are more rarely used are **EF**(**Q**) "there exists a path on which **Q** eventually holds" and **EG**(**Q**) "there exists a path on which **Q** always holds".

These types of invariant are currently processed by the RSDS tool. We also distinguish between *system* invariants, which are requirements on the control system developer, and *environment* invariants, which are assumptions about the behaviour of the equipment under control. The static and action invariants above are examples of system invariants, the temporal is an example of an environment invariant.

Statemachines for sensors and actuators can be selected from a set of standard components and may be edited by the user. Standard sensors include binary switch, toggle, linear measure, radio buttons, and standard actuators an On/Off component suitable for modelling bi-state devices, 3 state actuators (eg, for reversible motors), a timer, and a 4-state valve. More complex statemachines for custom components can be constructed by the user by using the statemachine editor.

The meaning of invariants of form 1 is that the formula must hold in all *stable* states of the complete system, ie, in all states where a reaction to an input event has been completed and there are no pending actuator commands. This corresponds to the states of a B machine [4] specification, where the reaction to an event is completed in 'the same step' as the event itself, intermediate states (between input event occurrences and the sending of commands to actuators in reaction to these occurrences) not being visible. Such invariants therefore also express requirements on the reactions to events which make their antecedents true. In the above example, events **swon** and **s2off** which make the antecedent true must be reacted to by a command **bmSetOn**. The tool derives such a reaction by generating action invariants from safety invariants, and then translating these into B or Java code. For example in the case of the static invariant in 1, we get the action invariants

```
swon & sw = Off & s2 = Off  =>  AX(bm = On)
s2off & s2 = On & sw = On  =>  AX(bm = On)
```

and B code:

```
swon = ...
  IF sw = Off & s2 = Off
  THEN bmSetOn
  ELSE ...
  END

s2off = ...
```

Environmental assumptions are separated from system invariants and are not used to generate action invariants or control code (since they do not describe obligations on the control system to achieve particular functionality or states). Environmental assumptions of the forms

$$\mathbf{P} \Rightarrow \mathbf{Q}$$

and

$$\alpha \Rightarrow \mathbf{Q}$$

where **P**, **Q** are sensor expressions and α is a sensor event, are used instead to generate fault detection code which is placed in a specialised fault detection controller **Fdc**. They are also used to reduce the state space of the generated controller state machine and the generated SMV modules.

DCFD views of a system are also presented, however these are purely to help in visualisation of system structure and have no semantic significance – all specification of a system is contained in the statemachines and invariants. In future versions of the tool UML class diagrams will be provided as a means of visualising controller structure.

1 Case Study: Fault Tolerant Production Cell

We will illustrate the use of RSDS on a simple example of an automated manufacturing system, a production cell containing feed and deposit belts to move work pieces into and out of the cell, robot arms for moving the pieces between the belts and a press, and a backup press to be used if the primary press fails [6].

Two subsystems of the production cell are the feedbelt control mechanism, consisting of switch **sw** to initiate processing, sensor **s1** at the start of the belt, a belt traffic light, a sensor **s2** at the end of the belt, and a motor **bm** to move the belt (Figure 1). All of these are two-state devices with states **On/Off**. The control invariants for the feedbelt are:

$$\mathbf{sw} = \mathbf{Off} \Rightarrow \mathbf{bm} = \mathbf{Off}$$
$$\mathrm{s2} = \mathbf{Off}\ \&\ \mathbf{sw} = \mathbf{On} \Rightarrow \mathbf{bm} = \mathbf{On}$$
$$\mathrm{s2} = \mathbf{On}\ \&\ (\mathbf{ts} = \mathbf{Off}\ \mathbf{or}\ \mathrm{s3} = \mathbf{On}) \Rightarrow \mathbf{bm} = \mathbf{Off}$$
$$\mathbf{sw} = \mathbf{On}\ \&\ \mathrm{s2} = \mathbf{On}\ \&\ \mathbf{ts} = \mathbf{On}\ \&\ \mathrm{s3} = \mathbf{Off} \Rightarrow \mathbf{bm} = \mathbf{On}$$

ts is the top position sensor of the elevating table subsystem which follows the feedbelt, and **s3** is the blank sensor of this subsystem. Blanks are transferred from the feedbelt to the table only when the table is in the top position and clear of a blank: **ts** = **On** & s3 = **Off**. Thus the feedbelt controller needs to know some information about the table state.

The table does not need to refer to the feedbelt state however. Its control invariants are:

$$\mathrm{s3} = \mathbf{On}\ \&\ \mathbf{bs} = \mathbf{Off} \Rightarrow \mathbf{tablemotor} = \mathbf{down}$$
$$\mathrm{s3} = \mathbf{On}\ \&\ \mathbf{bs} = \mathbf{On} \Rightarrow \mathbf{tablemotor} = \mathbf{Off}$$
$$\mathrm{s3} = \mathbf{Off}\ \&\ \mathbf{ts} = \mathbf{Off} \Rightarrow \mathbf{tablemotor} = \mathbf{up}$$
$$\mathrm{s3} = \mathbf{Off}\ \&\ \mathbf{ts} = \mathbf{On} \Rightarrow \mathbf{tablemotor} = \mathbf{Off}$$

The **tablemotor** is a reversible motor (Figure 2), **bs** is the bottom position sensor of the table and all sensors are two-state On/Off devices.

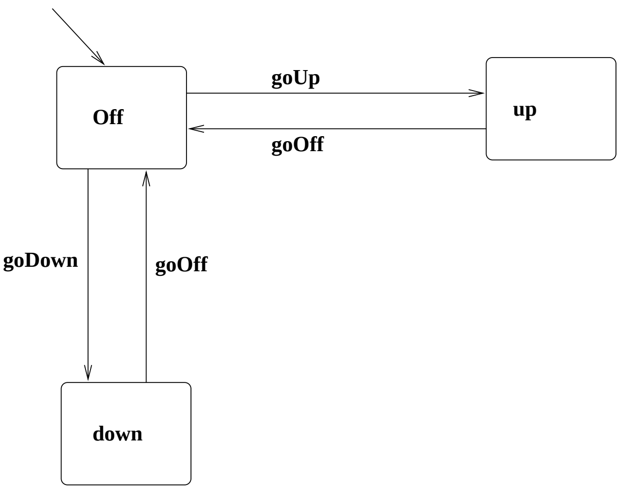


Figure 2: Statemachine of Table Motor

A static environmental assumption for the table is that not both of the table and motor sensors are **On** simultaneously:

$$\mathbf{ts} = \mathbf{On} \Rightarrow \mathbf{bs} = \mathbf{Off}$$

Temporal environmental assumptions are that blanks are only added when the table is in the top position:

$$\mathrm{s3on} \Rightarrow \mathbf{ts} = \mathbf{On}$$

and that if the table motor is set to **up**, the table will eventually reach the top position:

tablemotor = **up** $\Rightarrow$ **AF**(**ts** = **On**)

A temporal required system property which we would like to deduce about the table is that it regularly becomes available for receiving blanks if blanks are regularly removed:

s3 = **Off** $\Rightarrow$ **AF**(s3 = **Off** & **ts** = **On**)

We will show that this can be proved in SMV, assuming the above environmental properties.

2 Specification Analysis and Decomposition

The following techniques can be used to divide a reactive system specification into subsystems, synthesise executable code for these subsystems, and check consistency of the specification:

- Horizontal, phase and hierarchical decomposition
- Fault detection architecture construction
- Translation to Java code
- Consistency and type checks on invariants
- Completion of invariants: carrying out logical derivation to enlarge (and partially complete) a set of invariants.
- Abstraction construction between state machines.

Research is also ongoing to integrate synthesis techniques for schedulers into the control system synthesis process.

These facilities are described in the following sections.

2.1 Horizontal Decomposition

This design technique [5] splits a RSDS specification into two or more sub-specifications based on a partition of the set of actuators $\mathbf{A}$ into disjoint subsets $\mathbf{A}_1, \ldots, \mathbf{A_n}$ and the set of invariants $\mathbf{I}$ into disjoint subsets $\mathbf{I}_1, \ldots, \mathbf{I_n}$ where $\mathbf{I_j}$ only refers to actuators in the set $\mathbf{A_j}$.

Figure 3 gives a schematic structure of this type of decomposition.

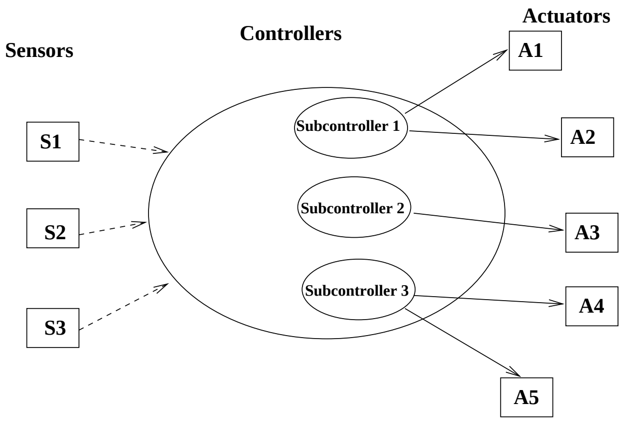


Figure 3: Horizontal Decomposition

RSDS provides proposals for horizontal decomposition based on data dependency analysis of invariants. This requires that the system invariants have been normalised, consistency and completeness checked, and dependencies computed. During dependency calculation groups of dependent variables are identified and the system invariants split into groups which are local to specific variable groups (ie, these invariants only involve components from the group). For example if we had the invariants

```
(1) sw = On & fd = present  =>  ig = Off
(2) sw = On & fd = absent  =>  ig = On
(3) sw = Off & fd = present  =>  av = open
(4) sw = Off & fd = absent  =>  av = closed
```

for a gas burner specification, where **fd** is a flame detector, **ig** an igniter, and **av** an air valve, then the dependencies are

$$\mathbf{sw}, \mathbf{fd} \longmapsto \mathbf{ig}$$
$$\mathbf{sw}, \mathbf{fd} \longmapsto \mathbf{av}$$

and so the variable groups are $\{\mathbf{sw}, \mathbf{fd}, \mathbf{ig}\}$ and $\{\mathbf{sw}, \mathbf{fd}, \mathbf{av}\}$. Invariants (1) and (2) belong to the first group, (3) and (4) to the second.

Choosing the option "Horizontal" (Decomposition) then presents a user with a list of all generated variable groups and asks the user to select a set of these.

The selection by the user must have the following properties:

1. No two groups can have a common actuator
2. All actuators of the main system must occur in (exactly) one group.

These properties are checked by the tool.

Once such a set of variable groups is selected the DCFD diagrams for these subsystems are automatically generated, and support all the operations of checking, derivation and synthesis that were possible with the original system. The subsystems can be developed independently.

The outer level of the complete system distributes events to all the new subcontrollers which have the event as an input.

2.2 Phase Decomposition

This design decomposition splits the specification into two or more subcases based on distinct phases or modes of behaviour, ie, different stages in the life cycle of the system. In a chemical plant for example one may have disjoint phases of filling a vessel, managing a reaction within that vessel, and emptying the vessel. In the production cell individual components such as the elevating table go through a sequence of such phases, from waiting for a blank to be delivered, through moving to the top or bottom position, and waiting for a blank to be removed. Each phase has a separate set of invariants, and a controller for this set can be developed. Once controllers for all phases exist, the system controller can be produced.

In RSDS we identify candidates for such a decomposition based on the range of possible values of a selected sensor. Phases are then identified with these possible values.

This requires that the system invariants have been normalised, consistency and completeness checked. The list of sensor variables which occur in a maximal number of invariants are presented to the user (in the above example these are **sw** and **fd** as both occur in 4 invariants). The user can select one and check its suitability for being the basis of a phase decomposition of the system over its values. The measure for this check is the number of actuator sharings between phase controllers which would result from this decomposition. In the above example selecting **fd** would give a measure of sharing equal to 2: both **ig** and **av** are shared between the $\mathbf{fd} = \mathbf{absent}$ and $\mathbf{fd} = \mathbf{present}$ phases. Selecting **sw** would give a sharing measure of 0 as there are no shared actuators between the $\mathbf{sw} = \mathbf{On}$ and $\mathbf{sw} = \mathbf{Off}$ phases. So **sw** would be selected for phase decomposition, and invariants generated for the new subcontrollers **CswOn** and **CswOff** for these phases:

```
fd = present  =>  ig = Off
fd = absent  =>  ig = On
```

for **CswOn**, and

```
fd = present  =>  av = open
fd = absent  =>  av = closed
```

for **CswOff**. The splitting of the DCFD into sub-DCFDs is then carried out as for horizontal decomposition.

2.3 Hierarchical Decomposition

This decomposition is a generalisation of horizontal decomposition in which there is a partition of the actuators $\mathbf{A}$ into sets $\mathbf{A}_1, \ldots, \mathbf{A_n}$, and we attempt to construct a similar partition of the sensors into groups $\mathbf{S}_1, \ldots, \mathbf{S_n}$ such that the sensors of $\mathbf{S_i}$ can be considered 'local' to the actuators of $\mathbf{A_i}$. Subcontrollers $\mathbf{C_i}$ which refer only to the values of the $\mathbf{S_i}$ sensors and $\mathbf{A_i}$ actuators are then constructed. Supervisor controllers may be needed between such $\mathbf{C_i}$ in order to inform one subsystem about the state of another.

In the production cell, the table can be controlled independently of the belt, and consists of the group $\{\mathbf{s3}, \mathbf{ts}, \mathbf{bs}\}$ of 'local' sensors for the actuator group $\{\mathbf{tablemotor}\}$. However the local sensors $\{\mathbf{sw}, \mathbf{s1}, \mathbf{s2}\}$ of the belt motor $\{\mathbf{bm}\}$ are insufficient to describe its control, so a supervisor controller is needed to communicate the truth or falsity of the condition $\mathbf{ts} = \mathbf{On}$ & $\mathbf{s3} = \mathbf{Off}$ "safe to deposit" to the belt controller. A hierarchical control structure can therefore be applied.

A series of heuristics are applied by the RSDS tool to identify the local variables of an actuator group, using data dependency analysis of the invariants.

2.4 Fault Detection Architecture

If there are environmental assumptions present then the user has the option to create fault detection rules based on these. There are two kinds of rule:

1. Static rules derived from static invariants (eg: $\mathbf{sw} = \mathbf{Off} \Rightarrow \mathbf{fd} = \mathbf{absent}$, or $\mathbf{ts} = \mathbf{On} \Rightarrow \mathbf{bs} = \mathbf{Off}$) which are checked in the outer level before any normal event detection:

   ```
   IF new_sw = On & new_fd = present THEN shutdown END
   ```

   The user must manually define a suitable **shutdown** operation in the main controller.

2. Dynamic rules derived from temporal invariants (eg: $\mathbf{swon} \Rightarrow \mathbf{fd} = \mathbf{absent}$ or $\mathbf{s3on} \Rightarrow \mathbf{ts} = \mathbf{On}$). These are checked by the **Fdc** as events are passed to it before forwarding to the main controller:

   ```
   fdc_swon = IF fd = absent THEN swon ELSE shutdown END
   ```

No representation of failure states is added to the system: it is assumed that the system invariants only relate to non-failed states and that calling **shutdown** terminates the control process.

2.5 Translation to Java

This option is available on the statemachine editors, where it produces static Java classes **Ms** for each sensor **s** and **Ma** and **Actuatora** for each actuator **a**. It is also provided on the DCFD editor, and produces Java classes **SystemTypes** and **Controller** corresponding to the B machines for the system types and system controller. Operational invariants must have been derived from any static invariants in order to enable this step.

2.6 Consistency and type checks of invariants

Consistency checks on safety invariants are carried out to ensure that contradictory requirements are identified. For example if we had written:

```
sw = On  =>  ig = On
fd = present  =>  ig = Off
```

then in the state where $\mathbf{sw} = \mathbf{On}$ & $\mathbf{fd} = \mathbf{present}$ there are contradictory requirements on **ig**. The invariants should be rewritten to make their antecedents disjoint in such cases.

Type-checking (during normalisation) checks that for each subformula $\mathbf{c} = \mathbf{val}$ of an invariant, **c** is the name of a component and **val** the name of one of its states. The 'modality' of the invariant is also computed (eg, **SENSEN**, **SENACT**, etc).

2.7 Completion of invariants

This step tries to identify further invariants which follow logically from the ones the user has entered. Any static invariant

$$\mathbf{C} \ \& \ \mathbf{a} = \mathbf{v} \ \Rightarrow \ \mathbf{a'} = \mathbf{v'}$$

where $\mathbf{a}$ and $\mathbf{a'}$ are actuators is used to generate a set of invariants

$$\mathbf{C} \ \& \ \mathbf{a'} = \mathbf{v_i} \ \Rightarrow \ \mathbf{a} = \mathbf{w}$$

where the $\mathbf{v_i}$ are the states of $\mathbf{a'}$ not equal to $\mathbf{v'}$, and $\mathbf{a}$ has 2 states $\mathbf{v}$ and $\mathbf{w}$.

Furthermore, from a pair of invariants

$$\mathbf{P} \ \Rightarrow \ \mathbf{a} = \mathbf{v}$$
$$\mathbf{Q} \ \& \ \mathbf{a} = \mathbf{v} \ \Rightarrow \ \mathbf{R}$$

we derive

$$\mathbf{P} \ \& \ \mathbf{Q} \ \Rightarrow \ \mathbf{R}$$

Redundant invariants are eliminated, ie, invariants

$$\mathbf{P} \ \Rightarrow \ \mathbf{Q}$$

where there is already an invariant

$$\mathbf{R} \ \Rightarrow \ \mathbf{Q}$$

in the system, with $\mathbf{P} \ \Rightarrow \ \mathbf{R}$.

Completeness checks such as that every actuator and sensor is referred to in at least one invariant, and that every actuator and sensor state is referred to in at least one invariant, are also carried out.

2.8 Abstraction Mappings

This option allows a user to select two statemachines (components) and check for the existence of an abstraction morphism between them. In this case the two components must have compatible sets of events (the events of the refinement are a subset of those of the abstract component) and only one will actually be included in the system as a whole. This option is used to prove that a component or flattened composition of components conforms to some more abstract safety or behavioural model.

2.9 Translation to SMV

A translation to SMV has been defined which expresses RSDS actuators, controllers and sensors as separate SMV modules and allows temporal properties to be verified or counter-examples generated.

The general form of the SMV modules generated from an RSDS specification is as follows. For the controller of a system consisting of sensors $\mathbf{S}_1, \ldots, \mathbf{S_n}$ we construct a controller state machine $\mathbf{C}$ whose states are all tuples $(\mathbf{x}_1, \ldots, \mathbf{x_n})$ of states $\mathbf{x}_1 \in \mathbf{States}_{\mathbf{S}_1}, \ldots, \mathbf{x_n} \in \mathbf{States}_{\mathbf{S_n}}$. The transitions of $\mathbf{C}$ are the transitions of the AND-composition of $\mathbf{S}_1, \ldots, \mathbf{S_n}$. From the system invariants we derive the actuator settings which must hold in each state of $\mathbf{C}$ (the RSDS specification is incomplete if there is not such a unique setting for each state of $\mathbf{C}$).

The SMV module for $\mathbf{C}$ is then generated from its statemachine:

```
MODULE Controller
VAR
  global_state: { c1, ..., cm };
  con_event: { e1, ..., er };
DEFINE
```

```
  CT1 := con_event = e1 &
         global_state = c2;
  ...
ASSIGN
  init(global_state) := c_init;

  next(global_state) :=
    case
      CT1: ct1;
      ...
      1: global_state;
    esac;
```

The **CT** represent all transitions of **C**, eg, the transition for $\mathbf{e}_1 : \mathbf{c}_2 \rightarrow \mathbf{ct}_1$ in the above schema.

Sensors are represented by modules

```
MODULE S1(C)
VAR
  s1: { ss1, ..., ssp };
DEFINE
  s1T0 := C.con_event = s1on & s1 = ss1;
  ...
ASSIGN
  init(s1) := s1_init;

  next(s1) :=
    case
      s1T0: st1;
      ...
      1: s1;
    esac;
```

Each controller event which is also an event of $\mathbf{S}_1$ is used in a definition for the LHS of a case in the **next** statement. The corresponding RHS is the projection of the resulting global state onto $\mathbf{S}_1$.

For actuators the same pattern is followed, except that the **next** statement uses the actuator settings which results from each global transition.

Finally there is a **main** module which combines the separate component modules and links them together:

```
MODULE main
VAR
  C: Controller;
  Ms1: S1(C);
  ..
SPEC AG(AG(Env) -> T1)
...
SPEC AG(AG(Env) -> Tr)
```

Env is the conjunction of environmental assumptions. The $\mathbf{T_i}$ are all temporal system invariants. The SPEC clauses of **main** must evaluate to true for the specification to be correct.

For the elevating table, SMV code can be generated as follows:

```
MODULE main
VAR
  C : Controller;
  Mbs : bs(C);
  Mts : ts(C);
  Ms3 : s3(C);
  Mtablemotor : tablemotor(C);
SPEC
```

```
  AG(AG(C.con_event = s3on -> Mts.ts = On) &
     AG(Mtablemotor.tablemotor = up -> AF(Mts.ts = On))  ->
            (Ms3.s3 = Off  ->  AF(Ms3.s3 = Off & Mts.ts = On)) )

MODULE s3(C)
VAR
  s3 : { Off, On };
DEFINE
  s3T0 := C.con_event = s3on & s3 = Off;
  s3T1 := C.con_event = s3off & s3 = On;
ASSIGN
  init(s3) := Off;

  next(s3) :=
    case
      s3T0 : On;
      s3T1 : Off;
      1 : s3;
    esac;

MODULE tablemotor(C)
VAR
  tablemotor : { Off, up, down };
ASSIGN
  init(tablemotor) := down;

  next(tablemotor) :=
    case
      C.CT0 : up;
      C.CT1 : Off;
      ...
      C.CT23 : Off;
      1 : tablemotor;
    esac;


MODULE Controller
VAR
  global_state : { Off_Off_Off, Off_Off_On, Off_On_Off, Off_On_On, On_Off_Off,
         On_Off_On, On_On_Off, On_On_On };
  con_event : { bson, bsoff, tson, tsoff, s3on, s3off };
DEFINE
  CT0 := con_event = bson & global_state = Off_Off_Off;
  CT1 := con_event = bson & global_state = Off_Off_On;
  ...
  CT23 := con_event = s3off & global_state = On_On_On;
ASSIGN
  init(global_state) := Off_Off_Off;

  next(global_state) :=
    case
      CT0 : On_Off_Off;
      CT1 : On_Off_On;
      ...
      CT23 : On_On_Off;
      1 : global_state;
    esac;
```

The temporal properties expressed in **main** are verified by the SMV analysis tool, proving the required liveness property.

3 Comparison with Related Work

The SCR method [2] addresses the specification construction and specification analysis of reactive control systems. It uses a series of tables to define how actuators should change state when a change in sensor state occurs: these are essentially the same information as contained in our action invariants. As in RSDS, variable dependency analysis is used to guide possible decomposition strategies, although SCR has not formalised the different forms of decomposition. Tabular description can be difficult for non-software specialists to understand, hence RSDS uses the more widely familiar engineering notation of state machines. SCR deduces static invariants from action invariants by using a state-space exploration algorithm, whereas in RSDS the reverse process is more usual, since static invariants are more abstract than action invariants. A comparative case study of RSDS and SCR has shown that the RSDS approach can produce more compact and abstract specifications than SCR, with a smaller state space and hence reduced cost of verification [1].

4 Conclusion

RSDS has been applied to a wide range of discrete event control system problems and has been shown to be a practical approach for design of small to medium scale control systems. Current work aims to integrate the current tool support with hazard and fault-tree analysis tools for automated control systems, in order to provide an integrated environment for the design of safety-critical systems following standards such as IEC 61508.

References

[1] K. Androutsopoulos, *Comparison of SCR Requirements Specification Method with RSDS*, Department of Computer Science, King's College, 2001. `http://www.dcs.kcl.ac.uk/pg/kelly/`

[2] R. Bharadwaj, C. Heitmeyer, *Model checking complete requirements specifications using abstraction.* In *Proceedings of Automated Software Engineering*, 6, 37–68, 1999.

[3] S. Goldsack and J. Kramer, *Invariants in the Application-oriented Specification of Control Systems*, Automatica, Vol. 18, No. 2, pp. 71 – 76, 1982.

[4] K Lano. *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag, 1996.

[5] K. Lano, K. Androutsopoulos, D. Clark, *Structuring and Design of Reactive Systems using RSDS and B*, FASE 2000, LNCS, Springer-Verlag, 2000.

[6] K. Lano, D. Clark, K. Androutsopoulos, P. Kan, *Invariant-based Synthesis of Fault-tolerant Systems*, FTRTFT 2000, Pune, India, 2000.

[7] C Lewerentz, T Lindner (Editors). *Case Study "Production Cell": A comparative study in formal software development*, FZI Publication 1/94, University of Karlsruhe, 1994. Also in LNCS, Springer-Verlag, 1996.

[8] A. Lötzbeyer, R. Mühlfeld. *Task Description of a Flexible Production Cell with Real Time Properties*, FZI, Karlsruhe, 1996.

Performance considerations on the WAP secure transport layer

Linda Brodo and Corrado Priami
Dipartimento Scientifico Tecnologico, Università di Verona
Ca' Vignal 2, Strada Le Grazie 15, I-37134 Verona, Italy
{brodo,priami}@sci.univr.it

Abstract

This paper applies the enhanced operational semantics framework to predict performance of the WTLS layer of the WAP reference model. The case study allows to single out a trade-off between security and performance which is a key ingredient in the design of wireless telecommunication applications.

Keywords. Performance prediction, Software verification, Security, Case studies.

1 Introduction

The emerging wireless scenario in which portable and programmable (handheld) terminals are connected to Internet backbones poses new challenges in the design of applications for this highly diverse and geographically dispersed computational environments. Key ingredients for these *global* applications are surely performance prediction and security issues.

Since the user is presumably charged when using mobile Internet added-value services and wireless connections have lower bandwidth then wired ones, it is essential to provide efficient applications to minimize costs. Another essential feature to be considered when estimating performance is the battery life of the portable terminals: sometimes the user is willing to pay more in order to be connected a longer time. This again means less the elaboration and communication time on the portable terminal, the better.

As far as security is concerned, the effort of EC in moving rapidly from e-commerce to m(obile)-commerce makes security analysis in wireless networks an highest priority research theme. Unfortunately, secure communications usually relies on cryptographic protocols that may involve expensive computational tasks on the mobile terminal (encryption and decryption at least). A security policy to be effective in a wireless world must be a reasonable trade-off between computational costs and expected security guarantees. The design choices on this trade-off must be taken in the early phases of global application development. In fact, a bad choice could lead to a definitive failure of the whole project due to highly changing market place of wireless technology and applications.

In the above scenario we propose a framework to carry out performance and security analysis over a uniform model that represents the expected behaviour of global applications. More precisely, we adopt the enhanced operational semantics (EOS for short) of process calculi [14, 13]: an extension of the SOS approach [23] in which the labels of transitions are encodings of their proofs in the formal system of axioms and inference rules defining the semantics of the language. The enhanced labels carry over a very concrete description of the corresponding action: it is possible to identify the syntactic instruction (prefix) of the calculus which originated the action. Furthermore, the operational semantics defines an abstract machine for the language such that the application of a rule in the derivation of a transition can be interpreted as a low level routine performed by the run-time to execute the action corresponding to the transition. Therefore the enhanced labels can be viewed as encodings of the corresponding set of low level routines.

The above idea allows us to associate probabilistic distributions to transitions simply by looking at their labels [22, 26] and without altering the syntax of the original calculus as usually happens in Markovian and stochastic process algebras [16, 17, 7, 10, 18, 11, 19, 3, 9, 24, 25, 5, 6]. If we limit distributions to exponential ones, we can derive a continuous time Markov chain from a transition system and then carry out quantitative analysis in the same way of Markovian process algebras.

We implemented a tool (EOS workbench) [8] that produces transition systems with enhanced labels for π-calculus processes [21]. The internal representation is based on the FC2 format [20] to easy the exchange of data with other process calculi tools. A module of the EOS workbench is in charge of associating exponential distributions to transitions according to a description of the target architecture including the costs [1] of the low-level routines. Then the continuous time Markov chain is obtained from the re-labelled transition system and quantitative analysis is carried out using standard mathematical tools.

We here apply the EOS framework to the analysis of the wireless transport layer security specification (WTLS) of the wireless application protocol (WAP) [2]. In the WAP reference model security is assumed to be an optional layer above the transport layer and therefore it must preserve the transport services interface to the upper layers. This optional layer is WTLS and it is designed to function both on connection-oriented and datagram transport protocols. The WTLS connection management allows a client to connect with a server in order to establish a secure connection through a negotiation phase, a key exchange and an authentication procedure. The security parameters involved in the negotiation are for instance cryptographic algorithms and key length.

Since the WTLS is an optional layer, it is essential to estimate the performance degradation due to its inclusion depending on the application requirements. Furthermore, once decided to rely on a WTLS implementation it is a critical task for the application client to choose adequately the characteristics of the security policy to be negotiated. We claim that performance prediction at specification level can be of great help in driving the design process and the above choices.

In the paper we present a formal specification of the WTLS layer in the π-calculus and we then use our EOS workbench to find some performance prediction measures that can help designer in deciding whether WTLS is acceptable in the implementation and eventually in selecting the appropriate parameters in the negotiation phase. To carry out the analysis we assume to run the WAP architecture over a GPRS bearer [1].

The above case study is an assessment of the feasibility of the approach we propose to estimate performance since the early stage of application design, and it is also a suitable example of application scenarios for formal verification and analysis of complex software systems and protocols.

The paper is organized as follows. In the next section we briefly recall the π-calculus and the main features of our EOS workbench. Section 3 reports a formal specification of the WTLS protocol, and Sect. 4 describes the preliminary analyses that we carried out with the EOS workbench.

2 Background

In this section we recall the π-calculus [21] and its enhanced operational semantics [15]. We then describe the essential features of our EOS workbench [8].

2.1 The π-calculus

We start with the syntax.

Definition 2.1 *Let $\mathcal{N} \ni a, b, x, y, \ldots$ be the set of* names*; let $\mathcal{S} \ni \tau_0, \tau_1, \ldots$ be the set of* invisible actions*, with $\mathcal{N} \cap \mathcal{S} = \emptyset$; let $\mathcal{I} \ni A, A_1, \ldots$ be the set of* agent identifiers. Processes *in $\mathcal{P} \ni P, Q, R, \ldots$ are defined as*

$$P ::= \mathbf{0} \mid X \mid \pi.P \mid (\nu x)P \mid [x = y]P \mid P|P \mid P + P \mid A(y_1, \ldots, y_n)$$

[1] We use here the word cost to mean any quantitative measure affecting the time or speed of execution of actions.

where π *may be either* $x(y)$ *for* input, *or* $\overline{x}y$ *for* output (*where* x *is the* subject *and* y *is the* object) *or* τ_i *for* silent *moves. The* precedence *among the operators is the same as above (from left to right). Hereafter, we will omit the trailing* $\mathbf{0}$.

In the above definition we used a set of silent moves to distinguish their different durations. We sometimes write $(\nu x, y)P$ for $(\nu x)(\nu y)P$. Each agent identifier A has a unique defining equation of the form $A(\tilde{y}) = P$ (hereafter, $\tilde{y}$ denotes $y_1, \ldots, y_n$), where the y_i are all distinct and are the only free names in P.

The operational semantics of the π-calculus is defined in the *SOS* style. We use μ as a metavariable for the labels of transitions. We introduce the set $\mathcal{A}$ of visible actions ranged over by α (i.e. $x(y)$ for input, $\overline{x}y$ for free output, and $\overline{x}(y)$ for bound output).

Names of labels and processes, as well as free and bound names, are as usual; they are computed by functions *fn*, *bn* and n, respectively. We define the *structural congruence* $\equiv$ on processes as the least congruence equating α-equivalent processes, and exchanging name binders ν, i.e. $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.

We define our *enhanced labels* as in [12, 4, 15]. The label of a transition records the inference rules used during its deduction, besides the action itself. Such an encoding is called *proof term*. Finally, we introduce a function ℓ that takes an enhanced label to the corresponding standard action label.

Definition 2.2 *Let* $\mathcal{L} = \{\|_0, \|_1\}$ *with* $\chi \in \mathcal{L}^*$, $\mathcal{O} = \{+_0, +_1, =_m, (\nu x), (\tilde{y})\}$ *with* $o \in \mathcal{O} \cup \mathcal{L}$ *and let* $\vartheta \in (\mathcal{L} \cup \mathcal{O})^*$. *Then the set* $\Theta \ni \theta$ *of* enhanced labels *is defined by the following syntax, where* $\alpha_0 = x(y)$ *iff* α_1 *is either* $\overline{x}y$ *or* $\overline{x}(y)$*, and vice versa:*

$$\theta ::= \vartheta\alpha \mid \vartheta\tau_i \mid \vartheta\langle\|_0\vartheta_0\alpha_0, \|_1\vartheta_1\alpha_1\rangle.$$

Function ℓ *is defined as* $\ell(\vartheta\alpha) = \alpha$, $\ell(\vartheta\tau_i) = \ell(\vartheta\langle\|_0\vartheta_0\alpha_0, \|_1\vartheta_1\alpha_1\rangle) = \tau$.

A tag $+_0$ ($+_1$) means that a nondeterministic choice has been resolved in favour of the left (right) component. Similarly, a tag $\|_0$ ($\|_1$) records that the left (right) component of a parallel composition is moving. The tag (νx) records that an environment was present. The resolution of a matching inserts a tag $=_m$, where m is the size of the data to be compared. Communications are labelled by a pair instead of a τ to show the components which interacted (and the proof of the relevant transitions). We also record in the labels the actual parameters $\tilde{y}$ of a definition because their number and size affect the instantiation cost.

Our transition system for the π-calculus is in Tab. 1; for intuition see [15]. The rules Par_1, Sum_1, Com_1 and $Close_1$ are symmetric and are omitted.

The standard semantics of the π-calculus is obtained by relabelling each transition in Tab. 1 through function ℓ in Def. 2.2.

2.2 The EOS workbench

We briefly recall in this section the main features of a tool to study the performance of distributed and mobile systems [8]. The tool is made up of two logical modules. The former is an interpreter of the π-calculus driven by the enhanced operational semantics [13]. The interpreter produces a labeled transition system (called proved) where the transitions are labeled by encoding of their proofs (enhanced labels). The latter module is a performance analyzer which uses architectural information to assign costs to transitions. Recall that we call here cost a measure of how fast an action is. Costs can be calculated using both the enhanced labels and some features of the actual architecture on which the program runs.

2.2.1 The interpreter

The interpreter takes as input a π-calculus process and returns as output the corresponding proved transition system, stored in a FC2 file [20] that Autograph [27] can visualize. The implementation

$$Act : \pi.P \xrightarrow{\pi} P \qquad Ide : \frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\theta} P'}{Q(\tilde{y}) \xrightarrow{(\tilde{y})\theta} P'}, Q(\tilde{x}) = P$$

$$Par_0 : \frac{P \xrightarrow{\theta} P'}{P|Q \xrightarrow{||_0\theta} P'|Q}, bn(\ell(\theta)) \cap fn(Q) = \emptyset \qquad Sum_0 : \frac{P \xrightarrow{\theta} P'}{P + Q \xrightarrow{+_0\theta} P'}$$

$$Com_0 : \frac{P \xrightarrow{\vartheta\overline{x}y} P', Q \xrightarrow{\vartheta' x(w)} Q'}{P|Q \xrightarrow{\langle ||_0\vartheta\overline{x}y, ||_1\vartheta' x(y)\rangle} P'|Q'\{y/w\}} \qquad Mat : \frac{P \xrightarrow{\theta} P'}{[x = x]P \xrightarrow{=_m\theta} P'}, size(x) = m$$

$$Res : \frac{P \xrightarrow{\theta} P'}{(\nu x)P \xrightarrow{(\nu x)\theta} (\nu x)P'}, x \notin n(\ell(\theta)) \qquad Open : \frac{P \xrightarrow{\vartheta\overline{x}y} P'}{(\nu y)P \xrightarrow{\vartheta\overline{x}(y)} P'}, x \neq y$$

$$Close_0 : \frac{P \xrightarrow{\vartheta\overline{x}(y)} P', Q \xrightarrow{\vartheta' x(w)} Q'}{P|Q \xrightarrow{\langle ||_0\vartheta\overline{x}(y), ||_1\vartheta' x(y)\rangle} (\nu y)(P'|Q'\{y/w\})}, \ y \notin fn(Q)$$

Table 1: Proved transition system for the π-calculus.

of the tool is in a proof-of-concept style and thus focuses on the feasibility of the functional tasks that we implement, rather than on their efficiency.

The kernel of the interpreter is the PTS constructor module that recursively builds the proved transition system. An arc is identified by the pair enhanced label and target node. After the execution of a prefix, names can change their nature and only a part of the transition system is usually interested in these changes. The name changing are stored in a data structure called *Local Environment*, associated to every node of the graph. The initial configuration of names is built by a LexYacc based file interpreter and is stored in a global table, called *Global Environment* which is visible by all nodes, except for the possible shadowing due to the *Local Environment*.

2.2.2 The analyzer

The analyzer takes his input data from two separate text files. One file is the representation of the proved transition system generated by the LexYacc based interpreter. The other file contains the measures which represent the programmer's knowledge about the actual underlying architecture used to execute the program. To obtain performance indexes, the analyzer associates costs to enhanced labels relying on the architectural cost file.

The costs are used to build the matrix of the continuous time Markov chain associated to the PTS. Then the user must specify by a composition of enhanced tags the resource of which s/he wants to study the performance. A resource can be, for example, a specific connection (as in the previous example), the memory, a specific node of the network. Then the reward vector is calculated. So the matrix and the vector are the input for a mathematical tool (Mathematica or Mathlab) to get the final result.

3 Specification of the WTLS

The Wireless Transport Layer Security (WTLS) is the security layer of the Wireless Application Protocol (WAP). The primary goal of the WTLS layer is to provide privacy, data integrity and authentication between two communicating applications [2]. The WTLS protocol is composed by three sub-protocols which are used to allow peers to agree upon security parameters to authenticate themselves, to instantiate security parameters and to report error conditions to each other.

We model the handshake protocol, the main sub-protocol, with a server and a client that want to establish a secure connection. We report here a simplified description to help the intuition and then we model below the whole sub-protocol in the π-calculus.

When both the server and the client are in the *NULL* state (see Fig. 1) and no communication is going to start, the server sends a *sHello* message to keep in touch with the client and waits for an answer by the client. If the client wants to open a secure connection sends a *cHello* (entering the state CREATING) to the server which answers with a *sHello* message (moving to CREATING). After the hello messages exchange, the server (passed to the state *CREATED*) sends a message indicating that the hello-phase is closed. The server also indicates if a full handshake or an optimized handshake is to be done moving to EXCHANGE or COMMIT, respectively. This choice is actually done by a configuration parameter. Then the client sends the *KExchange* message to comunicate the key exchange parameter reaching COMMIT1, while the server enters OPENING. At this point the client sends the *CipherSpec* message followed by the new secret channel named *secret*, which will be used to exchange secret data, and enters state COMMIT2. Now the server knows the secret channel and replies to the client with a *finish* message and the two peers enter the OPEN state. When both the client and the server are in their respective *OPEN* state, they share a *secret* channel which is used for the data transmission. Whenever the client sends a new *cHello* message, a new handshake has to start for generating a new *secret* channel.

In every phase of the protocol both server and client can send an *alert* or a *warning* message to inform the other peer that something bad is happened. In the first case both return in the NULL state, in the other case both remain in the same state in which they are.

Before reporting the π-calculus specification of WTLS, some comments on names and notation are mandatory. We implement any state of the transition diagram in Fig. 1 with a π-calculus agent with the same name. We only add the suffixes 'c' and 's' to distinguish client states and server states. For the sake of readability, the parameters of the agent definitions and of the agent calls are reduced to the ones meaningful for the study at hand. The *alert* and *warning* messages are specified only in the NULL and CREATING states (in the other states these messages are omitted for readability reasons). The channel *ch* is used by the peers to communicate before establishing a secure connection. The channel *KFull* (*KOpt*) is used when a full (optimized) handshake is performed. The *secret* channel is used for the (secure) data transmission between the applications.

The π-calculus specification that captures the abstract control structure and message flow of the protocol follows.

Client States

NULLc(CHello,HRequest,SHello,ch) =

$\overline{ch}$CHello.**CREATINGc**(CHello,HRequest,SHello,ch)
(*the client informs the server that wants to open a secure connection*)
+
$\overline{ch}$ alert.**NULLc**(CHello,HRequest,SHello,ch)
ch(msg).([msg=HRequest] $\overline{ch}$ CHello.**NULLc**(CHello,HRequest,SHello,ch)
+
[msg=alert]**NULLc**(CHello,HRequest,SHello,ch)
)

CREATINGc(CHello,HRequest,SHello,ch,KFull,KOpt)=

ch(msg).([msg=SHello] ch(msg1).([msg1=KFull]**EXCHANGEc**(CHello,HRequest,SHello,ch,KFull,KOpt)
+
[msg1=KOpt]**CREATEDc**(CHello,HRequest,SHello,ch,KFull,KOpt)
)
+
[msg=alert]**NULLc**(CHello,HRequest,SHello,ch,KFull,KOpt)
+
[msg=warning]**CREATINGc**(CHello,HRequest,SHello,ch,KFull,KOpt)
)
+
$\overline{ch}$ alert.**NULLc**(CHello,HRequest,SHello,ch,KFull,KOpt)
+
$\overline{ch}$ warning.**CREATINGc**(CHello,HRequest,SHello,ch,KFull,KOpt)

EXCHANGEc(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)=

$\overline{KFull}$CHello.**EXCHANGEc**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)
+
$\overline{KFull}$ CHello2.**CREATINGc**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)
+
$\overline{KFull}$KExchange.**COMMIT1c**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)
+
KFull(msg)[msg=CHello]
EXCHANGEc(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)

COMMIT1c(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

$\overline{KFull}$ CipherSpec.(ν secret)$\overline{KFull}$ secret.
COMMIT2c(KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
(* the client creates a new name*)

COMMIT2c(KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

secret(msg).[msg=finish] **OPENc**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)

CREATEDc(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

$\overline{KOpt}$ CHello.KOpt(msg). **CREATEDc**(KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
$\overline{KOpt}$CHello2.**CREATINGc**(KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
$\overline{KOpt}$ CipherSpec. (ν secret) $\overline{KOpt}$ secret.
OPENINGc(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
(* the client creates a new name*)

OPENINGc(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

secret(msg).[msg=finish]**OPENc**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)

OPENc(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

secret(msg).[msg=data]**OPENc**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
$\overline{secret}$ data.**OPENc**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
$\overline{secret}$ clientHello. **NULLc**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)

Server States

NULLs(CHello,HRequest,SHello,ch)=

$\overline{ch}$ HRequest.ch(msg).[msg=CHello]**NULLs**(CHello,HRequest,SHello,ch)
+
ch(msg).([msg=CHello]**CREATINGs**(CHello,HRequest,SHello,ch,KFull,KOpt)
+
[msg=alert]**NULLs**(CHello,HRequest,SHello,ch)
)

CREATINGs(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange)=

$\overline{ch}$ SHello. **CREATEDs**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange)
+
$\overline{ch}$ alert. **NULLs**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange)
+
$\overline{ch}$ warning. **CREATINGs**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange)
+
ch(msg).([msg=warning] **CREATING**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange)
+
[msg=alert]
NULLs(CHello,helloRequest,serverHello,ch,keyFull,keyOpt,clientHello2,clientKeyExchange)
)

CREATEDs(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)=

$\overline{ch}$ KFull.**EXCHANGEs**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)
+
$\overline{ch}$ KOpt.**COMMITs**(CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec)

EXCHANGEc(ch,keyFull,keyOpt,clientHello2,clientKeyExchange ,changeCipherSpec,finish,data) =

KFull(msg).([msg=CHello]$\overline{KFull}$ clientHello.
EXCHANGEs(KFull,CHello2,KExchange,CipherSpec,finish,data)
+
[msg=CHello2] **CREATINGs**(KFull,CHello2,KExchange,CipherSpec,finish,data)
+
[msg=KExchange] **OPENINGs**(KFull,CHello2,KExchange,CipherSpec,finish,data)
)

COMMITs(KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

KOpt(msg).([msg=clientHello]$\overline{keyOpt}$ msg.
COMMITs(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
[msg=CHello2] **CREATINGs**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
[msg=CipherSpec] KOpt(msg1). $\overline{msg1}$ finish.
OPENs(ch,keyFull,keyOpt,clientHello2,clientKeyExchange ,changeCipherSpec,finish,data,msg1)
)

OPENINGs(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data,keyFull2)=

KFull(msg).([msg=CipherSpec]KFull(msg).
OPENINGs(ch,msg,KOpt,CHello2,KExchange,CipherSpec,finish,data,KFull)
+
[msg=data] **OPENs**(ch,KFull2,KOpt,CHello2,KExchange,CipherSpec,finish,data,KFull)
)
+
$\overline{secret}$ finish. **OPENINGs**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data,KFull2)

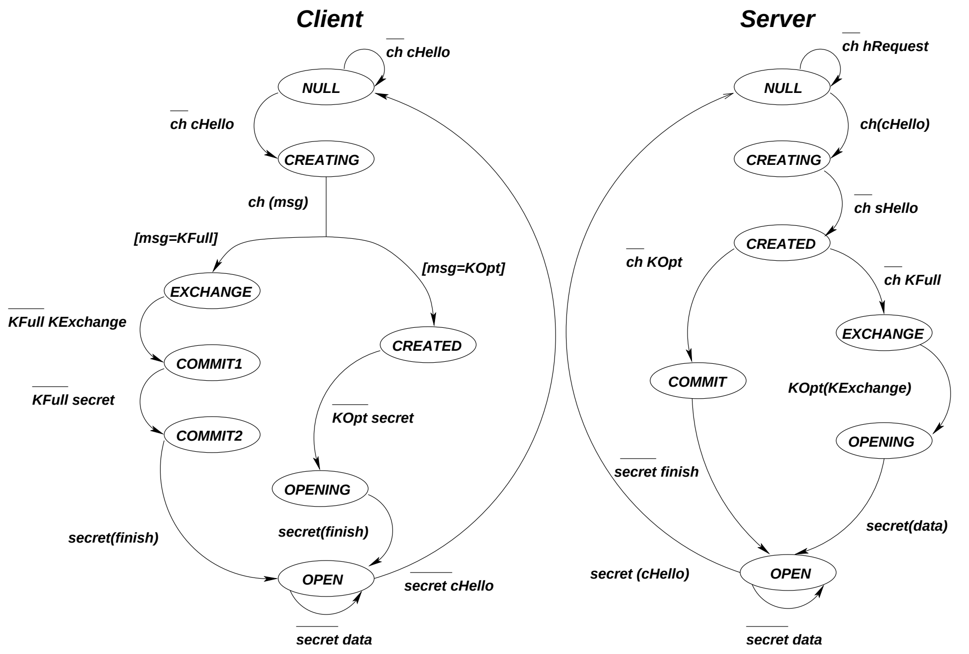


Figure 1: State scheme of the two peers

OPENs(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)=

secret(msg).([msg=data] **OPENs**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
+
[msg=CHello] **NULLs**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
)
+
$\overline{secret}$ data . **OPENs**(ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)

(ν ch,keyFull,keyOpt)
(
NULLc (CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
||
NULLs (CHello,HRequest,SHello,ch,KFull,KOpt,CHello2,KExchange,CipherSpec,finish,data)
)

4 Analysis

There are various network architectures which can support the WAP protocol. We chose the GPRS which is the evolution of the GSM bearer and which will be soon in use. The main difference between GSM and GPRS is that the former uses a resource reservation modality for connection, while the latter uses a context reservation one (see http://it.gsmbox.com/gprs/). The new technology was introduced by the European Telecommunication Standard Institute (ETSI, http://www.etsi.org/), and allows users to pay for the amount of data sent and not for the amount of time the communication

Reliability class	Lost SDU probability (a)	Duplicate SDU probability	Out of Sequence SDU probability	Corrupt SDU probability (b)	Example of application characteristics.
1	10^{-9}	10^{-9}	10^{-9}	10^{-9}	Error sensitive, no error correction capability, limited error tolerance capability.
2	10^{-4}	10^{-5}	10^{-5}	10^{-6}	Error sensitive, limited error correction capability, good error tolerance capability.
3	10^{-2}	10^{-5}	10^{-5}	10^{-2}	Nor error sensitive, error correction capability and/or very good error tolerance capability.

Table 2: Reliability classes

	Delay(maximum values)			
	SDU size: 128 octets		SDU size: 1024 octets	
Delay Class	Mean Transfer Delay (sec)	95 percentile Delay (sec)	Mean Transfer Delay (sec)	95 percentile Delay (sec)
1.(Predictive)	< 0.5	< 1.5	< 2	< 7
2.(Predictive)	< 5	< 25	< 15	< 75
3.(Predictive)	< 50	< 250	< 75	< 375
4.(Best Effort)	Unspecified			

Table 3: Delay classes

takes. We are interested in obtaining some architectural measures to associate a rate to each π-calculus action. In particular we would like to have information about the time that a communication takes. The tables 2 and 3 (see [1] for more details) contains reliability and delay parameters for GPRS connections, and represent a starting point for our analysis. The Tab. 2 reports the *Service Data Units error probability* (SDU) which is the ratio of total incorrect SDU to total successfully transferred SDU plus incorrect SDU in a specified sample. The Tab. 3 contains the maximum values for the mean delay and 95 -percentile delay to be incurred by the transfer of data through the GPRS networks. The delay parameter defines the end-to-end transfer delay incurred in the transmission of SDUs through the GPRS networks.

To have preliminary results we assume that all the messages have the same size of 128 bytes. With this assumption we use the delays in Tab. 3. We distinguish only the alert and warning messages from the others because of their lower frequency, and we estimate the waiting time for an alert or a warning message using the probabilities in Tab. 2.

We are interested here in finding how security affects performance. Hence, we study the throughput of the system with respect to the generation of the secret key (transitions generated by the prefix (ν secret)$\overline{KFull}$secret in the state COMMIT1c and by (ν secret)$\overline{KOpt}$secret in the state CREATEDc). We studied the case in which only the full handshake is available by assigning reward 1 to the corresponding secret key generation and the case in which both kind of handshakes are available by assigning reward 1 to both generation transitions. The results we got for reliability and delay class 2 are that in the former case the throughput is about 2% (0.0195577), while in the latter case it is about 4% (0.0391158). If the only handshake available is the optimized one, we got throughput 0.0195581. For the other classes we got similar results as the fraction of the involved values is more or less the same of class 2.

The lesson learned is that if a single handshake policy is fixed, one can invest in quite strong algorithms for key generation being the number of key generation during a normal session of the protocol quite small.

5 Conclusions and further work

The paper presented a formal specification of the secure transport layer of the WAP reference model. We applied the enhanced operational semantics framework to get performance prediction measures on the impact of the secret key generation in the WAP hierarchy of protocols. The case study has been performed assuming a GPRS bearer. We plan to investigate the different impact of the optional security layer according to other bearers, among which UMTS.

This study is a preliminary assessment of the feasibility of the EOS framework to carry out integrated qualitative and quantitative analysis. This feature is a key ingredient in the design and development of wireless telecommunication applications, where security is a trade-off between performance and probability of being broken.

References

[1] Digital cellular telecommunications system (phase 2+); general packet radio service (gprs); service description; stage1. ETSI EN 301 113 v6.3.1, ETSI, http://www.etsi.org/, 11 2000.

[2] WAP WTLS. Technical Report WAP-199-WTLS, WAP Forum Ltd., http://www.wapforum.org, 2000.

[3] M. Bernardo, L. Donatiello, and R. Gorrieri. A formal approach to the integration of performance aspects in the modelling and analysis of concurrent systems. *Information and Computation*, 144:83–154, 1998.

[4] G. Boudol and I. Castellani. A non-interleaving semantics for CCS based on proved transitions. *Fundamenta Informaticae*, XI(4):433–452, 1988.

[5] M. Bravetti, M. Bernardo, and R. Gorrieri. Towards performance evaluation with general distributions in process algebra. In *Proceedings of CONCUR'98, LNCS 1466*, pages 405–422, 1998.

[6] M. Bravetti and R. Gorrieri. Interactive generalized semi-Markov processes. In *Proceedings of PAPM'99*, 1999.

[7] E. Brinksma, J.-P. Katoen, R. Langerak, and D. Latella. A stochastic causality based process algebra. *The Computer Journal*, 38(6):552–565, 1995.

[8] L. Brodo, P. Degano, and C. Priami. A tool for quantitative analysis of π-calculus processes. In *Proceedings of PAPM'00*. Carleton Scientific, 2000.

[9] P. Buchholz. On a Markovian process algebra. Technical Report 500, Informatik IV, University of Dortmund, 1994.

[10] P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, University of Twente, 1999.

[11] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. An algebraic approach to the specification of stochastic systems. In *Proceedings of PROCOMET'98, IFIP Series*, pages 126–147. Chapman-Hall, 1998.

[12] P. Degano, R. De Nicola, and U. Montanari. Partial ordering derivations for CCS. In *Proceedings of FCT'85, LNCS 199*, pages 520–533. Springer-Verlag, 1985.

[13] P. Degano and C. Priami. Enhanced operational semantics: a tool for describing and analysing concurrent systems. *ACM Computing Surveys*. To appear.

[14] P. Degano and C. Priami. Causality for mobile processes. In *Proceedings of ICALP'95, LNCS 944*, pages 660–671. Springer-Verlag, 1995. Extended version to appear in Theoretical Computer Science.

[15] P. Degano and C. Priami. Non interleaving semantics for mobile processes. *Theoretical Computer Science*, 216:237–270, 1999.

[16] N. Götz, U. Herzog, and M. Rettelbach. TIPP- a language for timed processes and performance evaluation. Technical Report 4/92, IMMD VII, University of Erlangen-Nurnberg, 1992.

[17] P.G. Harrison and Strulo B. Stochastic process algebra for discrete event simulation. In *Quantitative Methods in Parallel Systems*, pages 18–37, 1995.

[18] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[19] J.-P. Katoen. *Quantitative and Qualitative Extensions of Event Structures*. PhD thesis, Center for Telematics and Information Technology, University of Twente, 1996. Available as CTIT No. 96-09.

[20] E. Madelaine and R. de Simone. Fc2: reference manual 1.1. Technical report, INRIA Technical Report, 1993.

[21] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.

[22] C. Nottegar, C. Priami, and P. Degano. Performance evaluation of mobile processes via abstract machines. *IEEE TSE*. To appear.

[23] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[24] C. Priami. Stochastic π-calculus. *The Computer Journal*, 38(6):578–589, 1995.

[25] C. Priami. Integrating behavioural and performance analysis with topology information. In *Proceedings of 29th Hawaian International Conference on System Sciences*, volume 1, pages 508–516, Maui, Hawaii, 1996. IEEE.

[26] C. Priami. Language-based performance prediction for distributed and mobile systems. *Information and Computation*, 2001. To appear.

[27] V. Roy and R. de Simone. An AUTOGRAPH primer. Technical Report 112, INRIA Technical Report, 1989.

Using Rewriting Logic to implement FULL

[Extended Abstract]

J. Bryans[1], A. Verdejo[2] and C. Shankland[1]

[1]Department of Computing Science and Mathematics,
University of Stirling, Stirling, FK9 4LA, UK
email: {jwb, ces}@cs.stir.ac.uk

[2]Universidad Complutense de Madrid, 28040 Madrid, Spain
email: alberto@sip.ucm.es

1 Introduction

A common problem of automated tools for Formal Methods is their difficulty in dealing with infinite systems. Such systems may arise through the use of infinite data types. For example, in the LOTOS [7] formal description technique the simple process $g?x : Nat; exit$ results in an infinite choice, one for each member of Nat. Rewriting logic [8] is a possible approach to dealing with such systems.

Rewriting logic is a logic of *becoming* or *change* that can naturally deal with state and with highly nondeterministic concurrent computations. It can be used as a semantic framework for a wide range of languages and models of concurrency. Maude [4] is an executable specification language based on rewriting logic and supporting both equational and rewriting logic computation. Maude can be used as a *metalanguage* [5] in which executable environments for different logics, languages, and models of computation are created. Maude has powerful metaprogramming capabilities, with which the user can define strategies to guide the deduction process.

We present here work on the use of rewriting logic and Maude to implement a model checker for the modal logic FULL [1, 2]. FULL is used to describe properties over data and processes, and is verified with respect to *symbolic transition systems* (STS).

2 Symbolic Transition Systems and FULL

Symbolic Transition Systems are essentially transition systems whose transitions can have free variables in the data label and are additionally labelled with a *transition condition* representing the conditions under which that transition is available. This approach was first introduced by Hennessy and Lin [6] who gave a symbolic semantics for value passing CCS.

Basically, an STS is a directed graph whose nodes are tagged with sets of free variables, and whose branches are labelled with a boolean condition and an event. For example, $g?x : Nat[x > 5]; P \xrightarrow{x > 5 \quad gx} P$ is the transition from the process "$g?x : Nat[x > 5]$ then P" via action gx, where g is a gate name and x is a symbolic value, with condition $x > 5$ to the process "P". This transition may not be taken if the condition is not satisfied. When reasoning about processes we simply collect such conditions and pass them to a separate verification system for the data type.

Although Symbolic Transition Systems can be used independently, our work has been focused on using them to give a new finitely branching semantics [3] for LOTOS. LOTOS is a popular process description language that has been used in a variety of applications. The language includes a rich set of operators for describing both process control *and* data, which may in turn affect control. However, much of the foundational work, and subsequently the verification tools, has ignored all, or parts, of the data aspect of the language because of the problem of infinite branching.

A set of rules presented in [3] define how a symbolic transition system may be constructed from a LOTOS process expression. The resulting transition system is typically a cyclic graph (if recursive processes are involved) and is always of finite width (since only a finite number of branches may be described in a LOTOS process).

In order to describe properties of both the actions and the data in symbolic transition systems we developed a modal logic called Full LOTOS Logic (FULL) Earlier work [1] describes the logic in more detail; here we give an overview.

FULL is based on HML as presented in [9]. The usual modal operators $\langle\ \rangle$ and $[\ ]$ have been supplemented by quantifiers over data. For example, $\langle \exists x\ g\rangle\Phi$ expresses that there is some action gv such that Φ with v substituted for x holds subsequently. The value v may be a specific value in the transition system, or another symbolic one. The other operators are $[\exists x\ g]$, $\langle\forall x\ g\rangle$ and $[\forall x\ g]$. These express different path computations through the STS.

3 Implementation of STSs and FULL using Maude

Following the ideas presented in [10] we have been working on representing the LOTOS symbolic semantics and the FULL logic in Maude.

Symbolic transitions are represented as terms of sort `Transition`, and the semantic rules are translated into rewrite rules where the conclusion is rewritten to the premisses. In this way, we start with a transition to be proved valid and work backwards using the rewriting process, maintaining a set of transitions that have to be fulfilled in order to prove the correctness of the first transition. This transition can be rewritten to the empty set if and only if it is a valid transition in the LOTOS symbolic semantics.

Examples of semantic rules and its translation as rewrite rules are

$$a; P \xrightarrow{\mathrm{tt} \quad a} P$$

```
rl [sym] :
     A ; P -- ?b -- ?a --> ?P
  => --------------------------------
        [ ?b := true ]
        [ ?a := A ]
        [ ?P := P ] .
```

which says that the transition `A ; P -- ?b -- ?a --> ?P` is a valid transition that produces three bindings: metavariable `?b` is bound to the value `true`, `?a` to action `A`, and `?P` to process `P`; and

$$\frac{P_1 \xrightarrow{b \quad \alpha} P_1'}{P_1\,[> P_2 \xrightarrow{b \quad \alpha} P_1'\,[> P_2} \quad \text{if } \mathbf{name}(\alpha) \neq \delta$$

```
rl [sym] :
     P1 [> P2 -- ?b -- ?a --> ?P
  => --------------------------------
        P1 -- ?b -- ?a --> ?(NEW1)P
        [ name(?a) =/= delta ]
        [ ?P := ?(NEW1)P [> P2 ] .
```

There are two problems in this approach: the existence of new variables in the righthand side of the rewrite rules and the nondeterministic application of the semantic rules, arising from, for example, rules for choice. To solve the first problem we use *explicit metavariables*, that will be bound to concrete values in the process of rewriting and these bindings will be propagated to the rest of transitions. For example, in the rule above `?(NEW1)P` is an explicit metavariable.

To solve the second problem of nondeterministic application of rewrite rules, we need a strategy to control the rewriting of a term and the search in the tree of all possible rewrites of a term. This will also help us deal with metavariables.

We extend `META-LEVEL` with operations that generate all the one-step rewritings of a term, and operations that search in a depth-first way the tree of possible rewritings looking for the empty set of transitions. `META-LEVEL` is a predefined Maude module which implements the rewriting logic reflective capabilities and allows us to deal with modules as normal terms and control the rewriting process. In this controlled process of rewriting, new metavariables are created and substituted for the new variables in the righthand side of the rewrite rules.

While these operations work, bindings are produced. If we keep these bindings, we can obtain new information besides the fact that the transition is possible. For example, if we start with the transition where the final process is a metavariable, it will be bound to one of the possible successors of the first process. This is useful to represent the FULL logic.

Some of the operators of the modal logic FULL are equivalent to operators within the Hennessy Milner logic HML, and can be implemented in very much the same way. We begin by defing a sort `FullFormula` to represent the logic and its operators. (At the time of writing not all of the operators of FULL have been implemented.)

```
sort FullFormula .
ops tt ff : -> FullFormula .
op _/\_ : FullFormula FullFormula -> FullFormula .
op `[_`]_ : TermSet FullFormula -> FullFormula .
op forall : TermSet FullFormula -> JudgementSeq .
op _|=_ : Term FullFormula -> Judgement .
```

Then, we define rewrite rules that rewrite a judgement $P \models \Phi$ into the set of judgements that have to be fulfilled.

```
rl [and] : P |= Phi /\ Psi => (P |= Phi) (P |= Psi) .

rl [box] : P |= [ K ] Phi => forall(succ(P, K), Phi) .

eq forall({}, Phi) = emptyJS .
eq forall(P U PS, Phi) = (P |= Phi) forall(PS, Phi) .
```

So, just as the rules for the symbolic semantics rewrite a transition to be proved valid into a set of transitions to be fulfilled, here we rewrite a FULL formula to be proved valid into a set of judgements to be fulfilled. The goal is to produce the empty judgement set, indicating that the original FULL formula is true. For example, the base cases include the rule where the judgement `P |= true` is rewritten to `emptyJS`.

An advantage of using Maude is that the reasoning can be carried out completely automatically by incorporating the Full LOTOS data types in the same tool. This will be done by translating the ACT ONE data type specifications into Maude equational specifications that will be included automatically in the Maude module containing the semantic rules. In this way, rewriting will be perfomed modulo the ACT ONE equations defining the particular data types of the Full LOTOS specification we are working with.

Work is ongoing to implement the rest of the FULL operators, i.e. the quantified modal operators. Particular problems to be addressed are the implementation of substitution (crucial to the quantified modal operators), and the representation of infinite depth transition systems.

References

[1] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS based on Symbolic Transition Systems. *The Computer Journal*, 2001. In press.

[2] M. Calder, S. Maharaj, and C. Shankland. An Adequate Logic for Full LOTOS. To appear in Proceedings of Formal Methods Europe'01, LNCS series, Springer-Verlag, 2001.

[3] M. Calder and C. Shankland. A Symbolic Semantics and Bisimulation for Full LOTOS. Technical Report CSM-159, University of Stirling, 2000.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, Jan. 1999, revised Aug. 1999. `http://maude.csl.sri.com`.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José Quesada. Maude as a metalanguage. In Claude Kirchner and Hélène Kirchner, editors, *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications, Pont-à-Mousson, Nancy, France*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[6] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[7] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[8] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[9] C. Stirling. Temporal Logics for CCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 660–672. Springer-Verlag, 1989. REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1988.

[10] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude. In Tommaso Bolognesi and Diego Latella, editors, *Formal Methods For Distributed System Development. FORTE/PSTV 2000 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX) October 10–13, 2000, Pisa, Italy*, pages 351–366. Kluwer Academic Publishers, 2000.

Dependable Mathematics for Critical Mathematical Modelling

Hanne Gottliebsen and Ursula Martin

University of St Andrews, Scotland
{hago,um}@dcs.st-and.ac.uk

Abstract Scientific computation based on continuous computational mathematics is widely used for applications as diverse as high energy physics, weather forecasting, and financial modelling, and is important in systems design for engineering applications like air-traffic control algorithms or synthesising code for control systems. Complex models and simulations are among the most demanding of computer power. Leading commercial software systems like Maple and NAG have around 2 mill users, and scientists and engineers have in-house code for many specialised applications.

Computational mathematics systems may be numeric or symbolic: both can be subject to error and problems of scale, and lack a framework for reasoning about the models they implement. We report on recently completed work which aimed to provide users of computational mathematics systems with computational logic support to increase their scope and dependability, while shielding them from its technicalities. This is realised in our Maple-PVS system, which augments Maple with schemata for PVS calls, which supports them with automated search and specialist libraries.

Our techniques address dynamical systems and differential equations, and include pre-condition checking, reasoning about side conditions, result verification and verifiable look-up tables for recording a variety of scientific formulae and data. To support our work we implemented in PVS specialist libraries for continuous mathematics, and strategies for automated checking of convergence and continuity.

References

1. Martin Dunstan, Tom Kelsey, Steve Linton and Ursula Martin, Lightweight formal methods for computer algebra systems, ISSAC'98, ACM Press, 1998
2. Martin Dunstan, Tom Kelsey, Steve Linton and Ursula Martin, Formal Methods for Extensions to CAS, FM'99, LNCS 1709, Springer 1999
3. Andrew Adams, Hanne Gottliebsen, Tom Kelsey, Steve Linton and Ursula Martin, VSDITLU: a verified symbolic definite integral table look-up, CADE 16, LNAI 1632, Springer 1999
4. Hanne Gottliebsen, Transcendental functions and continuity checking in PVS, T-PHOLS 2000, LNCS 1869, Springer 2000
5. Ursula Martin, Towards formal methods for mathematical modeling, Proceedings 5th NASA Langley Workshop on Formal Methods 2000, NASA Press 2000
6. The Maple-PVS system, www-theory.dcs.st-and.ac.uk/info/example/

Automatic Guidance for the Formal Verification of High Integrity Ada

Andrew Ireland
Department of Computing & Electrical Engineering
Heriot-Watt University
Edinburgh, Scotland
E-mail: A.Ireland@hw.ac.uk

Abstract

While the value of formal methods within the development of critical systems is recognized, scalability and usability remain key challenges to their wider adoption. Automation and high-level interaction will play an important role in meeting these challenges. Our aim is to investigate[1] techniques for automating the formal verification of software for critical systems. In particular we will focus upon SPARK [1], a subset of the Ada programming language developed by Praxis Critical Systems[2]. SPARK has a proven track record in the production of software for critical systems within the finance, aerospace, rail and telecommunications markets. The value of formal verification is recognized by Praxis as being crucial to the production of critical systems. This is reflected within the SPARK toolkit that supports formal verification through the SPADE Proof Checker. The SPADE Proof Checker, however, is interactive and requires highly skilled users with development times typically measured in person-months. The need for greater efficiency within the formal verification process is recognized by Praxis. Proof planning, as implemented within CLAM [3], has been investigated extensively within the context of proof by mathematical induction [2]. What distinguishes proof planning from other approaches to theorem proving is the flexibility it offers by separating the issues of search (method level) and soundness (tactic level). This means that the planning of a proof need not follow a strict backward or forward style of construction. This flexibility gave rise to *proof critics* [4, 6], an extension that supports the automatic analysis and patching of failed proof planning attempts. The most striking proof planning successes have been achieved where proofs require the discovery of auxiliary inductive lemmas or generalizations [5, 6, 7]. Such proof discovery capabilities outperform the conventional inductive theorem provers. Loop invariants are by their nature inductive so the proof plans for induction easily transferred to the problem of reasoning about imperative programs [9, 11, 12, 10]. In terms of high-level interaction, proof failures are also typically where experienced users make significant interventions. Proof critics, therefore, provide a very natural mechanism for managing user interaction as well as automatic proof patching. This observation gave rise to *interactive proof critics* [8], a CLAM interface that gives the user the opportunity of assisting with the proof patching process. Consequently, interactive proof critics extend the scope of the proof patching capabilities of CLAM by facilitating a collaborative style of interaction.

The time is now ripe to test the proof discovery and high-level interaction capabilities of proof planning on "industrial strength" problems. SPARK and the associated SPADE Proof Checker provide an the ideal vehicle given its commercial success within the production of

[1]The research proposal outlined here has been funded under the EPSRC Critical Systems Programme – EPSRC grant GR/R24081.

[2]http://www.praxis-cs.co.uk

critical systems. We believe that proof planning techniques can make a significant impact on the SPARK approach to producing high integrity software. We also believe that working on "industrial strength" problems will be beneficial to our basic research agenda.

References

[1] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison-Wesley, 1997.

[2] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[3] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[4] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[5] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[6] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[7] A. Ireland and A. Bundy. Automatic Verification of Functions with Accumulating Parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[8] A. Ireland, M. Jackson, and G. Reid. Interactive Proof Critics. *Formal Aspects of Computing: The International Journal of Formal Methods*, 11(3):302–325, 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/15.

[9] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[10] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[11] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, number 1559 in LNCS, pages 271–288. Springer-Verlag, 1998. An earlier version is available from the Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/2.

[12] J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In *The* 14^{th} *IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE Computer Society, 1999.

Formal Development and Verification of Security-Critical Systems with UML (Position Paper)

Jan Jürjens*

Computing Laboratory, University of Oxford, GB

Abstract. We sketch ongoing work on using a formal fragment of the object-oriented modelling language UML (Unified Modeling Language) in development and verification of secure systems.

We sketch some current challenges for the formal (in particular automated) development and verification of (in particular) security-critical systems and note how some of our work tries to address these.

When trying to use formal (and in particular automated) techniques to develop or verify real-life security-critical systems whose specifications can easily consist of several hundred pages or more (e.g. the Common Electronic Purse Specifications (CEPS) considered in [Jür01c]) one faces a number of problems in scaling up existing approaches:

(1) Usually a formal specification of the system is not available. Constructing it requires expert knowledge and can be very time-consuming. Currently a large part of effort both in verifying and in implementing specifications is wasted since these are often formulated imprecisely and unintelligibly [Pau98].

(2) It is usually only feasible to construct the specification of a small security-critical part of the system (e.g. security protocols [Low96, LR97, RSG$^+$01, SCW00], where formal analysis has in fact been quite successful). The boundaries of these components with the rest of the system need to be carefully examined, e.g. wrt implicit assumptions on the system context or the underlying physical layer [Gol00, Aba00]. In practice, most attacks do not address vulnerabilities in dedicated mechanisms (such as security protocols), but in the way these are employed in the system context [And01]. An example is given in [JW01a], where a vulnerability arises from the intended use of the CEPS payment scheme over the Internet.

* `http://www.jurjens.de/jan` – `jan@comlab.ox.ac.uk` - Supported by the Studienstiftung des deutschen Volkes and the Computing Laboratory.

(3) Stepwise development and modular reasoning are hindered by the fact that many security properties are not preserved under refinement and not compositional.
(4) Even when specifying small components, one often has to give a simplifying account. In particular, in symbolic modelling cryptographic operations one abstracts from the possibility that an adversary may simply guess secrect values.
(5) Often, vulnerabilities arise from flaws in the implementation rather than the design.

We try to address some of these issues in previous, current and future work as follows:

(1) As a specification language we propose to use a formal fragment of UML [OMG99], extended with security-relevant primitives using the UML extension mechanisms [Jür01e, Jür01c, Jür01f, Jür01b]. As UML is the de-facto industry-standard in object-oriented modelling, in some cases a specification of the system in it may already be available, or at least there may be knowledge available on the side of the developers how to construct the specification (additionally, there is work on deriving Java code from UML specifications [EHSW99] and vc. vs. [KG01], narrowing the gap between specification- and software-based verification). For mechanical verification one may take advantage of work giving UML diagrams a formal semantics[1] in CSP, allowing to use FDR2 [BD00, Cri01], and of other work towards tool-support for UML [Ste00].
(2) Through its different kinds of diagrams, UML allows to take different views on the system (e.g. on the physical layer). This allows expressing properties of and reasoning about boundaries of critical components.
(3) For stepwise development and modular reasoning, one may make use of work in [Jür01d, Jür01a] giving a notion of secrecy preserved under refinement and giving compositionality results in the setting of a simple specification language (the current work aims to transport these results into the setting of UML).
(4) The question to what extent the formal methods approach to modelling cryptographic primitives is sound is addressed in [AJ01], where a translation from a symbolic semantics of a specification language involving cryptographic operations to a complexity-theoretical one is given. Again these results apply (after adjusting the differences in the models) in particular to a formal core of UML.

[1] For some background on giving UML a formal semantics cf. e.g. [EFLR99].

(5) One may address vulnerabilities arising from flaws in the implementation by deriving test-sequences for implementations from formal specifications (e.g. following ideas from [JW01b], where this is done using a formal notation close to UML).

Beyond verification, one may also employ UML in requirements capture for security protocols [Low00] or to encapsulate rules of prudent engineering for secure systems [Jür01b].

In this talk we report on work towards verifying security properties in a formal core of UML in a compositional way.

References

[Aba00] M. Abadi. Security protocols and their properties. In F. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*. IOS Press, 2000.

[AJ01] M. Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation, 2001. Submitted.

[And01] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[BD00] C. Bolton and J. Davies. Activity graphs and processes. In *Integrated Formal Methods*, LNCS. Springer, 2000.

[Cri01] C. Crichton. UML statecharts and CSP, 2001. In preparation.

[EFLR99] A. Evans, R. France, K. Lano, and B. Rumpe. The UML as a formal modeling notation. In J. Bezivin and P.-A. Muller, editors, *The Unified Modeling Language - Workshop UML'98: Beyond the Notation*, LNCS. Springer, 1999.

[EHSW99] G. Engels, R. Hücking, S. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In *[FR99]*, pages 473–488, 1999.

[FR99] R. France and B. Rumpe, editors. *Second International Conference on the Unified Modeling Language - UML'99*, volume 1723 of *LNCS*. Springer, 1999.

[Gol00] Dieter Gollmann. On the verification of cryptographic protocols - a tale of two committees. In *Workshop on Security Architectures and Information Flow*, volume 32 of *Electronical Notes in Theoretical Computer Science*, 2000.

[Huß01] H. Hußmann, editor. *Fundamental Approaches to Software Engineering (FASE/ETAPS, International Conference)*, LNCS. Springer, 2001.

[Jür01a] Jan Jürjens. Composability of secrecy. In *International Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security (MMM-ACNS 2001)*, LNCS, St. Petersburg, 21-23 May 2001. Springer.

[Jür01b] Jan Jürjens. Encapsulating rules of prudent security engineering. In *International Workshop on Security Protocols*, LNCS. Springer, 2001.

[Jür01c] Jan Jürjens. Modelling audit security for smart-card payment schemes with UMLsec. In P. Paradinas, editor, *IFIP/SEC 2001 – 16th International Conference on Information Security*, Paris, 11–13 June 2001. Kluwer.

[Jür01d] Jan Jürjens. Secrecy-preserving refinement. In *Formal Methods Europe (International Symposium)*, LNCS. Springer, 2001.

[Jür01e] Jan Jürjens. Towards development of secure systems using UMLsec. In *[Huß01]*, 2001. Also OUCL TR-9-00 (Nov. 2000), http://web.comlab.ox.ac.uk/oucl/publications/tr/tr-9-00.html.

[Jür01f] Jan Jürjens. Transformations for introducing patterns – a secure systems case study. In *WTUML: Workshop on Transformations in UML (ETAPS 2001 Satellite Event)*, Genova, 7 April 2001.

[JW01a] Jan Jürjens and Guido Wimmel. Security modelling for electronic commerce: The Common Electronic Purse Specifications. Submitted, 2001.

[JW01b] Jan Jürjens and Guido Wimmel. Specification-based testing of firewalls. Submitted, 2001.

[KG01] R. Kollmann and M. Gogolla. Capturing dynamic program behaviour with UML collaboration diagrams. In *5th European Conference on Software Maintenance and Reengineering*. IEEE, 2001.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. *Software Concepts and Tools*, 17:93–102, 1996.

[Low00] G. Lowe. External specifications for security protocols, 2000. Draft.

[LR97] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10), 1997.

[OMG99] UML Revision Task Force, OMG. UML Specification 1.3. Available at http://www.omg.org/uml, 1999.

[Pau98] L. Paulson. Inductive analysis of the Internet protocol TLS (transcript of discussion). In B. Christianson, B. Crispo, W.S. Harbison, and M. Roe, editors, *Security Protocols – 6th International Workshop*, number 1550 in LNCS, page 13 ff., Cambridge, UK, April 1998.

[RSG+01] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *Modelling and Analysis of Security Protocols*. Addison Wesley, 2001. (to be published).

[SCW00] S. Stepney, D. Cooper, and J. Woodcock. *An Electronic Purse: Specification, Refinement, and Proof*. Oxford University Computing Laboratory, 2000. Technical Monograph PRG-126.

[Ste00] P. Stevens. Advanced tools for UML: now and in the future, 2000. Tutorial given at UML2000.

Symbolic CTMC Model Checking*

Joost-Pieter Katoen[a], Marta Kwiatkowska[b],
Gethin Norman[b] and David Parker[b]

[a] *Formal Methods and Tools Group, Faculty of Computer Science*
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands

[b] *School of Computer Science, University of Birmingham*
Edgbaston, Birmingham B15 2TT, United Kingdom

Abstract. In model-based performance and dependability evaluation, techniques such as stochastic Petri nets, stochastic process algebras, stochastic activity networks, or queueing networks are used to specify the system behaviour at a high level of abstraction. Most of these techniques assume a continuous-time Markov chain (CTMC) as underlying stochastic process. Whereas traditionally the analysis of CTMCs has been focussed on their transient-state and steady-state (i.e, long run) characteristics, the specification and analysis of path-based measures is receiving growing interest.

The temporal logic CSL (Continuous Stochastic Logic) developed originally by Aziz *et al.* [ASSB96,ASSB00] and extended by Baier *et al.* [BKH99], provides a powerful means to specify path-based as well as traditional state-based measures in a compact, flexible and unambiguous way. CSL is based on the well-known branching temporal logic CTL (computation tree logic) and PCTL (probabilistic CTL [HJ94]); a steady-state operator, a time-bounded until, and a probabilistic (path) operator constitute its main ingredients. It allows one, for example, to state that the probability of reaching a certain set of goal-states within a number of time-units, provided that all paths to these states obey certain properties, is within some probability bound.

We report on the implementation of *symbolic* model checking for CTMCs and CSL, using multi-terminal binary decision diagrams (MTBDDs) [CFM+93]. All CSL operators can be treated using standard operations on MTBDDs, thus allowing a rather straightforward implementation of symbolic CSL model checking on existing MTBDD-based platforms such as the model checker PRISM[1].

References

[ASSB96] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In *Proc. Conference on Computer-Aided Verification*, July 1996.

[ASSB00] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous time Markov chains. *ACM Trans. on Computational Logic*, 1(1), 2000.

[BKH99] C. Baier, J.-P. Katoen, and H. Hermanns. Approximative symbolic model checking of continuous-time Markov chains. In *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, Eindhoven, 1999.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi-Terminal Binary Decision Diagrams: An Effiicient Data Structure for Matrix Representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, pages 6a:1–15, 1993.

[HJ94] H. Hansson and B. Jonsson. A logic for reasoning about time and probability. *Formal Aspects of Computing*, 6:512–535, 1994.

---

* Supported in part by EPSRC grants GR/M04617 and GR/N31573.

[1] `http://www.cs.bham.ac.uk/~dxp/prism/`

Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM*

Marta Kwiatkowska[1], Gethin Norman[1], and Roberto Segala[2]

[1] School of Computer Science, University of Birmingham, Birmingham B15 2TT, United Kingdom
{M.Z.Kwiatkowska,G.Norman}@cs.bham.ac.uk
[2] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy
segala@cs.unibo.it

Abstract. Randomization in the form of coin flipping is a tool increasingly often used as a symmetry breaker in distributed algorithms, for example, to solve leader election, mutual exclusion or consensus problems. Such algorithms are inevitably very difficult to analyse, and therefore appropriate methods of automating their correctness proofs are called for. Furthermore, the use of random choices means that certain properties, such as eventual termination, become probabilistic, and thus cannot be handled by conventional model checking tools.

We consider the *randomized consensus* protocol due to Aspnes and Herlihy [1] for achieving agreement among N asynchronous processes that communicate via read/write shared registers, which guarantees termination in the presence of stopping failures within expected polynomial time. The processes proceed through possibly *unboundedly many* rounds; at each round, they read the status of all other processes and attempt to agree. The agreement attempt involves a *distributed random walk* (a Markov decision process, i.e. a combination of nondeterministic and probabilistic choices): when the processes disagree, they use a shared coin-flipping protocol to decide their next preferred value. Achieving polynomial expected time depends in an essential way on ensuring that the *probability* that all non-failed processes draw the same value is above an appropriate bound.

We adopt the approach of Pogosyants, Segala and Lynch [4] to analyse the Aspnes and Herlihy algorithm: we separate the algorithm into two communicating components, one non-probabilistic (asynchronous parallel composition of processes which periodically request the outcome of a coin protocol) and the other probabilistic (a coin-flipping protocol shared by the processes). For the non-probabilistic part we use the proof assistant Cadence SMV[1], which enables us to verify the non-probabilistic requirements *for all N* and *for all rounds* by applying the *compositional methods* introduced in [3]. The shared coin-flipping protocol is modelled and verified using the probabilistic model checker PRISM [2]. For a finite number of processes (up to 10) we are able to mechanically calculate the exact (minimum) probability of the processes drawing the same value, as opposed to a lower bound established analytically in [1] using random walk theory. The correctness of the full protocol (for the finite configurations mentioned above) follows from the separately proved properties. This is the first time a complex randomized distributed algorithm has been mechanically verified.

References

1. J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–460, 1990.
2. L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic systems using MTBDDs and the Kronecker representation. In *Proc. TACAS'2000*, volume 1785 of *LNCS*, pages 395–410. Springer, 2000.
3. K. McMillan, S. Qadeer, and J. Saxe. Induction and compositional model checking. In E. A. Emerson and A. P. Sistla, editors, *Proc. CAV 2000*, volume 1855 of *LNCS*, pages 312–327. Springer, 2000.
4. A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing*, 13(3):155–186, 2000. An extended abstract appears in Proc. WDAG, pages 111–125, LNCS 1320, 1997.

* Supported in part by EPSRC grants GR/M04617 and GR/M13046.
[1] http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/

Probabilistic Model Checking of Deadline Properties in the IEEE1394 FireWire Root Contention Protocol *

Marta Kwiatkowska, Gethin Norman, and Jeremy Sproston

School of Computer Science, University of Birmingham,
Birmingham B15 2TT, United Kingdom
M.Z.Kwiatkowska,G.Norman,J.Sproston@cs.bham.ac.uk

1 Introduction

The increasing dependence of businesses on distributed architectures and computer networking places heavy demands on the speed and reliability of data exchange, leading to the emergence of sophisticated protocols which involve both real-time and randomization, for example FireWire IEEE1394. Automatic verification techniques such as model checking have been adapted to this class of *probabilistic*, *timed* systems [1, 9, 3, 14]. This abstract considers an application of such techniques to the IEEE1394 (FireWire) root contention protocol, in which the interplay between timed and probabilistic aspects is used to break the symmetry which may arise during the leader election process. Here, the properties of interest concern the election of a leader *within a certain deadline, with a certain probability or greater.*

Our specification formalism is that of *probabilistic timed automata* [14], a variant of timed automata [2] extended with discrete probability distributions. We claim that this model is sufficiently expressive to describe formally both the timed and probabilistic aspects of the protocol. Furthermore, the formalism is amenable to model checking against probabilistic timed temporal logic properties, for which satisfaction is required with a certain probability bound (for example, the deadline property mentioned above). In contrast to the models of [1, 3], probabilistic timed automata exhibit nondeterminism, which can be used to represent unknown timing delays of the protocol, and, unlike the model of [9], can be verified against deadline properties.

A number of examples of formal verification applied to the IEEE1394 protocol exist. The approach of [19] is to consider a probabilistic timed model of the protocol, which is then verified *manually* in a process of stepwise abstraction. Such a refinement process is repeated in [17]; here, the model takes the form of a non-probabilistic timed automaton, with the real-time model checker Uppaal [15] being used to establish implementation relations. Real-time model checking is also performed by [5], in which probabilistic behaviour is modelled by fairness requirements. Our aims are somewhat different to these case studies: in all three, deadline properties are not considered, with their emphasis concerning functional correctness rather than performance issues. In this sense, our approach has more in common with the discrete-event simulation performed on a stochastic process algebra model of the protocol in [6].

Noting that the algorithm of [14] for reachability properties proceeds by constructing a finite-state quotient of the infinite-state probabilistic timed automaton, which is subsequently subject to probabilistic analysis, we use a two-phase verification technique. The first phase involves the use of the tool HyTech [12] to analyse the timed aspects of the protocol, whereas the second uses the probabilistic timed model checking tool Prism [10]. Observe that the use of only *either* HyTech or Prism is insufficient for the verification of probabilistic deadline properties (we either lose all of the timing or probabilistic information in such a case). Although HyTech was ostensibly introduced for the analysis of hybrid systems, it is also useful in our context, primarily due to its powerful analysis language. The tool Prism supports model checking of Markov decision process models (which exhibit probabilistic and nondeterministic behaviour) against properties expressed in the temporal logic Probabilistic CTL (PCTL) [11].

* Supported in part by the EPSRC grants GR/M04617 and GR/N22960.

2 Probabilistic Timed Automata

In this section, we introduce probabilistic timed automata [14] as an extension of timed automata [2] (henceforth referred to as *classical timed automata*) with *probabilistic branching* over edges. Recall that a classical timed automaton is obtained by augmenting a finite labelled transition system with a finite set of real-valued *clocks*, and with the facility to test and reset (to 0) the values of these clocks. Then a probabilistic timed automaton is obtained by equipping a classical timed automaton with discrete probability distributions, the support (the nonzero elements) of which are edges. To arrive at a meaningful model in which all of the edges in the support of a distribution are enabled at the same time, we associate guards (conditions on clocks which enable transitions) with distributions rather than with individual edges. However, the notion of clock resetting is still associated with edges. Note that, conversely, a probabilistic timed automaton may be thought of as a Markov decision process with the facility to test and reset clocks in the style of classical timed automata.

Model checking strategies for classical and probabilistic timed automata take into account the following: clocks are real-valued variables, and therefore the set of possible values that they take in a given model is infinite. Since a state of a classical or probabilistic timed automaton is defined as a pair consisting of a location and a valuation of all of the clocks, the underlying semantic model of such a system is infinite-state. However, under certain conditions (e.g. clocks are only tested against rational constants [2]), a *finite* transition system which can be model checked can be derived. One derivation concerns the verification of timed reachability properties by *forward exploration* through the state space by iterating successively transition successor relations from state sets (see, for example [15, 7]). In this case, a set of states computed at any point during the algorithm is a pair comprising of a location and a set of clock valuations called a *zone*. Such a pair is called a *symbolic state*, and for any classical or probabilistic timed automaton, the number of symbolic states is finite [7]. *Probabilistic timed reachability properties* of probabilistic timed automata can be verified by defining a “symbolic state Markov decision process”, the states of which are the generated symbolic states, and the transitions of which are derived from the probabilistic timed automaton's distributions and edges [14].

3 Modelling the Root Contention Protocol

We now explain how the IEEE1394 root contention protocol can be specified and verified formally within the framework of probabilistic timed automata. The classical timed automaton models of [17] are taken as the basis for the probabilistic models introduced here. First observe that the model $\mathtt{Node}_i, i \in \{1, 2\}$, of the ith node of the protocol can be extended with probability in a straightforward manner; simply let the two edges corresponding to the outcome of a coin flip each be assigned probability 0.5 by a distribution. All other edges in the node model, and in the wire model, can be taken with probability 1. The parallel composition $\mathtt{Impl}^p$ of the resulting probabilistic timed automata can be defined in the context of an extension of the model of [14] with synchronising events.

During initial verification experiments with HyTech, however, it became clear that analysis of a model of the size of the composed probabilistic timed automaton is not feasible given our current implementation of the forward reachability algorithm of [14]. Therefore, we also study the smaller, more abstract probabilistic timed automaton $\mathtt{I}_1^p$ of the root contention protocol. The model for $\mathtt{I}_1^p$ is shown in Figure 1, and is a probabilistic extension of the classical timed automaton $\mathtt{I}_1$ of [17]. Each distribution (denoted by the dotted line over edges) corresponds to a coin being flipped; for example, in the location start_start, there is a nondeterministic choice corresponding to node 1 (re)starting the root contention protocol and flipping its coin, leading with probability 0.5 to each of slow_start and fast_start. The explanation of the remainder of the model follows from that of [17], to which the reader is referred. The timing constraints used in $\mathtt{I}_1^p$ correspond to those specified in the updated standard IEEE1394a; we also set the transmission delay be 360ns.

As $\mathtt{I}_1^p$ represents an *abstraction* of the root contention protocol, it may exhibit extraneous behaviours that do not correspond to protocol execution. This means that the most unfavourable

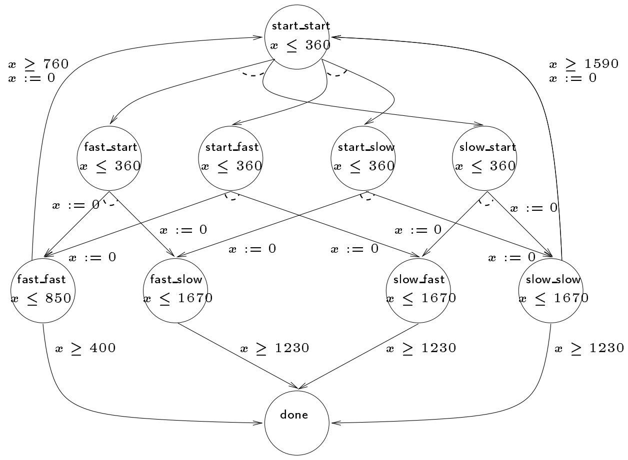


Fig. 1. The probabilistic timed automaton $\mathtt{I}_1^p$.

behaviour with respect to the satisfaction of a property of the protocol must also be exhibited by $\mathtt{I}_1^p$. Hence the minimal probability of property satisfaction that is computed for $\mathtt{I}_1^p$ will form a *lower bound* on the probability in the protocol, and the verification results presented in the next section reflect this. We plan to formalise the way in which $\mathtt{I}_1^p$ is a probabilistic, timed abstraction of the probabilistic timed automaton $\mathtt{Impl}^p$, described at the start of this section, in the full paper.

4 Verification and Analysis

We now report on the verification of a probabilistic deadline property of the root contention protocol using HyTech and Prism. The property of interest requires that the system will have elected a leader before a certain deadline has expired with a certain probability or greater, *for all* ways in which system events may be timed. In contrast, the forward reachability procedure of [14] analyzes properties considering the *existence* of a behaviour in which a certain location is reached with a given probability or greater. The former type of property can be represented in terms of the latter given adjustments to the structure of the model in question (following precedents for classical timed automata such as [16]), as we now illustrate with the probabilistic timed automaton $\mathtt{I}_1^p$. We augment $\mathtt{I}_1^p$ with the additional clock y, which is only reset at the commencement of system execution, and the new location deadline_exceeded (which has no outgoing edges); furthermore, from all locations apart from done, we add a distribution over a single edge with the target location deadline_exceeded. The enabling condition of each edge takes the form $y > deadline$, where *deadline* refers to the value of the time bound considered. We denote the new probabilistic timed automaton resulting from these changes by $\mathtt{I}_1^{p+}$. Then the target location of the forward reachability property is taken to be deadline_exceeded. If the maximal probability with which deadline_exceeded can *possibly* (existentially) be reached is λ, then $1 - \lambda$ will be the maximal probability with which deadline_exceeded will *inevitably* (universally) be avoided. A more formal explanation, in which such properties are represented as probabilistic temporal logic formulae, and which uses the duality rules for such logic given in [4], is deferred to the full paper.

Although, in general, the forward reachability process described in the Section 2 computes an *upper* bound on the reachability probability in the probabilistic timed automaton, for certain classes of system, the technique will result in the computation of the *exact* probability; indeed, the probabilistic timed automaton $\mathtt{I}_1^{p+}$ falls into this class. An intuitive and formal explanation of this phenomenon will be included in the full paper.

The verification of the probabilistic timed automaton $\mathtt{I}_1^{p+}$ against the reachability property concerning the location deadline_exceeded takes the following form.

Deadline (ns)	Symbolic states	Probability
2000	48	0
2250	48	0.5
2500	68	0.5
2750	72	0.5
3000	72	0.5
3250	102	0.625
3500	103	0.625
3750	107	0.625
4000	137	0.625
4250	138	0.75
4500	142	0.78125
4750	172	0.78125
5000	183	0.78125
5500	218	0.84375
6000	234	0.851562

Table 1. Results from verification using HyTech and Prism.

- Firstly, we generate the relevant symbolic states, together with information concerning transitions made between such symbolic states, using commands written in the analysis language of HyTech. These commands form an implementation of the forward reachability algorithm of [14]. Note that a number of auxiliary, discrete variables are added to the model so that sufficient information concerning the transition relation between the symbolic states is maintained.
- Secondly, we perform probabilistic model checking on the resulting "symbolic state Markov decision process" against a PCTL property which refers to the maximum probability of reaching a symbolic state with the location component deadline_exceeded.

The results obtained using this method are shown in Table 1. The left-hand column shows the deadline used in the property; the centre column gives the number of symbolic states generated by the forward reachability algorithm, as implemented in HyTech; and the third column shows the minimum probability with which the system has reached a leader elected state before the deadline (which is one minus the maximum of the existential reachability property computed by Prism). The results reflect the obvious fact that increasing the time bound has the effect of increasing the probability; intuitively, the greater the deadline, the "more chances" the system has of flipping different results for each node (therefore electing a leader). Observe that the rate of increase both in the number of computed symbolic states and the probability proceeds in "jumps" as the deadline increases. In particular, the probability of satisfying the required property is dependent on how many protocol restarts (and therefore coin flips) the system can perform before expiration of the deadline. Although the generation of probabilities for low values of the deadline (particularly 3000 or less) is trivial, the probability computation becomes more involved for greater deadlines.

5 Conclusions

We have considered an approach to the specification of both timed and probabilistic aspects of the root contention protocol of IEEE1394 using probabilistic timed automata, and have investigated the way in which the tools HyTech and Prism may be used to verify the resulting model. The timing parameters from the updated IEEE1394a standard were used in the model, although observe that it is straightforward to change the values of the parameters and re-run the experiments if required.

Note that our approach can be used as a basis for the computation of an approximation of the *expected time* to leader election; it would therefore be of interest to formalise such a notion in the context of probabilistic timed automata, and to relate it to other precedents in the probabilistic model checking literature, for example [8]. One aspect that is missing from our approach is an automatic method for verifying that the probability of the system electing a leader within a deadline of infinity is 1. This requires a thorough investigation into such a class of probabilistic "liveness" properties, which cannot be analysed by use of the reachability algorithm of [14].

The verification of the root contention protocol was carried out as part of an ongoing activity concerning the practical implementation of model checking methods for probabilistic timed automata. One lesson learned was that, although the size of the generated state spaces can be handled easily by the tool PRISM, the high memory usage of our algorithm implementation in HYTECH proved to be the bottleneck of the analysis, and meant that we were able to perform verification of only short deadlines. Experimentation with other real-time verification tools, such as UPPAAL, may prove to be useful.

References

1. R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking for probabilistic real-time systems. In J. L. Albert, B. Monien, and M. A. Rodríguez, editors, *Proceedings of the 18th International Conference on Automata, Languages and Programming (ICALP'91)*, volume 510 of *LNCS*, pages 115–136. Springer-Verlag, 1991.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In J. Baeten and S. Mauw, editors, *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 142–162. Springer-Verlag, 1999.
4. C. Baier and M. Z. Kwiatkowska. Model checking for a probabilistic branching time logic with fairness. *Distributed Computing*, 11(3):125–155, 1998.
5. G. Bandini, R. L. Spelberg, and W. J. Toetenel. Parametric verification of the IEEE1394a root contention protocol using LPMC. Submitted, 2000.
6. P. R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. PhD thesis, Department of Computer Science, University of Twente, 1999.
7. C. Daws and S. Tripakis. Model–checking of real–time reachability properties using abstractions. In Steffen [18], pages 313–329.
8. L. de Alfaro. How to specify and verify the long-run average behavior of probabilistic systems. In V. Pratt, editor, *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 454–465. IEEE Computer Society Press, 1998.
9. L. de Alfaro. Stochastic transition systems. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the 9th International Conference on Concurrency Theory (CONCUR'98)*, volume 1466 of *LNCS*, pages 423–438. Springer-Verlag, 1998.
10. L. de Alfaro, M. Z. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using MTBDDs and the Kronecker representation. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2000)*, volume 1785 of *LNCS*, pages 395–410. Springer-Verlag, 2000.
11. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.
12. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
13. J.-P. Katoen, editor. *Proceedings of the 5th International AMAST Workshop on Real-Time and Probabilistic Systems (ARTS'99)*, volume 1601 of *LNCS*. Springer-Verlag, 1999.
14. M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Theoretical Computer Science*, 2001. To appear. A preliminary version of this paper appeared in Katoen [13], pages 75–95.
15. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.
16. M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear-box controller. In Steffen [18], pages 281–297.
17. D. P. L. Simons and M. I. A. Stoelinga. Mechanical verification of the IEEE1394a root contention protocol using UPPAAL2k. Technical Report CSI-R009, Computing Science Institute, University of Nijmegen, 2000.
18. B. Steffen, editor. *Proceedings of the 4th Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *LNCS*. Springer-Verlag, 1998.
19. M. I. A. Stoelinga and F. Vaandrager. Root contention in IEEE1394. In Katoen [13], pages 53–74.

On model checking data-independent systems with arrays (work in progress)

Ranko Lazić, Tom Newcomb and Bill Roscoe

Abstract

A system is data-independent with respect to a data type X iff the only operation it can perform on values of type X is equality testing. The system may also store, input and output values of type X. This can be formalised with X being a type variable.

We study model checking of systems which are data-independent with respect to two distinct type variables X and Y, and may in addition use arrays with indices from X and values from Y. The main problem of interest is whether a given system satisfies a given temporal-logic formula for all instances of X and Y. We investigate decidability of this problem depending on the temporal logic used, and on characteristics of the system such as available array operations, number of arrays and available kinds of equality testing.

Practical motivation for model checking data-independent systems with arrays includes verification of memory systems of shared-memory multi-processors, where X is the type of memory addresses, and Y the type of storable values.

On Definability of Parametric Families of Labelled Transition Systems*

Ranko Lazić[1]** and David Nowak[2]

[1] Department of Computer Science, University of Warwick
Ranko.Lazic@dcs.warwick.ac.uk
[2] Computing Laboratory, University of Oxford
David.Nowak@comlab.ox.ac.uk

Abstract. We define symbolic labelled transition systems (for short, SLTSs) as a formalism to represent concurrent systems. An SLTS is parameterised by its signature, which consists of type variables and operations on them. Its semantics is given by a labelled transition system (for short, LTS) for each instantiation of the signature, and this family is always parametric. Intuitively, parametricity means that all members of the family behave the same way. This is because an SLTS can only use its signature in a polymorphic fashion. The main contribution of this research [1] is a theorem which states that any parametric family of LTSs (with some restrictions on types) is definable by an SLTS. To the best of our knowledge, it is the first definability result in the presence of nondeterminism.
Keywords: definability, nondeterminism, symbolic LTSs

References

1. R. Lazić and D. Nowak. On definability of parametric families of labelled transition systems. Technical report, Department of Computer Science, University of Warwick, 2001. To appear.

* This research was supported in part by the EPSRC Standard Research Grant 'Exploiting Data Independence', GR/M32900.

** Also affiliated to the Mathematical Institute, Belgrade. This author was supported in part by a Junior Research Fellowship from Christ Church, Oxford, and previously by a scholarship from Hajrija & Boris Vukobrat and Copechim France SA.

Temporal Logic Model Checking of CSP: Tools and Techniques

(abstract)

Michael Leuschel[1], Ivan Wolton[1], Thierry Massart[2] and Laksono Adhianto[1]

[1] Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton, SO17 1BJ, UK
e-mail: {mal,ajc}@ecs.soton.ac.uk

[2] Computer Science Department
University of Brussels ULB - CP 212
Bld du Triomphe, B-1050 Brussels, Belgium
e-mail: tmassart@ulb.ac.be

Abstract. We study the possibility of doing LTL model checking on CSP specifications in the context of refinement. We present a technique to perform LTL model checking of CSP processes using refinement checking in general and the FDR tool in particular. We present a tool which automates the translation process from LTL model checking to CSP refinement. Also, if time permits, we will present another tool which uses latest generation Prolog technology to symbolically animate, compile, and model check CSP specifications.

1 Introduction

Recent years have seen dramatic growth [4] in the application of model checking [3, 2] techniques to the validation and verification of correctness properties of hardware, and more recently software systems.

One of the methods is to model a hardware or software system as a finite, labelled transition system (LTS) which is then exhaustively explored to decide whether a given temporal logic specification ϕ holds, i.e., *checking* whether the system is a *model* of the formula ϕ. This approach has lead to various implementations, such as SPIN [7] for model checking of formulas in LTL [1] or SMV [9] for CTL [3] model checking.

Another, quite different, approach is based on the notion of *refinement* and is used by tools such as FDR [10, 5]. Here, the idea is to model *both* the system and the property in the *same* formalism, e.g., as CSP [6] processes. A system is said to satisfy a property ϕ if it is a refinement of ϕ. In CSP, refinement can be defined as language containment, failures containment, or failures and divergences containment.

The refinement-based approach suits itself very nicely to the stepwise development of systems, while the temporal logic approach often allows for more natural or succinct temporal specifications. It is quite surprising that the relationship between these two approaches appears not to have been studied. For instance, on the practical side, one might be interested in using tools such as FDR to do classical temporal logic model checking of CSP processes. In that context it would be interesting to know how (subclasses of) LTL or CTL temporal logic formulas can be translated into refinement tests. On the theoretical side, one might be interested in studying the expressive power of full LTL or CTL compared to refinement based model checking.

In this talk, we study the possibility of doing LTL model checking on CSP specifications in the context of refinement in general and using the FDR tool in particular. We discuss some unfruitful attempts at this translation, which show that it is surprisingly difficult to find intuitive formulations of classical model checking tasks as refinement checks. In particular, this means that a tool such as FDR can currently not be used by ordinary users to perform LTL or CTL model checking. This is a pity, as FDR can handle full CSP extended with functions and advanced datatypes (such as lists, integers, sets), thus providing a powerful specification and prototyping language, and it would be extremely valuable to apply "classical" model checking to such specifications (e.g., to validate the initial specification).

To remedy this problem, we will present a translation from LTL to refinement which *does* work and allows one to perform "classical" model checking in a CSP setting with refinement (full details about the technique can be found in [8]). We will also present a tool (currently under development) which automates this transformation process. This tool is given a LTL formula which is first translated into a Büchi automata and then further into a CSP tester process. This tester process is then combined with the CSP specification/program to be verified and two refinement conditions are generated, which are then checked using FDR. We also show how our approach can be used to perform some "non-propositional" model checking tasks (such as $\forall c, o. \Box(reqtoks?c?o \Rightarrow \exists t. \Diamond colltoks!c!o?t)$). Finally, if time permits, we will also present another tool which uses latest generation Prolog technology to symbolically animate, compile, and model check CSP specifications.

History of the paper: This paper arouse out of discussions with proponents of refinement (and CSP) who proclaimed that "One does *not* need LTL model checking (for CSP or FDR), one can always write a specification describing all admissible behaviours and then checking that the system under consideration is a valid *refinement* of that specification." As we showthis turns out to be extremely difficult (or even impossible in general). We have thus developed another, tester based approach, which can be fully automated. This paper thus underlines that "One *does* need LTL model checking, as one can *not* always (easily) write a specification describing all admissible behaviours and then checking that our system is a valid refinement of that specification."

References

1. A.Pnueli. The temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
2. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
3. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
4. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, Dec. 1996.
5. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual*.
6. C. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
8. M. Leuschel, T. Massart, and A. Currie. How to make FDR spin: LTL model checking of CSP by refinement. Technical Report DSSE-TR-2000-10, Department of Electronics and Computer Science, University of Southampton, September 2000. To appear in Proceedings of FME'2001.
9. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.
10. A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

Quantifying Information Flow

Gavin Lowe*

March 30, 2001

Abstract

We extend definitions of information flow so as to *quantify* the amount of information passed; in other words, we give a formal definition of the capacity of covert channels. Our definition uses the process algebra CSP, and is based upon counting the number of different behaviours of a high level user that can be distinguished by a low level user.

1 Introduction

Previous work, for example [1, 13, 4, 11, 6, 5], has sought to capture the notion of *information flow* (sometimes called non-interference or independence) in a multi-level security system, that is the question of whether a high level user, High, can pass information to a low level user, Low, via a covert channel. Most definitions are based around asking whether Low can distinguish between two different behaviours of High, so that High can use the system to pass at least one bit of information to Low.

However, in many circumstances, some flow of information will be inevitable and acceptable, providing it is not too high. For example, the Orange Book [16] includes requirements for the estimation of the capacity of covert channels, and recommendations for acceptable values.

In this paper we extend the previous work on defining information flow, and produce a definition, using the process algebra CSP, of the *quantity* of information flow. We will define the *information flow quantity* to be the number of behaviours of High that are distinguishable from Low's point of view. If there are N such distinguishable behaviours, then High can use the system to encode an arbitrary number in the range $0, \ldots, N-1$ to send it to Low. Information flow is often measured in bits: we can easily calculate this as $\log_2 N$. (A disadvantage of this definition, that the reader should be aware of, is that an absence of information flow is represented by an information flow quantity of 1, rather than 0.)

We will define the quantity of information flow in both untimed and timed models. One advantage of working in a timed model is that we can define the quantity of information flow within some finite time period, and hence the *rate* of information flow.

*Author's address: Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; e-mail: gavin.lowe@comlab.ox.ac.uk.

In the following section, we consider some example processes, and for each give the information flow quantity, thus helping to describe our intuitions. In Section 3, we formalise the notion and give our main definition; we illustrate it with some examples. The definition is slightly non-intuitive, and proved surprisingly difficult to get right: a number of simpler definitions fail to give the right answer for example processes. We show that those processes for which our definition gives an information flow quantity of 1—i.e. no information flow—are precisely those processes that satisfy a more intuitive definition of lack of information flow, similar to that in [4, 3].

In Section 4, we extend our definition to timed processes. In Section 5 we consider the amount of information passed within a bounded amount of time, thus leading to a definition of the capacity of covert channels in bits per time unit. Finally, we sum up and give pointers to future work in Section 6. A brief overview of CSP and Timed CSP can be found in Appendix A.

2 Thought experiments

To help motivate the problem further, we consider some example processes, and in each case quantify the information passed, thus explaining our intuitions.

Throughout this paper we will let H be the set of events for High, which will have names like h, and we will let L be the set of events for Low, which will have names like l.

Note that we are working in an untimed model: any information flow will be as a result of different events becoming available—or not—to Low, as opposed to the times at which they become available. In particular, Low will only be able to observe that an event is unavailable if it is unavailable for all time. This is clearly an abstraction, as untimed models always are; one could think of this model as a stepping stone towards the timed model.

Example 1 Consider

$$P_1 \;\widehat{=}\; h \to l \to STOP.$$

This has an information flow quantity of 2. If High chooses to perform an h then Low will be able to perform an l; if High chooses not to perform an h then Low will not be able to perform an l. Looked at from the other side, if Low is able to perform l then he can deduce that High performed h; and if Low sees l refused—that is, l is unavailable for all time—then he can deduce that High did not perform h. Thus High can use P_1 to pass one bit of information to Low.

Example 2 Consider:

$$\begin{aligned} P_2 \;\widehat{=}\; & h1 \to (l1 \to STOP \sqcap l2 \to STOP) \\ & \Box\, h2 \to (l1 \to STOP \sqcap l2 \to STOP). \end{aligned}$$

This process certainly has an information flow of at least 2, for Low can tell whether or not High has performed some event. The question is whether Low can distinguish the two behaviours of the system following $h1$ and $h2$ events. If the two nondeterministic choices were implemented identically, then clearly the answer is no (and the information flow quantity would be 2). However, the normal intuition in CSP is that a nondeterministic choice represents under-specification: this can be resolved by the implementer deciding how to implement the under-specification, or it can be resolved by some mechanism at run time. For example, if the first nondeterministic choice were implemented to always select its first argument, and the second nondeterministic choice were implemented to always select its second argument, then Low would indeed be able to distinguish the two behaviours, and so the information flow quantity would be 3.

We indeed take the information flow quantity of this process to be 3, being the higher of the above possible values; in other words, we consider the worst case scenario.

Example 3 Let

$$\begin{aligned} P_3 \;\widehat{=}\; & h1 \to l1 \to STOP \\ & \Box\, h2 \to l2 \to STOP \\ & \Box\, h3 \to (l1 \to STOP \sqcap l2 \to STOP). \end{aligned}$$

This has an information flow quantity of 3, as with the process $h1 \to l1 \to STOP \Box h2 \to l2 \to STOP$. The $h3$ branch does not add any information flow: if Low can perform an $l1$ then he can deduce that High performed either $h1$ or $h3$, but he cannot tell which; similarly, if Low can perform an $l2$ then he can deduce that High performed either $h2$ or $h3$, but he cannot tell which.

We are assuming that Low cannot make copies of the system (for example, via something like the Unix `fork` command) so as to perform repeated experiments: otherwise, he might be able to make a copy of the system after High has performed the event $h3$, and then in repeated experiments observe both $l1$ and $l2$ events (corresponding to different resolutions of the nondeterministic choice) and hence deduce that $h3$ had indeed been performed. This relates to our choice of model, namely the Failures Model of CSP; if Low were able to make copies of the system, then a model based around bisimulation would be appropriate, because bisimulation allows the observer to find out more about the branching structure of a process.

Example 4 Let

$$P_4 \;\widehat{=}\; h1 \to l1 \to STOP \Box h2 \to (l1 \to STOP \Box l2 \to STOP).$$

Depending upon how High acts, from Low's point of view the system will appear to act like one of the processes:

$$STOP, \qquad l1 \to STOP, \qquad l1 \to STOP \Box l2 \to STOP.$$

Our model will say that Low is not able to distinguish between all three of these processes (although he can distinguish between any pair of them) and so the information flow quantity is two. For example, if Low tries to perform $l2$, and it is refused, then he cannot tell whether he is looking at the first or second process. Alternatively, if Low manages to perform $l1$, he cannot tell whether he is looking at the second or third process.

One experiment that Low might—depending on the assumptions one makes—use to distinguish the three processes is to attempt the event $l2$, and if that is refused to then attempt the event $l1$. However, the normal assumption in the Failures Model of CSP is that one cannot observe a refusal, and subsequently observe an event. Thus we will not allow the above experiment.

The above assumption is far from clear cut. If Low is able to detect that internal activity by the process has ceased (so the events available will not change), then it is reasonable to suppose that Low can detect a refusal and then perform further events. In this case, the information flow quantity for the above process will be 3, and the model we present in this paper will not be suitable. This latter philosophy is that of the Refusal Testing Model [8]; we briefly consider a model of information flow based on refusal testing in the conclusion. It is also the philosophy of the Timed Failures Model of CSP, and so the timed model we present in Section 4 would allow an experiment similar to the one above.

Example 5 Consider

$$P_5 \ \widehat{=}\ l1 \to h \to l3 \to STOP \,\Box\, h \to l2 \to STOP.$$

The information flow quantity for this process depends upon the assumptions we make about High: we will assume that High can choose whether to perform his event before or after Low has performed $l1$, and so the quantity is 3, corresponding to the cases where High performs h before $l1$, after $l1$, or not at all. In effect, we allow High to observe Low's event.

Example 6 Consider

$$P_6 \ \widehat{=}\ h \to l \to l \to STOP \,\Box\, l \to STOP.$$

If Low is able to perform two l events, then he can deduce that High performed h; however, if Low is able to perform only one l, he cannot tell whether or not High was attempting to perform h: High's h might have been preempted by Low's l. High and Low might be able to use this process to pass one bit of information, but they cannot reliably do so. We again consider the worst case scenario, and take the information flow quantity to be 2, for clearly there is the possibility of some flow of information.

An alternative way of considering this process is that High might be able to ensure that his h is performed before Low's first l; for example, they might have agreed a strategy that High does his event at time 0, and Low attempts his event at time 1; alternatively, even if they both attempt their event at the same time, some property of the system, below the level of abstraction at which we are modelling it, might resolve the resulting nondeterminism in favour of High, i.e. give priority to High.

In effect, we allow High to have some limited control over events available to Low, preventing them from happening by preempting them with his own events.

3 Quantifying information flow in an untimed model

In this section we formalise information flow quantity, by considering the way in which High and Low interact with the system in order for High to pass information to Low. We assume that the behaviour of the system is modelled by a divergence-free CSP process P.

It would seem natural to model High's behaviour by a non-divergent CSP process Q with alphabet H; we write CSP_H for the set of such processes. In this case Low's view of the system would be given by $(P \underset{H}{\|} Q) \setminus H$. This is the approach considered in [3, 4], where a process is defined to satisfy *Failures Non Deducibility on Composition (FNDC)* if all such views are equal. We are interested (roughly speaking) in *how many* such views are distinguishable.

However, it turns out that this won't allow us to model the assumptions about High from Section 2, especially those in Examples 5 and 6. In particular, we need to assume that High is able to arrange for his events to occur either before or after events of Low.

Instead, we use a modelling trick: we include Low's events within the alphabet of the process Q. Low's view of the system will then be given by the process $(P \underset{\Sigma}{\|} Q) \setminus H$. However, we do not want to allow all such processes Q; in a state of P where only events of Low are available, High should not be able to block any of those events: High can only prevent events of Low from occurring by preempting them with one of his own events. For example, if $P = l \to STOP$, then the Q process $STOP$ should not be allowed, but $l \to STOP$ or $RUN(L)$ would be allowed. We therefore restrict ourselves to processes Q satisfying the following condition for the given P:

$$\forall (s, X) \bullet (s, X \cup H) \in \mathcal{F}[\![P \underset{\Sigma}{\|} Q]\!] \Rightarrow (s, X \cap L) \in \mathcal{F}[\![P]\!].$$

This says that in every state of the system where no events from H are available, if any events from L are refused, it is because P is refusing those events.

This condition prevents examples like the following:

$$\begin{aligned} P \;&\widehat{=}\; h1 \to (l1 \to STOP \;\Box\; l2 \to STOP) \\ &\quad\;\Box\; h2 \to (l1 \to STOP \;\Box\; l2 \to STOP), \\ Q \;&\widehat{=}\; h1 \to l1 \to STOP. \end{aligned}$$

This shouldn't be allowed: High shouldn't be able to determine which of $l1$ and $l2$ becomes available for Low after $h1$. This is forbidden by the above condition: $P \underset{\Sigma}{\|} Q$ has failure $(\langle h1 \rangle, \{l2\} \cup H)$, but P does not have failure $(\langle h1 \rangle, \{l2\})$.

In fact, as explained in Example 2, we have to consider all ways in which nondeterminism in P might be resolved. We will do this by considering all refinements of P—that is, all the ways in which P might act—and so we require that all refinements R of P satisfy the above equation:

$$\forall R \sqsupseteq P \bullet \forall (s, X) \bullet (s, X \cup H) \in \mathcal{F}[\![R \underset{\Sigma}{\parallel} Q]\!] \Rightarrow (s, X \cap L) \in \mathcal{F}[\![R]\!]. \qquad (1)$$

The condition is satisfied by the process $RUN(L)$, and so we will make frequent use of this process when defining strategies for High.

For simplicity, we restrict ourselves to processes P where $(P \underset{\Sigma}{\parallel} Q) \setminus H$ is non-divergent for all non-divergent Q; this will be true if P itself is non-divergent and cannot perform an unbounded number of consecutive high level events.

The following lemma shows that under the above conditions, considering compositions of the form $(P \underset{\Sigma}{\parallel} Q) \setminus H$ does not make more pairs of processes distinguishable from Low's point of view than by considering compositions of the form $(R \underset{H}{\parallel} Q') \setminus H$.

Lemma 1 If Q_0 and Q_1 satisfy equation (1), and $(P \underset{\Sigma}{\parallel} Q_0) \setminus H \not\equiv_F (P \underset{\Sigma}{\parallel} Q_1) \setminus H$ then there exist $R \sqsupseteq P$ and $Q_0', Q_1' \in CSP_H$ such that

$$(R \underset{H}{\parallel} Q_0') \setminus H \not\equiv_F (R \underset{H}{\parallel} Q_1') \setminus H.$$

(We conjecture that the lemma can be strengthen so that the result talks about P rather than a refinement R of P.) This lemma is proved in [7].

We presume that High and Low have devised some strategy for passing values in the range $\{0, \ldots, N-1\}$. What form must that strategy take? We suppose that for each value k that High wants to pass, he will act like some process $\mathcal{Q}(k)$; we will therefore model High's behaviour as a function of the form[1] $\mathcal{Q} : \mathbb{N} \nrightarrow CSP_H$ where every $\mathcal{Q}(k)$ satisfies equation (1).

Given some particular High strategy $\mathcal{Q}$, Low's possible views of the system will be given by $\{(P \underset{\Sigma}{\parallel} \mathcal{Q}(k)) \setminus H \mid k \in \operatorname{dom} \mathcal{Q}\}$. However, there might be no strategy for Low that distinguishes all of these. For example, if P is the process

$$\begin{array}{l} h1 \rightarrow (l \rightarrow l1 \rightarrow STOP \,\Box\, l' \rightarrow l1 \rightarrow STOP) \\ \Box\, h2 \rightarrow (l \rightarrow l2 \rightarrow STOP \,\Box\, l' \rightarrow l1 \rightarrow STOP) \\ \Box\, h3 \rightarrow (l \rightarrow l1 \rightarrow STOP \,\Box\, l' \rightarrow l2 \rightarrow STOP) \\ \Box\, h4 \rightarrow (l \rightarrow l2 \rightarrow STOP \,\Box\, l' \rightarrow l2 \rightarrow STOP), \end{array}$$

then the above set has 19 members, corresponding to different ways that High can nondeterministically choose different sets of events to offer[2]. Even if we

[1] The notation $A \nrightarrow B$ represents *partial* functions from A to B.

[2] The figure 19 is derived as follows. If High always offers at least one event, then l and l' will be available; after an l either $l1$ will be available, or $l2$ will be available, or it will be nondeterministic which is available; the same is true after an l'; this makes nine possibilities. In addition, High could nondeterministically offer events as above or offer no event, adding another nine possibilities. Or High could deterministically offer no event.

restrict High to deterministically offering at most one event at a time, then the above set will have 5 members (up to failures-equivalence); however, Low cannot distinguish all of these, because he has to commit to exploring either the l-branches or the l'-branches. We return to this example in Section 3.1.

Low will interact with the system by repeatedly offering a set of events and either seeing one of them accepted, or seeing them all refused; Low will associate results with particular sequences of event acceptances and refusals. We will call Low's strategy for interacting with a process a *test*. The precise form of the tests will reflect the assumptions we make about Low, discussed informally above, in particular that Low cannot offer further events after observing a refusal.

Traditional approaches to testing (e.g. [15]) make use of a test process $SUCCESS$, which represents successful completion of a test. We extend this to use test processes of the form $SUCCESS(k)$, for $k \in \mathbb{N}$, to represent successful completion of the test, giving a result of k.

We also write $(\Box_{a \in A}\, a \to T_a)$ else $SUCCESS(k)$ to represent a test that attempts the set of event A (where $A \subseteq L$), and which then acts like T_a if a particular event a is performed, and which gives the result k if all the events are refused.

We can therefore define failures-based tests using a BNF-style grammar as follows:

$$T \in \mathit{Test} ::= SUCCESS(k) \mid (\Box_{a \in A}\, a \to T_a) \text{ else } SUCCESS(k).$$

We consider only *finite* tests, i.e. where the test bottoms out to give a result after a bounded number of steps.

We will write the indexed external choice as a binary infix external choice, where convenient; for example, the test

$$T \mathrel{\widehat{=}} l1 \to SUCCESS(1) \Box l2 \to SUCCESS(2) \text{ else } SUCCESS(0)$$

represents Low offering $l1$ and $l2$, and giving the result 1 if $l1$ is accepted, the result 2 if $l2$ is accepted, and the result 0 if both are refused.

We associate a set of (trace, refusal, result) triples with each test, representing that the test can observe the given trace and refusal of the process it is testing, and then return the given result.

$$\begin{array}{l}
\mathcal{F}[\![SUCCESS(k)]\!] \mathrel{\widehat{=}} \{(\langle\rangle, \{\}, k)\}, \\
\mathcal{F}[\![(\Box_{a \in A}\, a \to T_a) \text{ else } SUCCESS(k)]\!] \mathrel{\widehat{=}} \\
\quad \{(\langle a \rangle ^\frown s, X, i) \mid a \in A \wedge (s, X, i) \in \mathcal{F}[\![T_a]\!]\} \cup \{(\langle\rangle, A, k)\}.
\end{array}$$

The test $SUCCESS(k)$ can immediately give the result k (following no trace or refusal information). The test $(\Box_{a \in A}\, a \to T_a)$ else $SUCCESS(k)$ can see an event $a \in A$ performed and then act like the test T_a; alternatively, it can see all of A refused and then give the result k.

We will write $\mathit{results}_F(S, T)$ for the set of results obtainable from failures-based test T with system S:

$$\mathit{results}_F(S, T) \mathrel{\widehat{=}} \{k \mid \exists (s, X) \in \mathcal{F}[\![S]\!] \bullet (s, X, k) \in \mathcal{F}[\![T]\!]\}.$$

The testing system can give a result k if S can display some failure (s, X) such that the test associates result k with this failure. The subscript "F" indicates that we are working in the Failures Model of CSP; we will drop the subscript where it is obvious from the context.

Note that if S is non-divergent—as we are assuming—then $results_F(S, T)$ will contain at least one value: for each state $(\Box_{a \in A}\, a \to T_a)$ else $SUCCESS(k)$ of T, the corresponding state of S will either be able to perform one of the events from A or refuse all of A; and we are considering finite tests so the testing system will eventually bottom out to give a result.

When we are talking about a system composed of the given process and a high level process, we will overload the *results* function to take those two processes as separate arguments:

$$results_F(P, Q, T) \;\widehat{=}\; results_F((P \underset{\Sigma}{\parallel} Q) \setminus H, T).$$

For example, recall $P_1 \widehat{=} h \to l \to STOP$, and consider the strategy defined by:

$$\mathcal{Q}(0) \widehat{=} RUN(L), \qquad \mathcal{Q}(1) \widehat{=} h \to RUN(L),$$
$$T \widehat{=} l \to SUCCESS(1) \text{ else } SUCCESS(0).$$

Then $results_F(P_1, \mathcal{Q}(0), T) = \{0\}$ and $results_F(P_1, \mathcal{Q}(1), T) = \{1\}$, corresponding, respectively, to the failure $(\langle\rangle, \{l\})$ of $(P_1 \underset{\Sigma}{\parallel} \mathcal{Q}(0)) \setminus H$, and the failure $(\langle l \rangle, \{\})$ of $(P_1 \underset{\Sigma}{\parallel} \mathcal{Q}(1)) \setminus H$.

We should only consider strategies of High and Low that are compatible, in the sense that if High wants to send a value k, then Low can obtain the result k. We will write $ok_F(P, \mathcal{Q}, T)$ to capture this condition:

$$\begin{aligned} ok_F(P, \mathcal{Q}, T) \;\widehat{=}\; & \forall\, k \in \operatorname{dom} \mathcal{Q} \bullet results_F(P, \mathcal{Q}(k), T) = \{k\} \\ & \qquad\qquad\qquad \text{and } \mathcal{Q}(k) \text{ satisfies (1).} \end{aligned}$$

For example, the process P_1 satisfies this condition with the above strategy.

Note that we insist that the results caused by a particular $\mathcal{Q}(k)$ are precisely $\{k\}$: it is not enough simply for different $\mathcal{Q}(k)$ processes to lead to different sets of results. For example, the process and test

$$\begin{aligned} P \;\widehat{=}\; & h1 \to l1 \to STOP \\ & \Box\, h2 \to l2 \to STOP \\ & \Box\, h3 \to (l1 \to STOP \,\Box\, l2 \to STOP), \\ T \;\widehat{=}\; & l1 \to SUCCESS(1) \,\Box\, l2 \to SUCCESS(2) \text{ else } STOP \end{aligned}$$

give different sets of results for the four high level processes $\mathcal{Q}(0) \widehat{=} RUN(L)$, $\mathcal{Q}(1) \widehat{=} h1 \to RUN(L)$, $\mathcal{Q}(2) \widehat{=} h2 \to RUN(L)$ and $\mathcal{Q}(3) \widehat{=} h3 \to RUN(L)$; however the results from $\mathcal{Q}(3)$—namely 1 and 2—are both results that could be obtained from other high level behaviours—$\mathcal{Q}(1)$ and $\mathcal{Q}(2)$—and so this does not contribute any additional information flow.

Given some process P and some strategy represented by $\mathcal{Q} \in \mathbb{N} \nrightarrow CSP$ and $T \in Test$ such that $ok_F(P, \mathcal{Q}, T)$, the associated information flow is the number of different values that can be sent, i.e., $\# \operatorname{dom} \mathcal{Q}$. However, as discussed in Section 2, particularly Example 2, we want to consider not just P, but also all refinements R of P; the information flow quantity is then the maximum flow achievable over all such R, and over all corresponding $\mathcal{Q}$ and T, i.e., such that $ok_F(R, \mathcal{Q}, T)$. This prompts the following definition of information flow quantity:

$$
\begin{aligned}
IFQ_F(P) \;\widehat{=}\; \max\{\# \operatorname{dom} \mathcal{Q} \mid\ & \mathcal{Q} \in \mathbb{N} \nrightarrow CSP \wedge T \in Test \wedge R \in CSP \\
& \wedge P \sqsubseteq R \wedge ok_F(R, \mathcal{Q}, T)\}.
\end{aligned}
$$

For example, this definition gives an information flow quantity of 2 for the process P_1, as illustrated by the above $\mathcal{Q}$ and T. If the set in the above definition is unbounded, then we will define the information flow quantity to be infinite.

3.1 Examples

Recall the process $P_6 \;\widehat{=}\; h \to l \to l \to STOP \;\Box\; l \to STOP$. The appropriate strategy for High and Low to adopt is the following:

$$
\begin{gathered}
\mathcal{Q}(0) \;\widehat{=}\; RUN(L), \qquad\qquad \mathcal{Q}(1) \;\widehat{=}\; h \to RUN(L), \\
T \;\widehat{=}\; l \to (l \to SUCCESS(1) \;\mathsf{else}\; SUCCESS(0)) \;\mathsf{else}\; SUCCESS(0).
\end{gathered}
$$

Note that $\mathcal{Q}(1)$ prevents an initial l from occurring by preempting it with an h. (The final $SUCCESS(0)$ in T is unreachable, but is syntactically necessary.) With this strategy we have $results_F(P_6, \mathcal{Q}(0), T) = \{0\}$ and $results_F(P_6, \mathcal{Q}(1), T) = \{1\}$. Note also that the $\mathcal{Q}(k)$ processes satisfy equation (1): in particular, $P_6 \underset{\Sigma}{\|} \mathcal{Q}(1)$ cannot refuse H until after the h has occurred, after which $\mathcal{Q}(1)$ does not block events of L. Hence this process has an information flow quantity of at least 2. Clearly, there are at most two different ways that the process can behave from Low's point of view—either one or two ls will be available—and so the information flow quantity is exactly 2.

As another example, consider again

$$
\begin{aligned}
& h1 \to (l \to l1 \to STOP \;\Box\; l' \to l1 \to STOP) \\
& \Box\; h2 \to (l \to l2 \to STOP \;\Box\; l' \to l1 \to STOP) \\
& \Box\; h3 \to (l \to l1 \to STOP \;\Box\; l' \to l2 \to STOP) \\
& \Box\; h4 \to (l \to l2 \to STOP \;\Box\; l' \to l2 \to STOP).
\end{aligned}
$$

An information flow quantity of 3 can be obtained using the strategy

$$
\begin{aligned}
& \mathcal{Q}(0) \;\widehat{=}\; RUN(L), \quad \mathcal{Q}(1) \;\widehat{=}\; h1 \to RUN(L), \quad \mathcal{Q}(2) \;\widehat{=}\; h2 \to RUN(L), \\
& T \;\widehat{=}\; l \to (l1 \to SUCCESS(1) \;\Box\; l2 \to SUCCESS(2) \;\mathsf{else}\; SUCCESS(0)) \\
& \qquad \mathsf{else}\; SUCCESS(0).
\end{aligned}
$$

However, no higher information flow quantity can be obtained, for example by exploiting the l' branches: Low has to commit himself to either exploring the

l branches, or exploring the l' branches, or maybe nondeterministicly searching either, for example with a test like:

$$
\begin{array}{l}
l \to (l1 \to SUCCESS(1) \text{ else } SUCCESS(2)) \\
\Box\ l' \to (l2 \to SUCCESS(2) \text{ else } SUCCESS(1)) \\
\text{else } SUCCESS(0),
\end{array}
$$

(which would distinguish the High behaviours $RUN(L)$, $h1 \to RUN(L)$ and $h4 \to RUN(L)$); but in no case can Low explore all of the resulting process tree.

In general, it is difficult to find upper bounds on the information flow quantity, without considering all possible High behaviours and Low tests that might interact with the process in different ways. One normally has to resort to informal arguments like the above.

Finally, we consider an example that gives an infinite information flow quantity. Let $P \mathrel{\widehat{=}} h \to l \to P$. High can pass an arbitrary value k by performing k h events. Fix N, and consider the following strategy:

$$
\begin{array}{rcll}
\mathcal{Q}(0) & \widehat{=} & RUN(L), & \\
\mathcal{Q}(k+1) & \widehat{=} & h \to l \to \mathcal{Q}(k), & \text{for } k = 0, \ldots, N-1, \\
T(k) & \widehat{=} & l \to T(k+1) \text{ else } SUCCESS(k), & \text{for } k = 0, \ldots, N-1, \\
T(N) & \widehat{=} & SUCCESS(N). &
\end{array}
$$

Then this strategy, using the test $T(0)$, will pass a value in the range $0, \ldots, N$. (Remember the requirement for finite tests; this is why we have to consider a fixed N.) But N is arbitrary so the information flow quantity of this process is infinite.

3.2 Formal analysis

In this section we give a number of results about our model and definition. Some of the proofs are omitted because of lack of space; they can be found in [7]. Our main goal is to understand under what circumstances our definition gives a process an information flow quantity of 1—i.e., no information flow.

In [3], Riccardo Focardi defines a security property *Failures Non Deducibility on Composition (FNDC)*, which can be written in CSP as:

$$P \textbf{ sat } FNDC \mathrel{\widehat{=}} \forall\, Q \in CSP_H \bullet P \mathbin{\underset{H}{\|}} STOP \equiv_F (P \mathbin{\underset{H}{\|}} Q) \setminus H,$$

which is equivalent to:

$$\forall\, Q_0, Q_1 \in CSP_H \bullet (P \mathbin{\underset{H}{\|}} Q_0) \setminus H \equiv_F (P \mathbin{\underset{H}{\|}} Q_1) \setminus H.$$

We propose the following strengthening of FNDC: a process P satisfies *Strong Failures Non Deducibility on Composition (SFNDC)* as follows:

$$P \textbf{ sat } SFNDC \mathrel{\widehat{=}} \forall\, R \sqsupseteq P \bullet R \in FDNC.$$

Our main result is that our definition gives information flow quantity of 1 to precisely those processes that satisfy SFNDC. Hence our definition agrees with SFNDC on those processes that have no information flow.

We prefer the condition SFNDC to FNDC because it overcomes the following objection of Forster and Roscoe [6]. Let $LEAK$ be any insecure process, and consider the two processes $LEAK \sqcap CHAOS(L)$ and $LEAK \mathbin{|||} CHAOS(L)$. These processes have clearly insecure behaviours, coming from $LEAK$; however they satisfy $FNDC$, because the insecure behaviour is masked by behaviours of $CHAOS(L)$. These processes do not satisfy SFNDC, because they are refined by $LEAK$.

We begin by proving some results about our testing scheme. First, we show that refining a process reduces the set of results obtained from a given test.

Lemma 2 If $P \sqsubseteq Q$ then $\forall\, T \bullet results_F(P, T) \supseteq results_F(Q, T)$.

Proof: Suppose $P \sqsubseteq Q$, and let T be a test. Then

$$\begin{aligned} results(P, T) &= \{k \mid \exists (s, X) \in \mathcal{F}[\![P]\!] \bullet (s, X, k) \in \mathcal{F}[\![T]\!]\} \\ &\supseteq \{k \mid \exists (s, X) \in \mathcal{F}[\![Q]\!] \bullet (s, X, k) \in \mathcal{F}[\![T]\!]\} \\ &= results(Q, T). \end{aligned}$$

□

We now show that our testing scheme precisely captures failures equivalence.

Lemma 3 Suppose P and Q are non-divergent. Then

$$P \equiv_F Q \Leftrightarrow (\forall\, T \bullet results_F(P, T) = results_F(Q, T)).$$

Proof: The left-to-right implication follows immediately from Lemma 2.

For the converse, we prove the contrapositive. Suppose $P \not\equiv_F Q$. Then without loss of generality, there is some failure (s, X) such that $(s, X) \in \mathcal{F}[\![P]\!]$ but $(s, X) \notin \mathcal{F}[\![Q]\!]$. Let $s = \langle a_1, a_2, \ldots, a_n \rangle$. We construct a test that will succeed with value 1 after this failure, but will succeed with value 0, otherwise:

$$\begin{aligned} T \mathrel{\widehat{=}} a_1 \to (a_2 \to (\ldots (a_n \to (&\Box_{x \in X}\, x \to SUCCESS(0) \\ &\mathsf{else}\ SUCCESS(1)) \\ &\mathsf{else}\ SUCCESS(0)) \\ &\ldots) \\ &\mathsf{else}\ SUCCESS(0)) \\ &\mathsf{else}\ SUCCESS(0). \end{aligned}$$

Then $results(Q, T) = \{0\}$ but $results(P, T) \supseteq \{1\}$. □

In [2], de Nicola and Hennessy prove that must testing is equivalent to failures equivalence. Hence our testing scheme is equivalent to must testing.

If two high level processes can make the system look different from Low's point of view, then we can find some testing strategy to give an information flow quantity of 2.

Lemma 4 If $(P \underset{H}{\parallel} Q_0) \setminus H \not\equiv_F (P \underset{H}{\parallel} Q_1) \setminus H$ then there exist $R \sqsupseteq P$, $Q_0', Q_1' \in CSP$ satisfying equation (1), and $T \in Test$ such that

$$results(R, Q_0', T) = \{0\} \quad \text{and} \quad results(R, Q_1', T) = \{1\}.$$

The proof makes use of the test T from Lemma 3, which distinguishes processes according to whether or not they have a particular failure (s, X); the proof constructs a suitable refinement of P, and suitable high level processes, such that one of the resulting systems will always exhibit the failure (s, X), and the other system will never exhibit this failure. The full proof is in [7].

The following theorem identifies the circumstances under which our definition identifies no information flow, i.e. gives an information flow quantity of 1:

Theorem 5 A process P has information flow quantity 1 iff for all refinements R, High cannot change the way the process appears to Low:

$$IFQ_F(P) = 1 \Leftrightarrow \forall R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP_H \bullet (R \underset{H}{\parallel} Q_0) \setminus H \equiv_F (R \underset{H}{\parallel} Q_1) \setminus H.$$

The above theorem is equivalent to

$$IFQ_F(P) = 1 \Leftrightarrow P \textbf{ sat } SFNDC.$$

Proof: We prove the contrapositive. In one direction:

$$\begin{aligned}
& \neg(\forall R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP_H \bullet (R \underset{H}{\parallel} Q_0) \setminus H \equiv_F (R \underset{H}{\parallel} Q_1) \setminus H \\
\Leftrightarrow\ & \exists R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP_H \bullet (R \underset{H}{\parallel} Q_0) \setminus H \not\equiv_F (R \underset{H}{\parallel} Q_1) \setminus H \\
\Rightarrow\ & \langle \text{Lemma } 4 \rangle \\
& \exists R \sqsupseteq P \bullet \exists R' \sqsupseteq R \,;\, Q_0', Q_1' \in CSP \,;\, T \in Test \bullet \\
& \qquad results(R', Q_0', T) = \{0\} \wedge results(R', Q_1', T) = \{1\} \wedge Q_0',\ Q_1' \text{ satisfy } (1) \\
\Rightarrow\ & IFQ(P) \geqslant 2.
\end{aligned}$$

In the other direction:

$$\begin{aligned}
& IFQ(P) \geqslant 2 \\
\Rightarrow\ & \exists R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP \,;\, T \in Test \bullet \\
& \qquad results(R, Q_0, T) = \{0\} \wedge results(R, Q_1, T) = \{1\} \wedge Q_0,\ Q_1 \text{ satisfy } (1) \\
\Rightarrow\ & \langle \text{Lemma } 3 \rangle \\
& \exists R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP \bullet \\
& \qquad (R \underset{\Sigma}{\parallel} Q_0) \setminus H \not\equiv_F (R \underset{\Sigma}{\parallel} Q_1) \setminus H \wedge Q_0,\ Q_1 \text{ satisfy } (1) \\
\Rightarrow\ & \langle \text{Lemma } 1 \rangle \\
& \exists R \sqsupseteq P \,; \exists R' \sqsupseteq R \,;\, Q_0', Q_1' \in CSP_H \bullet (R' \underset{H}{\parallel} Q_0') \setminus H \not\equiv_F (R' \underset{H}{\parallel} Q_1') \setminus H \\
\Rightarrow\ & \neg(\forall R \sqsupseteq P \,;\, Q_0, Q_1 \in CSP_H \bullet (R \underset{H}{\parallel} Q_0) \setminus H \equiv_F (R \underset{H}{\parallel} Q_1) \setminus H.
\end{aligned}$$

□

4 Quantifying information flow in a timed model

We now extend the definition of the quantity of information flow to timed processes. We assume that the behaviour of the system is modelled by a Timed

CSP process P. Much of our development is very similar to as in the untimed setting; we concentrate on describing the differences.

We will model High's behaviour by a Timed CSP process Q, whose alphabet may include events from L. The system from Low's point of view will then look like $(P \underset{\Sigma}{\parallel} Q) \setminus H$. We will insist that Q satisfies the following condition, which is the timed equivalent of equation (1):

$$\begin{array}{l} \forall R \sqsupseteq P \bullet \forall (s, X) \bullet \\ \quad (s, X \cup [0, \mathrm{end}(s, X)) \times H) \in \mathcal{TF}[\![R \underset{\Sigma}{\parallel} Q]\!] \Rightarrow (s, X \upharpoonright L) \in \mathcal{TF}[\![R]\!]. \end{array} \tag{2}$$

We will model High's strategy as a function $\mathcal{Q} : \mathbb{N} \nrightarrow \mathit{TCSP}$ such that $\mathcal{Q}(k)$ satisfies equation (2) for all $k \in \mathrm{dom}\,\mathcal{Q}$.

As in the untimed case, Low will interact with the system via a test, offering events, and maybe seeing them observed. However, if none of the events are accepted within some time period, we will allow Low to timeout and change the events it offers. We will write $(\Box_{a \in A}\, a \to T_a) \overset{t}{\triangleright} T'$ to represent a test that attempts the set of events A (where $A \subseteq L$), and which then acts like T_a if a particular event a is performed; if all the events of A are refused for t time units, then the test times-out and acts like T'.

We will assume that Low does not have access to a clock. However, tests of the above form can be used to simulate a discreet-time clock, by timing out at regular intervals and counting the number of such timeouts.

However, we need to ensure that Low cannot simulate an arbitrarily accurate clock, which would be possible if we allowed Low to timeout after arbitrarily short amounts of time. For example, consider $h \to l \to \mathit{STOP} \overset{1}{\triangleright} \mathit{STOP}$. If Low could simulate an arbitrarily accurate clock, then a value in an arbitrary range $0, \ldots, N$ could by passed as follows: split the time interval $[0, 1)$ into N sub-intervals of length $1/N$; High could then pass a value $k \in \{0, \ldots, N-1\}$ by performing h during the kth such interval; Low could recover the value from the time at which l becomes available. As N is arbitrary, this would give an unbounded quantity of information flow, which seems unreasonable.

We will therefore assume a finite speed for Low: we assume a *minimum delay* $\delta > 0$, such that Low cannot timeout after times of less than δ. We will restrict tests of the above form to those that have $t \geqslant \delta$.

We will write $\mathit{Test}(\delta)$ for the set of tests with minimum delay δ, defined as follows:

$$T \in \mathit{Test}(\delta) \;::=\; \mathit{SUCCESS}(k) \mid (\Box_{a \in A}\, a \to T_a) \overset{t}{\triangleright} T', \qquad \text{where } t \geqslant \delta.$$

We will write $\mathit{WAIT}\; t\,; T$ as a shorthand for $(\Box_{a \in \{\}}\, a \to T_a) \overset{t}{\triangleright} T$, for $t \geqslant \delta$.

We associate a set of (timed trace, timed refusal, result) triples with each test, representing that the test can observe the given timed trace and refusal of the process it is testing, and then return the given result.

$$\mathcal{TF}[\![\mathit{SUCCESS}(k)]\!] \;\widehat{=}\; \{(\langle\rangle, \{\}, k)\},$$

$$
\begin{aligned}
\mathcal{TF}[\![(\Box_{a \in A}\, a \to T_a) \overset{t}{\triangleright} T']\!] \;\widehat{=}\; & \{(\langle(t', a)\rangle ^\frown s + t', [0, t') \times A \cup X + t', k) \mid \\
& \qquad a \in A \wedge (s, X, k) \in \mathcal{TF}[\![T_a]\!] \wedge t' \in [0, t)\} \\
& \cup \\
& \{(s + t, [0, t) \times A \cup X + t, k) \mid \\
& \qquad (s, X, k) \in \mathcal{TF}[\![T']\!]\}.
\end{aligned}
$$

The test $SUCCESS(k)$ can immediately give the result k, having observed no failures information. The test $(\Box_{a \in A}\, a \to T_a) \overset{t}{\triangleright} T'$ can see an event $a \in A$ performed at some time $t' \in [0, t)$, and then act like the test T_a, provided all of A is refused up to that time; alternatively, it can see all of A refused throughout $[0, t)$, and then act like the test T'.

We will write $results_{TF}(S, T)$ for the set of results obtainable from timed-failures-based test T with system S; that is, the set of results k such that there is some timed failure (s, X) that the system can perform, and such that the test associates result k with this failure:

$$
results_{TF}(S, T) \;\widehat{=}\; \{k \mid \exists (s, X) \in \mathcal{TF}[\![S]\!] \bullet (s, X, k) \in \mathcal{TF}[\![T]\!]\}.
$$

As in the untimed model, we will overload the *results* function as follows:

$$
results_{TF}(P, Q, T) \;\widehat{=}\; results_{TF}((P \underset{\Sigma}{\|} Q) \setminus H, T).
$$

However, we should only consider strategies of High and Low that are compatible:

$$
\begin{aligned}
ok_{TF}(P, \mathcal{Q}, T) \;\widehat{=}\; \forall\, k \in \operatorname{dom} \mathcal{Q} \bullet\ & results_{TF}(P, \mathcal{Q}(k), T) = \{k\} \\
& \text{and } \mathcal{Q}(k) \text{ satisfies (2).}
\end{aligned}
$$

The definition of information flow quantity is then analogous to as in the untimed model:

$$
\begin{aligned}
IFQ_{TF}(P) \;\widehat{=}\; \max\{\#\operatorname{dom} \mathcal{Q} \mid\ & \mathcal{Q} \in \mathbb{N} \nrightarrow TCSP \wedge T \in Test \wedge R \in TCSP \\
& \wedge P \sqsubseteq R \wedge ok_{TF}(R, \mathcal{Q}, T)\}.
\end{aligned}
$$

4.1 Example

Consider again the process:

$$
P \;=\; h \to l \to STOP \overset{1}{\triangleright} STOP.
$$

Let $N \;\widehat{=}\; \lceil 1/\delta \rceil$. Suppose for the moment that $1/\delta$ is not an integer. We can construct a strategy for High as follows: in order to pass a value $k \in \{0, \ldots, N-1\}$, perform h at time $k\delta$; and in order to pass the value N, do not perform h:

$$
\begin{aligned}
\mathcal{Q}(k) &= WAIT\ k\delta\,;\, h \to l \to STOP, \qquad \text{for } k \in \{0, \ldots, N-1\} \\
\mathcal{Q}(N) &= STOP.
\end{aligned}
$$

Then

$$\begin{aligned}
(P \underset{\Sigma}{\parallel} \mathcal{Q}(k)) \setminus H &= \mathit{WAIT}\, k\delta \,;\, l \rightarrow \mathit{STOP}, \qquad \text{for } k \in \{0, \ldots, N-1\} \\
(P \underset{\Sigma}{\parallel} \mathcal{Q}(N)) \setminus H &= \mathit{STOP}.
\end{aligned}$$

We construct a corresponding strategy for Low, attempting to perform l, and giving result k if l is accepted during time $[k\delta, (k+1)\delta)$ for $k = 0, \ldots, N-1$, and result N otherwise. This is the test $T(0)$ where:

$$\begin{aligned}
T(k) &= l \rightarrow \mathit{SUCCESS}(k) \overset{\delta}{\triangleright} T(k+1), \qquad \text{for } k \in \{0, \ldots, N-1\} \\
T(N) &= \mathit{SUCCESS}(N).
\end{aligned}$$

Then $\mathit{results}(P, \mathcal{Q}(k), T(0)) = \{k\}$, for $k \in \{0, \ldots, N\}$, so with this strategy we achieve an information flow of $N + 1 = \lceil 1/\delta \rceil + 1$.

It is not difficult to see that High and Low cannot pass more information than this, because Low cannot sample more quickly, and so this is indeed the information flow quantity.

If $1/\delta$ is an integer, then the same strategy works, and achieves an information flow quantity of $N+1 = 1/\delta + 1$. It might be thought that we can achieve one higher, because h can now be performed at time $N\delta = 1$. However, that strategy:

$$\begin{aligned}
\mathcal{Q}(k) &= \mathit{WAIT}\, k\delta \,;\, h \rightarrow l \rightarrow \mathit{STOP}, \qquad \text{for } k \in \{0, \ldots, N\}, \\
\mathcal{Q}(N+1) &= \mathit{STOP}, \\
T(k) &= l \rightarrow \mathit{SUCCESS}(k) \overset{\delta}{\triangleright} T(k+1), \qquad \text{for } k \in \{0, \ldots, N\}, \\
T(N+1) &= \mathit{SUCCESS}(N+1)
\end{aligned}$$

doesn't work. The reason is that when an event is withdrawn by a process—for example, when h is withdrawn at time 1 by P—it is nondeterministic whether or not the event is available at that instant; this is often referred to as a *point nondeterminism*. In the combination $(P \underset{\Sigma}{\parallel} Q(N)) \setminus H$, it is nondeterministic whether the h will be performed at time 1, or whether it will be preempted by the timeout. Hence

$$(P \underset{\Sigma}{\parallel} Q(N)) \setminus H = \mathit{WAIT}\, 1 \,;\, l \rightarrow \mathit{STOP} \sqcap \mathit{STOP},$$

and $\mathit{results}(P, Q(N), T(0)) = \{N, N+1\}$, and so the *ok* predicate is not satisfied.

One might consider the refinement R of P such that the point nondeterminism at time 1 is resolved so that the h is always available at that time. However, this R is not a Timed CSP process: the Timed Failures Model [10, 15] defines refusals to be composed of refusal tokens of the form $[t_0, t_1) \times A$ (where $t_0, t_1 \in [0, \infty)$, $A \in \mathbb{P}\,\Sigma$); however R would have a maximal refusal including $(1, 2) \times \{h\}$ but not $(1, h)$. This property is an artifact of the Timed Failures Model, chosen because it makes sense in most circumstances; however, it is not what we want here. We believe the way to overcome this would be to allow refusal tokens of the form $(t_0, t_1) \times A$. We intend to investigate the consequences of this change to the model.

However, we do not consider this a major shortcoming: in any real system, it will not be possible to estimate the minimum delay δ and the timeout 1 with total accuracy, so any values we give them will be approximations; the chances of $1/\delta$ really being an integer seem remote.

4.2 Analysis

The following lemma is analogous to Lemma 2 in the untimed model:

Lemma 6 If $P \sqsubseteq_{TF} Q$ then $\forall\, T \bullet \mathit{results}_{TF}(P, T) \supseteq \mathit{results}_{TF}(Q, T)$.

The following corollary is immediate from the previous result.

Corollary 7 If $P \equiv_{TF} Q$ then $\forall\, T \bullet \mathit{results}_{TF}(P, T) = \mathit{results}_{TF}(Q, T)$.

One might suspect, as in the untimed case, that the converse of the above result holds: that if P and Q are any two distinct processes, then given a suitably small minimum delay, we can find a test T that distinguishes them. This turns out to be not quite true. Consider the two processes

$$P_1 \mathrel{\widehat=} \textstyle\bigsqcap_{t \geqslant 0} \mathit{WAIT}\ t\ ;\ a \to \mathit{STOP}, \qquad P_2 \mathrel{\widehat=} \textstyle\bigsqcap_{t > 0} \mathit{WAIT}\ t\ ;\ a \to \mathit{STOP}.$$

P_1 and P_2 are distinct: only the former can perform a at time 0. However no test distinguishes them. The obvious form of test to consider is:

$$T \mathrel{\widehat=} a \to \mathit{SUCCESS}(1) \overset{t}{\rhd} \mathit{SUCCESS}(0),$$

for some t. However, this will only distinguish P_1 and P_2 if we take $t = 0$, which is not allowed, because of our bounded-speed restriction on tests.

The two processes above are unusual in that they involve an infinite nondeterministic choice. If we restrict ourselves to sufficiently well behaved processes, then tests can be found to distinguish any pair of processes. More precisely, we restrict ourselves to processes satisfying the following condition:

$$\forall\, s, a, t \bullet \exists\, \varepsilon > 0 \bullet \nexists\, t' \in (t, t+\varepsilon) \bullet (s \frown \langle (t', a) \rangle, [t, t') \times \{a\}) \in \mathcal{TF}[\![P]\!]. \quad (3)$$

The condition says that following trace s, there is a lower bound $t + \varepsilon$ on those times strictly after t at which a can become available. This condition is satisfied by all processes that do not use an infinite nondeterministic choice or an infinite prefix choice.

Lemma 8 If P and Q satisfy equation (3), then

$$(\forall\, \delta > 0 \bullet \forall\, T \in \mathit{Test}(\delta) \bullet \mathit{results}(P, T) = \mathit{results}(Q, T)) \;\Rightarrow\; P \equiv_{TF} Q.$$

The proof considers (without loss of generality) a failure (s, X) of P but not of Q, and constructs a test T that observes the events of s and the refusal tokens of X, and then gives result 1. Equation (3) is used to show that no other behaviour of Q can lead to the same result. The full proof can be found in [7].

In [14], Schneider proves that timed may testing is equivalent to finite timed failures equivalence; hence for processes satisfying equation (3), our testing scheme is equivalent to may testing.

By analogy with the untimed case, one might expect that if there are high level processes Q_0 and Q_1 such that $(P \underset{H}{\parallel} Q_0) \setminus H \not\equiv_{TF} (P \underset{H}{\parallel} Q_1) \setminus H$,, and both systems satisfy equation (3), then P would have information flow quantity of at least 2, given a suitable minimum delay. However, this is not the case, because of the possibility of point nondeterminism, mentioned in Section 4.1. Consider

$$P \mathrel{\widehat{=}} h \to l \to STOP \overset{0}{\triangleright} STOP, \qquad Q_0 \mathrel{\widehat{=}} STOP, \qquad Q_1 \mathrel{\widehat{=}} h \to STOP.$$

Then $(P \underset{H}{\parallel} Q_0) \setminus H \not\equiv_{TF} (P \underset{H}{\parallel} Q_1) \setminus H$, because the latter process can perform l, but the former can't. However, if we consider the corresponding strategy

$$\begin{gathered} \mathcal{Q}(0) \mathrel{\widehat{=}} STOP, \qquad \mathcal{Q}(1) \mathrel{\widehat{=}} h \to l \to STOP, \\ T \mathrel{\widehat{=}} l \to SUCCESS(1) \text{ else } SUCCESS(0) \end{gathered}$$

then this does not work: it is nondeterministic whether or not the h will be performed at time 0 or not, and so $results(P, \mathcal{Q}(1), T) = \{0, 1\}$. As above, we would like to consider the refinement of P where the h is always accepted at time 0; however this is not a Timed CSP process. The problem can again be considered unlikely: in practice the timeout is unlikely to occur at time 0, but instead at some small time $\varepsilon > 0$; in this case, the above strategy does indeed achieve an information flow of 2. We suspect that adapting the Timed Failures Model, as described above, would fix this problem.

In [5], Focardi, Gorrieri and Martinelli define a security property *Timed Bisimulation Non Deducibility on Composition (TBNDC)* (within the context of a discreet-time process algebra) which can be written using our notation as:

$$P \ \mathbf{sat}\ TBNDC \Leftrightarrow \forall\, Q \in TCSP_H \bullet P \underset{H}{\parallel} STOP \equiv_B (P \underset{H}{\parallel} Q) \setminus H,$$

where $\equiv_B$ is bisimulation equivalence. This prompts definitions for security properties *Timed Failures Non Deducibility on Composition (TFNDC)* and *Strong Timed Failures Non Deducibility on Composition (STFNDC)* as follows:

$$\begin{aligned} P \ \mathbf{sat}\ TFNDC &\Leftrightarrow \forall\, Q \in TCSP_H \bullet P \underset{H}{\parallel} STOP \equiv_F (P \underset{H}{\parallel} Q) \setminus H, \\ P \ \mathbf{sat}\ STFNDC &\Leftrightarrow \forall\, R \sqsupseteq P \bullet R \ \mathbf{sat}\ TFNDC. \end{aligned}$$

We conjecture that the above changes to the Timed Failures Model would mean that no information flow in our model would correspond to Strong Timed Failures Non Deducibility on Composition.

5 Bounded time information flow

Many processes can be used to pass an unbounded amount of information given sufficient time; however, they might still have a finite *rate* of information flow. In this section we extend the work of the previous section to consider the amount of information passed in some finite interval, and hence define the rate of information flow.

We can define the results obtainable from a system S with test T up to time t as follows:

$$\mathit{results}_t(S, T) \mathrel{\widehat{=}} \{k \mid \exists (s, X) \in \mathcal{TF}[\![S]\!] \cdot (s, X, k) \in \mathcal{TF}[\![T]\!] \wedge \mathrm{end}(s, X) \leqslant t\}.$$

Note that we only consider observations up to time t.

The following definitions are obvious adaptations of previous ones.

$$\begin{aligned}
\mathit{results}_t(P, Q, T) &\mathrel{\widehat{=}} \mathit{results}_t((P \underset{\Sigma}{\parallel} Q) \setminus H, T), \\
\mathit{ok}_t(P, \mathcal{Q}, T) &\mathrel{\widehat{=}} \forall k \in \operatorname{dom} \mathcal{Q} \cdot \mathit{results}_t(P, \mathcal{Q}(k), T) = \{k\} \\
&\qquad \text{and } \mathcal{Q}(k) \text{ satisfies (2)}, \\
\mathit{IFQ}_t(P) &\mathrel{\widehat{=}} \max\{\# \operatorname{dom} \mathcal{Q} \mid \mathcal{Q} \in \mathbb{N} \nrightarrow \mathit{TCSP} \wedge T \in \mathit{Test} \wedge R \in \mathit{TCSP} \\
&\qquad \wedge R \sqsupseteq P \wedge \mathit{ok}_t(R, \mathcal{Q}, T)\}.
\end{aligned}$$

We can define the *long term information flow rate* (in bits per time unit) as follows:

$$\mathit{LTIFR}(P) \mathrel{\widehat{=}} \lim_{t \to \infty} \frac{\log_2(\mathit{IFQ}_t(P))}{t} \qquad \text{if the limit exists.}$$

Of course, the above definition doesn't give the whole story. A process that leaked a gigabyte of information at time 0, but nothing subsequently, would have a long term information flow rate of zero, but would probably not be acceptable as a secure system.

5.1 Examples

As a first example, consider $P \mathrel{\widehat{=}} h \to l \to \mathit{STOP}$. Then, by reasoning as in Section 4.1, we can show that $\mathit{IFQ}_t(P) = \lfloor t/\delta \rfloor + 1$, for all t. So the long term information flow rate is zero.

As another example, consider a processor, shared by High and Low, scheduled using a round robin scheduler with time quantum q. Scheduling consists of two phases: an execution phase, where one agent has control of the processor; and a contention phase, where it is decided who should have the processor next.

Let PROC_L and PROC_H denote the behaviour of the processor when scheduled for Low and High, respectively. During the execution phase, the processor acts like either PROC_H or PROC_L for a period of q, before being interrupted and moving to the contention phase:

$$\begin{aligned}
\mathit{EXEC}_H &\mathrel{\widehat{=}} \mathit{PROC}_H \triangle_q \mathit{CONTENTION}_L, \\
\mathit{EXEC}_L &\mathrel{\widehat{=}} \mathit{PROC}_L \triangle_q \mathit{CONTENTION}_H.
\end{aligned}$$

During the contention phase, High and Low can request the processor via events req_h and req_l, respectively. The scheduler will repeatedly offer one *req* event, but if it is not accepted within time w, the offer is withdrawn and the other *req* event offered. After an execution phase, the processor is first offered to the other agent.

$$\begin{aligned} CONTENTION_H &\mathrel{\widehat{=}} req_h \to EXEC_H \overset{w}{\triangleright} CONTENTION_L, \\ CONTENTION_L &\mathrel{\widehat{=}} req_l \to EXEC_L \overset{w}{\triangleright} CONTENTION_H. \end{aligned}$$

Initially, High has priority:

$$SCHEDULER \mathrel{\widehat{=}} CONTENTION_H.$$

We assume that δ is less than w and q.

It turns out that High and Low will be able to pass one bit per $q + w$ time units. Fix t, and let $n \mathrel{\widehat{=}} \lfloor \frac{t}{q+w} \rfloor$ (this will be the number of bits passable in time t). The strategy will involve High using the processor to pass a bit 1, and not using the processor to pass a bit 0. We pass the bits one at a time, least significant bit first. High's strategy to pass the value k can be modelled by the process $Q(k) \mathbin{|||} RUN(L)^3$ where:

$$\begin{aligned} Q(k) \mathrel{\widehat{=}} {} & \text{if } k \bmod 2 = 1 \text{ then } req_h \to WAIT(q + w) \,;\, Q(k \text{ div } 2) \\ & \text{else } WAIT(q + w) \,;\, Q(k \text{ div } 2). \end{aligned}$$

Low's strategy is to attempt req_l events at times of the form $w + (q + w)i$, and is modelled by the test $WAIT\ w \,;\, T(0, 0)$, below. The test accumulates the result in the parameter r; the parameter i records the number of bits received so far.

$$\begin{aligned} T(i, r) &\mathrel{\widehat{=}} (req_l \to WAIT(q + w) \,;\, T(i + 1, r)) \overset{w}{\triangleright} (WAIT(q) \,;\, T(i + 1, 2^i + r)) \\ &\qquad \text{for } i = 0, \ldots, n - 1, \\ T(n, r) &\mathrel{\widehat{=}} SUCCESS(r). \end{aligned}$$

Note that High will succeed in performing req_h at times that are multiples of $q + w$. Low will perform req_l at times of the form $w + (q + w)i$ if and only if High did not perform req_h at the corresponding time $(q + w)i$.

This strategy achieves an information flow quantity of 2^n where $n = \lfloor \frac{t}{q+w} \rfloor$, and so the long term information flow rate is $\frac{1}{q+w}$.

6 Conclusions

In this paper we have presented a formal definition for the quantity of information passed from a high level user to a low level user in a system, based upon the number of behaviours of High that can be accurately distinguished from Low's point of view.

[3] Strictly speaking, $RUN(L)$ is not a Timed CSP process, because it does not satisfy the finite speed axiom of the Timed Failures Model; we take $RUN(L)$ to be a process that will perform as many events from L as its environment requires.

The correct definition proved remarkably difficult to find: many alternative definitions failed to give the correct values for example processes. The aspect that proved hardest to get right was that High might be able to prevent events of Low occurring by preempting them with his own events.

The untimed model gives the correct values for all the examples considered. It agrees with a more intuitive definition on those processes that exhibit no flow of information. The timed model runs into problems with point nondeterminism if High attempts events at the precise point at which they are withdrawn, but this is an artifact of the Timed Failures Model, which we believe can be overcome by adapting the model.

In this section we discuss a slight alternative to the untimed model, and then suggest some directions for future work.

6.1 On refusal testing

The untimed model of information flow in Section 3 was built around the Failures Model of CSP. The intuition in this model is that Low is unable to perform any events after observing a refusal. We now consider briefly how we could relax this assumption.

We extend our language of tests to allow Low to continue to offer events after observing a refusal:

$$T \in \mathit{Test} \ ::= \ \mathit{SUCCESS}(k) \mid (\Box_{a \in A}\, a \to T_a) \ \mathsf{else}\ T'.$$

The test $(\Box_{a \in A}\, a \to T_a)\ \mathsf{else}\ T'$ offers the events from A; if one is performed, it then acts like the corresponding T_a; if all are refused, then it acts like T'.

In [8], Mukkaram introduced the Refusal Testing Model of CSP, influenced by similar work by Phillips [9]. The Refusal Testing Model models a process by a set of refusal traces, where a refusal trace is an alternating sequence of events and refusals, where a refusal is either a set of events—representing that those events were refused—or a special value "$\bullet$"—representing lack of refusal information.

We can associate a set of (refusal trace, result) pairs with tests of the above form:

$$\begin{aligned}
\mathcal{R}[\![\mathit{SUCCESS}(k)]\!] \ &\mathrel{\widehat{=}}\ \{(\langle \bullet \rangle, k)\}, \\
\mathcal{R}[\![(\Box_{a \in A}\, a \to T_a)\ \mathsf{else}\ T']\!] \ &\mathrel{\widehat{=}}\ \{(\langle \bullet, a \rangle ^\frown s, k) \mid a \in A \wedge (s, k) \in \mathcal{R}[\![T_a]\!]\} \\
&\qquad \cup \{(\langle R \sqcup A \rangle ^\frown s', k) \mid (\langle R \rangle ^\frown s', k) \in \mathcal{R}[\![T']\!]\},
\end{aligned}$$

where the $\sqcup$ operator forms the upper bound of two pieces of refusal information; if X and Y range over sets of events, then

$$\bullet \sqcup \bullet = \bullet, \qquad \bullet \sqcup X = X \sqcup \bullet = X, \qquad X \sqcup Y = X \cup Y.$$

We will write $\mathit{results}_R(S, T)$ for the set of results obtainable from refusal-testing-based test T with system S:

$$\mathit{results}_R(S, T) \ \mathrel{\widehat{=}}\ \{k \mid \exists\, s \in \mathcal{R}[\![S]\!] \cdot (s, k) \in \mathcal{R}[\![T]\!]\}.$$

We can then use this information to define the information flow quantity, as in the previous models.

For example, recall the following process from Example 4:

$$P_4 \;\hat{=}\; h1 \to l1 \to STOP \;\Box\; h2 \to (l1 \to STOP \;\Box\; l2 \to STOP).$$

This process had an information flow quantity of 2 in the Failures Model. However, consider the following strategy in the Refusal Testing Model:

$$\mathcal{Q}(0) = RUN(L), \qquad \mathcal{Q}(1) \;\hat{=}\; h1 \to RUN(L), \qquad \mathcal{Q}(2) \;\hat{=}\; h2 \to RUN(L),$$
$$T \;\hat{=}\; l2 \to SUCCESS(2) \text{ else } l1 \to SUCCESS(1) \text{ else } SUCCESS(0).$$

This strategy gives an information flow quantity of 3. Note in particular that $results_R(P, \mathcal{Q}(1), T) = \{1\}$, corresponding to the refusal trace $\langle \{l2\}, l1, \bullet \rangle$.

It is interesting that the choice between the Failures and Refusal Testing Models seems to make more of a difference to our results than it normally does in related areas.

6.2 Future work

As discussed above, we intend to investigate the consequences of adapting the Timed Failures Model to allow refusal tokens of the form $(t, t') \times A$; we need to ensure that the model, with this change, is still closed under the operators of the language. We will then attempt to prove the above conjecture, that lack of information flow in our model corresponds to Strong Timed Failures Non Deducibility on Composition.

It would clearly be useful to have an automated procedure for calculating the quantity of information flow in a system. We believe there is some scope for considering all possible High processes that can successfully interact with the system (i.e. only those High processes that attempt events the system might actually offer), and all corresponding Low tests (i.e. only those tests that attempt events the system might actually offer), and identifying which subsets satisfy the *ok* predicate. Making this efficient might prove challenging.

It would also be interesting to ask what algebraic laws are satisfied by our definition. For example, we conjecture that the following laws hold:

$$\begin{aligned} IFQ(P1 \sqcap P2) &= \max(IFQ(P1), IFQ(P2)), \\ IFQ(P1 \;|||\; P2) &= IFQ(P1) + IFQ(P2). \end{aligned}$$

Acknowledgements

I would like to thank Bill Roscoe, Steve Schneider and Philippa Broadfoot for useful discussions about information flow. This work was partially supported by a grant from the US Office of Naval Research. It was started while I was employed by the University of Leicester.

References

[1] P. G. Allen. A comparison of non-interference and non-deducibility using CSP. In *Proceedings of the 4th IEEE Computer Security Foundations Workshop*, 1991.

[2] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[3] Riccardo Focardi. Comparing two information flow security properties. In *Proceedings of 9th IEEE Computer Security Foundations Workshop*, pages 116–122, 1996.

[4] Riccardo Focardi and Roberto Gorrieri. A classification of security properties. *Journal of Computer Security*, 1995.

[5] Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. Information flow analysis in a discrete-time process algebra. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 170–184, 2000.

[6] Richard Forster. *Non-Interference Properties for Nondeterministic Processes.* D.Phil, Oxford University, 1999. Available from `http://www.comlab.ox.ac.uk/oucl/research/areas/concurrency/papers/thesis.ps.gz`.

[7] Gavin Lowe. Quantifying information flow. Technical report, Oxford University Computing Laboratory, 2001.

[8] Abida Mukkaram. *A Refusal Testing Model for CSP.* D.Phil, Oxford University, 1993.

[9] Iain Phillips. Refusal testing. *Theoretical Computer Science*, 1987.

[10] G. M. Reed and A. W. Roscoe. The timed failures-stability model for CSP. *Theoretical Computer Science*, pages 85–127, 1999.

[11] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of 1995 IEEE Symposium on Security and Privacy*, 1995.

[12] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

[13] P. Y. A. Ryan. A CSP formulation of non-interference. *Cipher*, pages 19–27, 1991. Also in Proceedings of the 3rd IEEE Computer Security Foundations Workshop, 1990.

[14] Steve Schneider. An operational semantics for Timed CSP. *Information and Computing*, 116(2):193–213, 1995.

[15] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Wiley, 1999.

[16] US Department of Defense. *DoD Trusted Computer System Evaluation Criteria*, 1985. DOD 5200.28-STD.

A A brief overview of CSP

A.1 Untimed syntax

In this section we give a brief overview of CSP. More details can be obtained from [12, 15].

An event represents an atomic communication; this might either be between two processes or between a process and the environment. Channels carry sets of events; for example, $l.5$ is an event of channel l. Σ represents the set of all events.

In this paper we use processes defined using the following syntax:

$STOP$	process that can perform no events.
$a \rightarrow P$	process that can perform the event a, and then act like P.
$P \mathrel{\Box} Q$	external choice; the process can act like either P or Q; the choice is made by the environment.
$\Box_{i \in I} P_i$	external choice between the processes P_i, where i ranges over I.
$P \sqcap Q$	internal or nondeterministic choice; the process can nondeterministically choose to act like either P or Q.
$\bigsqcap_{i \in I} P_i$	nondeterministic choice between the processes P_i, where i ranges over I.
$P \setminus A$	process that acts like P, except all events from the set A are hidden, i.e. made internal.
$RUN(A)$	process that can perform any sequence of events from A, and never refuse any events from A.
$CHAOS(A)$	the most nondeterministic, nondivergent process with alphabet A; the process can perform any sequence of events from A, and refuse any events.
$P \underset{A}{\parallel} Q$	parallel composition of P and Q, synchronising on events from A.
$P \mathbin{\vert\vert\vert} Q$	interleaving of the processes P and Q; i.e. parallel composition with no synchronisation.

A.2 Untimed semantics

A *trace* is a sequence of events performed by a process. A *failure* is a pair (s, X), where s is a trace and X is a set of events known as a *refusal*. This represents that the process can perform the trace s, and then reach a stable state (i.e. from where no internal activity is possible) from which it can perform none of the events in X.

The Failures Model of CSP represents a process by the sets of its traces and failures. We write $\mathcal{T}[\![P]\!]$ for the traces of P, and $\mathcal{F}[\![P]\!]$ for the failures of P. We say that process P is failures equivalent to process Q if they have the same traces and failures; we say that P is refined by Q if the traces and failures of P are supersets of those of Q:

$$\begin{aligned} P \equiv_F Q &\Leftrightarrow \mathcal{T}[\![P]\!] = \mathcal{T}[\![P]\!] \wedge \mathcal{F}[\![P]\!] = \mathcal{F}[\![Q]\!], \\ P \sqsubseteq_F Q &\Leftrightarrow \mathcal{T}[\![P]\!] \supseteq \mathcal{T}[\![P]\!] \wedge \mathcal{F}[\![P]\!] \supseteq \mathcal{F}[\![Q]\!]. \end{aligned}$$

For non-divergent processes, the traces are uniquely defined by the failures:

$$\mathcal{T}[\![P]\!] \;=\; \{s \mid \exists\, X \bullet (s, X) \in \mathcal{F}[\![P]\!]\},$$

and so it is enough to consider just the failures.

A.3 Timed syntax

The syntax of Timed CSP is that of untimed CSP, extended as follows:

$a \xrightarrow{t} P$	process that can perform the event a at any time, and then act like P after a delay of t.
$a \to P$	shorthand for $a \xrightarrow{0} P$.
$WAIT\ t\ ; P$	process that acts like P after a delay of t.
$P \overset{t}{\triangleright} Q$	process that acts like P, except if no event occurs within t time units, the process times-out and acts like Q.
$P \triangle_t Q$	process that acts like P for t time units, after which P is interrupted and the process then acts like Q.

A.4 Timed semantics

A *timed event* is a pair (t, a), representing the occurrence of event a at time t. A *timed trace* is a sequence of timed events in non-decreasing order of times. A *refusal token* is a set of timed events of the form $[t, t') \times A$. A *timed refusal* is a set of timed events formed as the union of a finite number of refusal tokens. A *timed failure* is a pair (s, X), where s is a timed trace and X is a timed refusal, representing the performance of the events in s, and the refusal to perform the events in X.

The Timed Failures Model represents a process P by the set of timed failures it can exhibit, written as $\mathcal{TF}[\![P]\!]$. Equivalence and refinement are defined analogously to as in the untimed model:

$$P \equiv_{TF} Q \Leftrightarrow \mathcal{TF}[\![P]\!] = \mathcal{TF}[\![Q]\!],$$
$$P \sqsubseteq_{TF} Q \Leftrightarrow \mathcal{TF}[\![P]\!] \supseteq \mathcal{TF}[\![Q]\!].$$

We use the following notation on failures in this paper: $X \upharpoonright A$ represents refusal X restricted to events from A; $\mathrm{end}(s, X)$ represents the supremum of times of events in s and X; $s + t$ and $X + t$ represent s and X temporally shifted, by adding t onto all times.

Can verification be approximated ?

Sylvain Peyronnet

syp@lri.fr
University Paris-XI
LRI Bâtiment 490,
F-91405 Orsay Cedex, France

1 Introduction

Model Checking [CGL94, McM93] is an algorithmic method to verify whether a system satisfies a specification written in a logic-based language. In *Symbolic Model Checking* , OBDD's are often used to represent the binary accessibility relation, and a given formula ψ by combining OBDD's for the explicit relations of the formula. A central problem is that some relations as the multiplication admit OBDD's of exponential size only [Bry91].

For probabilistic systems, general methods have been presented in [HJ94, CY95] where the verification of a probabilistic statement expressed in PCTL is reduced to the solution of a linear problem of size proportional to the size of S, the state space of the system. MTBDD's (variations of OBDD's) have been used in [HGCC99, dAKN$^+$], Kronecker representations in [dAKN$^+$] in order to generalize the classical model-checking.

In the cases where classical verification is computational intractable, we consider feasible possible approximations. We trade exact verification for approximate verification but gain an efficient procedure. We give examples in the case of deterministic systems and a detailed implementation for probabilistic ones.

In section 2, we define the notion of approximation for the verification. In section 3 we give ideas about the use of approximation in model checking of deterministic systems. The section 4 deals with approximation for probabilistic system, we give an algorithm and show that the method is efficient on the dining philosophers problem.

2 Approximation

Our aim is to approximately verify wether $\mathcal{M} \models \phi$, where $\mathcal{M} = (S, R, U_1, ..., U_n)$ is a Kripke structure (S is the state space, R is the transition relation and $U_1, ..., Un$ are unary relations) implicitely given by a program.

2.1 Randomized algorithms

For many intractable problems there exist randomized algorithms wich give the right answer with high probability (1-ϵ with $\epsilon = 10^{-10}$ for example). An algorithm $\mathcal{A}$ in the BPP complexity class (BPP stands for Bounded Probability error and Polynomial time) would have to satisfy:

- $\mathcal{M} \models \phi \Rightarrow Prob[\mathcal{A}\ accept] \geq 1 - \epsilon$
- $\mathcal{M} \not\models \phi \Rightarrow Prob[\mathcal{A}\ accept] \leq \epsilon$

2.2 Optimization problems

An optimization problem is a problem that consists in maximizing a cost function associated to the set of solutions to the problem. One can construct efficient heuristic to solve NP-complete optimization problem that are intractable otherwise and say that the algorithm gives an approximation for the problem. For the SAT problem, we look at the MAXSAT optimization problem : Given n clauses $\{c_1, ..., c_n\}$, we compute the maximum number of clauses that can be satisfied by a valuation.

There exist a (3/4)-approximation algorithm for MAXSAT, i.e an algorithm which gives an approximate value that is at least 3/4 of the true value and it provides a heuristic for SAT using the approximation of MAXSAT. This reduction of SAT to MAXSAT will be useful for section 3.

2.3 Property Testing

The question in property testing is : let ϕ be a property and $\mathcal{M}$ a system, we want to decide if $\mathcal{M}$ is far from any system which has the property ϕ ? A property testing algorithm is an algorithm that satisfy the two following properties :

- $\mathcal{M} \models \phi \quad \Rightarrow \quad Prob(\mathcal{M}'_i \models \psi) = 1$
- $\mathcal{M} \not\models_\epsilon \phi \quad \Rightarrow \quad Prob(\mathcal{M}'_i \not\models \psi) > 1 - \delta$

ϵ is the distance parameter of the test and δ is the confidence parameter.
$\mathcal{M} \not\models_\epsilon \phi$ stands for ϕ is far to be true in $\mathcal{M}$, formally :

- $\forall \mathcal{M}',\ dist(\mathcal{M}, \mathcal{M}') < \epsilon \ \Rightarrow\ \mathcal{M}' \not\models \phi$

$dist$ is a distance on structures (for example, for two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, $dist(G_1, G_2) = \infty\ if\ V_1 \neq V_2$ and $dist(G_1, G_2) =\mid E_1 \Delta E_2 \mid /n^2$ otherwise). For many intractable properties, there are efficient testers. A classical example is the test of a graph property, like k-colorability. [GGR96].
The test is quite simple : sample $\mathcal{O}(\frac{k^2.\log(k/\delta)}{\epsilon^3})$ vertices of the graph and check if the induced subgraph is k-colorable using a good average case algorithm. The complexity of the test is independent of the size of the graph. In [GGR96], it is shown that this procedure is a property testing algorithm.

3 Application to deterministic systems

Several approaches can be considered.

3.1 Approximation and BDDs

In [MB80], it is shown that we can decide the equality of BDDs in randomized polynomial time with a BPP algorithm. One could conceive similar BPP algorithms for the manipulation of BDDs (generalization of classicals OBDDs). Such algorithms would lead to a randomized algorithm for symbolic model checking.

3.2 Approximation and Bounded Model Checking

Bounded Model Checking (BMC) is an approach for symbolic model checking due to Biere and al. [BCCZ99]. The idea is to consider paths of depth k of Kripke tree and to generate a SAT instance that is satisfiable iff there exists a counterexample. Consider a system $\mathcal{M}$ and a formula ϕ to be verified : we construct a set of clauses that depend on $\mathcal{M}$ and $\neg\phi$. If we find a valuation that satisfies this set of clauses, we can ensure that the system is incorrect. The valuation gives us a counterexample to the property. In order to construct a finite set of clauses, we only consider paths of depth k. The clauses of an instance SAT with variables $s_0, ...s_k$ are grouped in 2 components such that :

- there exists a path σ of depth k (in the Kripke structure) iff there exists a satisfying assignment of the first component.
- the formula $\neg\phi$ is satisfied on σ iff the second component is satisfied.

We know that we can 3/4-approximate the MAXSAT optimization problem which is associated to SAT.

Input : $\mathcal{M}$, ϕ

1. *Generate the SAT instance $\mathcal{C}$ for $\neg\phi$ and $\mathcal{M}$ (following the BMC approach)*
2. *Compute the approximation of MAXSAT for $\mathcal{C}$.*
3. *If MAXSAT=$|C|$ then we conclude that the system is incorrect.*
If MAXSAT is $<|\ \mathcal{C}\ |\ .3/4$ then with high probability ϕ is not true.
For other values of MAXSAT, we can't conclude.

For example, assume that we have a set $\mathcal{C}$ with 1000 clauses. If we have MAXSAT$_{approx}$=1000, we conclude that the formula is not valid in the model. If we have MAXSAT$_{approx}$=50, we conclude with high probability that the formula is valid.

4 Application to probabilistic systems

We show how to develop approximate algorithms in the context of probabilistic systems.

4.1 Probabilistic systems

We first present the classical logical framework which allows to express quantitative properties of probabilistic processes and to check these properties.

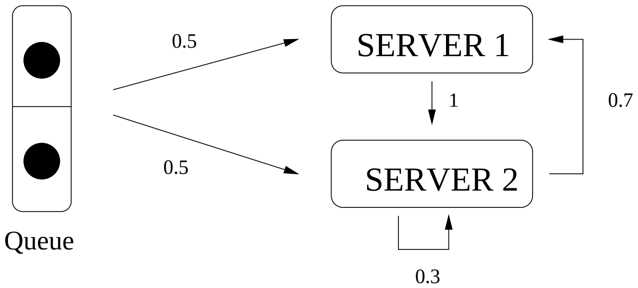
Definition 1 *A probabilistic transition system is a structure* $\mathcal{M} = (S, P, U_1, \ldots, U_m)$ *where:*

- S *is a set of states,*
- $P : S \times S \to [0, 1]$ *is a transition probability function, such that for all* $s \in S$, $\sum_{t \in S} P(s, t) = 1$,
- *for each* $i = 1, 2, \ldots, m$, U_i *is a unary relation on* S.

If S is finite or denumerable, we can consider P as a transition matrix. For each structure M and state s, it is possible to define a probability measure on the set of paths from s:

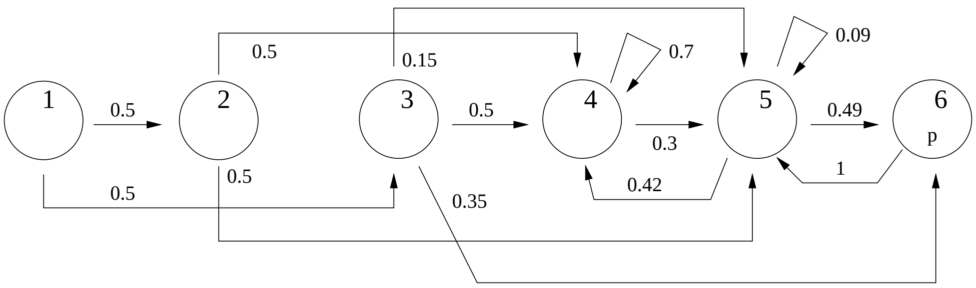
4.1.1 Queue-server example

Let's consider the following simple probabilistic system with a queue and two servers that can accomodate an arbitrary number of processes. Initially the queue contains two processes and at each stage a process is sent to one of the server with probability 1/2. We want to verify that the probability of the two processes to be together in the server 1 is greater than 0.99.



Succinct representation of the queue-server protocol

The succinct representation is the one above. The size of this representation is constant even if the queue contains a huge number of processes. If we want the classical representation, we must build the state space and the transitions : the states of such a system are described by vectors (a, b, c) where a is the number of processes in the queue, b is the number of processes in the server 1 and c is the number of processes in the server 2. For a queue of length 2 the initial state is $(2, 0, 0)$.



Transition of the queue-server.

For simplicity we code the states by an integer given by the following table :

1	2	3	4	5	6
(2,0,0)	(1,1,0)	(1,0,1)	(0,1,1)	(0,0,2)	(0,2,0)

The probabilistic space is the set of infinite paths of the Kripke structure.

4.2 Probabilistic CTL

The logic PCTL was introduced by Hansson and Jonsson [HJ94]. Formulas are built from atomic formulas, logical connectives, temporal connectives for expressing time (next-step X and until U) and a probabilistic operator.

Definition 2 *The set of PCTL formulas is defined inductively as follows:*

- *each atomic formula is a PCTL formula,*
- *if ϕ and ψ are PCTL formulas, then so are $\neg\phi$ and $(\phi \wedge \psi)$.*
- *if ϕ and ψ are PCTL formulas and p is a real number $\in [0,1]$, then $\mathbb{P}rob_{\succeq p}(X\phi)$ and $\mathbb{P}rob_{\succeq p}(\phi U \psi)$ are PCTL formulas (where $\succeq$ is $\geq$ or $>$).*

PCTL formulas are interpreted over the states of a probabilistic transition system, whereas path formulas are interpreted over paths. Let $\mathcal{M} = (S, P, U_1, \ldots, U_k)$ a probabilistic transition system, $s \in S$ a state and σ a path. The satisfaction relation is defined by induction. The cases of atomic formulas, boolean connectives and path formulas are usual. We consider the case of the probabilistic operator:

- $(\mathcal{M}, s) \models \mathbb{P}rob_{\succeq p}(X\phi)$ iff $Prob(\{\sigma \in Path(s) \ : \ (\mathcal{M}, \sigma) \models X\phi\}) \succeq p$,
- $(\mathcal{M}, s) \models \mathbb{P}rob_{\succeq p}(\phi U \psi)$ iff $Prob(\{\sigma \in Path(s) \ : \ (\mathcal{M}, \sigma) \models [\phi U \psi]) \succeq p$

For a path formula H, the set $\{\sigma \in Path(s) \ : \ (\mathcal{M}, \sigma) \models H\}$ is measurable and $Prob(\{\sigma \in Path(s) \ : \ (\mathcal{M}, \sigma) \models H\}$ is well defined.

To verify such formulas [HJ94] describe a technique which given a structure $\mathcal{M}$, an initial state s and a PCTL formula ϕ determines whether $(\mathcal{M}, s) \models \phi$. The method associates a linear system of size $O(\mid S \mid)$ whose solution determines the answer. The complexity of the model checking problem is $O(\mid S \mid^3 . \mid \phi \mid)$. C.Courcoubetis and M.Yannakakis give another algorithm [CY95] whose complexity is linear in the size of the state space and exponential in the size of the formula. We clearly need methods polynomial in the size of the succinct representation and not the size of $\mathcal{M}$.

Example : Let T be the transition matrix of the queue-server system. In order to verify the formula $\mathbb{P}rob\ [TrueUp]_{\geq 0.5}$, we solve a linear system which includes the restriction of the transition matrix to the set S' of states wich satisfy $E(TrueUp)$ and not p. We are led to the system $x = T_{S'} * x + (T * y)_{S'}$ where y is a vector for the set S_0 of states where p is true, $S' = \{x_1, ..., x_5\}$, $S_0 = \{x_6\}$ and :

$$T = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0.5 & 0.15 & 0.35 \\ 0 & 0 & 0 & 0.7 & 0.3 & 0 \\ 0 & 0 & 0 & 0.42 & 0.09 & 0.49 \end{pmatrix}$$

When we solve the system, we find a vector x where x_i represents the probability of $[TrueUp]$ to be true in our model at state i.

4.3 The random sampling method

Let us assume that the probabilistic system is represented by a complete binary trees. At depth k, we have exactly 2^k paths and each one is realized with probability $1/2^k$. It is always possible to transform the system by adding a few extra states in order to have uniformly distributed execution paths. We try to prove that $I\!Prob\ \psi \geq b$ by considering $I\!Prob_k\ \psi \geq b$, the probability defined on the probabilistic space limited to the Kripke paths of depth k. We then compare the #SAT version of the problem, i.e. the counting version of the problem with $I\!Prob_k\ \psi$. The number of satisfying valuations is directly related to $I\!Prob_k\ \psi$. The difficulty is to associate a polynomial set of clauses to the formula ψ. This is not possible for an arbitrary LTL formulas as the evaluation problem is PSPACE complete [SC85]. We will nevertheless consider a fragment where the evaluation problem is P.

It is a classical observation that the satisfaction in the Kripke model and the satisfaction on a path of depth k are incomparable. But for many natural formulas the truth at depth k implies the truth in the global model and we can infer similar connections for the probabilities. We show that we can approximate with a simple randomized algorithm the probabilities in certain cases. We will generate random paths with a uniform distribution in the same probabilistic space (underlined by the Kripke structure of depth k) and compute a random variable A (the proportion of paths satisfying the formula ψ) which approximates $I\!Prob_k\ [\psi]$.

If we need to verify a statement $I\!Prob_k\ (\psi) > b$ we will decide with $b(1-\epsilon) < A$. Our decision will be correct with probability $1-\delta$ after a polynomial number of samples in $k, \frac{1}{\epsilon}, \log\frac{1}{\delta}$ directly obtained from the Chernoff bound. We wont be able to approximate $I\!Prob_k\ (\psi) \leq b$ but in this case we will consider $I\!Prob_k\ (\neg\psi) = 1 - I\!Prob_k\ (\psi)$ and check that $I\!Prob_k\ (\neg\psi) > 1-b$. The essential advantage of the method is that we can proceed just with the **symbolic representation** of the program, i.e. the diagram, without constructing the global model.

Notice the two parameters ϵ and δ play a different role : ϵ is the approximation ration and δ is the confidence ratio. In a practical situation where we want to verify that $I\!Prob_k\ \phi > 0.99$, we would take $\epsilon = 0.01$ and $\delta = 10^{-10}$: after $O(k.\frac{1}{\epsilon^2}.\log\frac{1}{\delta})$ paths, we would get an answer.

Consider the random sampling algorithm which modifies the BMC approach as follows:

- the generation of SAT clauses which define a path σ is replaced by the random sampling of a path (assuming a generator for arbitrary probabilities $p, 1-p$).

- we maintain the same SAT clauses for the verification of ψ on a path σ.

Notice that this algorithm is exact for the monotonic fragment of PCTL (ϕ is in the monotonic fragment if : ϕ is true for a path σ of depth k imply ϕ is true for all paths which extend σ).

$\mathcal{A}$. Input : Diagram, $\phi, b, \epsilon, \delta$

1. *Generate the SAT instance C for ϕ (following the BMC approach)*
2. *For $i < N = 4\log(\frac{2}{\delta})/b\epsilon^2$, generate a random path σ {If C is true on σ, A=A+1 }*
3. *If $b.(1-\epsilon) < (A/N)$, Accept else Reject*

Example : Here are some results with this method for the formula $I\!\!Prob_k\ (TrueUp) >$ 0.99 in the queue-server example :

k	ϵ	δ	$I\!\!Prob_k\ (TrueUp)$	$\mathcal{M} \models I\!\!Prob\ (TrueUt)_{>0.99}$
10	10^{-2}	10^{-10}	0.72	no
20	10^{-2}	10^{-10}	0.94	no
30	10^{-2}	10^{-10}	0.98	no
40	10^{-2}	10^{-10}	0.99	yes

With these results, we conclude that the formula is true in our model with probability $1 - 10^{-10}$.

4.4 Experimental Results

We have experimented our method on the probabilistic version of the randomized dining philosophers [PZ86]. This example have been originally modeled using nondetermistics and probabilistics choices, but we replace nondeterministic choices by uniform probabilities. We check the well-known property of liveness :

"If a philosopher is hungry then eventually some philosopher eats"

Actually, we check that the probability of the previous formula is greater or equal to 1. The following table show our results for the model with n philosophers, we have chosen $\epsilon = 10^{-2}$ and $\delta = 10^{-10}$:

number of philosophers	depth needed	time (s)
3	20	13.95
5	25	26.94
10	35	81.88
15	45	166.55
20	55	292.32
25	60	488.14
30	70	741.00
50	100	2588.15
100	150	16960.33

We have implemented the method specially for this example using the Ocaml language on a pentium III 500 MHZ with 128 mo of RAM.
The main point is that we never construct the entire model and so the verification uses few memory. The first experimental results are promising : we are able to verify the property of liveness with a hundred philosophers in less than five hours.

5 Concluding remarks and further directions

There are many notions of approximation and we developed a system which approximately verifies that some monotonic formulas are verified by probabilistic systems.

We intend to study the other notions of approximations in future work, in the case when the classical verification is computationally infeasible.

References

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdd's. *TACAS*, 1999.

[Bry91] R.E. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, 1991.

[CGL94] E.M. Clarke, O. Grumberg, and D.A. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *JACM*, 42(4):857–907, 1995.

[dAKN+] L. de Alfaro, M. Kwiatkowska, G. Norman, D. Parker, and R. Segala. Symbolic model checking of concurrent probabilistic processes using mtbdds and the kronecker representation. In *TACAS 2000*, LNCS 1785. Springer Verlag.

[GGR96] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. In *IEEE Symposium on Foundations of Computer Science*, pages 339–348, 1996.

[HGCC99] V. Hartonas-Garmhausen, S. Campos, and E. Clarke. Probverus: Probabilistic symbolic model checking. In *5th International AMAST Workshop, ARTS'99, Bamberg, Germany, May 1999.*, LNCS 1601. Springer Verlag, 1999.

[HJ94] H.Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

[MB80] M. N. Wegman M. Blum, A. K. Chandra. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Information Processing Letters*, 10:80–82, 1980.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[PZ86] A. Pnueli and L. Zuck. Verification of multiprocess probabilistic protocols. *Distribued Computing*, pages 1:53–72, 1986.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

Implementing the modal logic FULL using Ergo

[Extended Abstract]

Peter Robinson
Software Verification Research Centre
Australia
pjr@csee.uq.edu.au

Carron Shankland
University of Stirling
United Kingdom
ces@cs.stir.ac.uk

1 Introduction

Although formal methods are becoming more popular, their uptake is at least partially governed by the success of associated tools. Theorem provers give the most flexibility, but are often difficult to use, while model checking tools are easy to use, but seldom deal effectively with large and infinite systems. The solution has often been to ignore such systems, but that is no longer a tenable position. Infinite state systems arise regularly and therefore must be dealt with satisfactorily.

We present a prototype model checker for the modal logic FULL [1]. Two strategies are used to deal with infinite systems. Firstly, the logic is defined over *Symbolic Transition Systems* (STSs), which remove infinite branching. Secondly, the model checker is implemented using the theorem prover Ergo [9], which can reason about those symbolic values.

2 LOTOS, Symbolic Transitions and FULL

LOTOS [5] is a popular process description language that has been in use for well over a decade in a number of domains, including protocols and services [7], and distributed systems [6]. The success of these applications has been amplified by the use of a number of mature verification tools.

LOTOS includes a rich set of operators for describing both process control *and* data. However, much of the foundational work on LOTOS, and subsequently the verification tools, has ignored all, or parts, of the data aspect of the language because of the problem of infinite state. In particular, the use of data introduces infinite branching into the underlying state transition systems. For example, the simple process `g?x:Nat; exit` results in an infinite choice, one for each member of `Nat`. This presents a serious obstacle to reasoning, particularly to approaches based on (finite) model-checking.

The focus of our work has been to provide a complete approach to data including semantics, abstract properties expressed using logic, equivalence relations, and tools to support reasoning about behaviours. At the heart of this framework is a *symbolic* treatment of data. A new *finitely branching* semantics of LOTOS was previously given in terms of symbolic state transition systems (STSs) [3].

Symbolic Transition Systems are essentially transition systems whose transitions can have free variables in the data label and are additionally labelled with a *transition condition* representing the conditions under which that transition is available. This approach was first introduced in [4] which gave a symbolic semantics for value passing CCS.

Basically, an STS is a directed graph whose nodes are tagged with sets of free variables, and whose branches are labelled with a boolean condition and an event. A set of rules presented in [3] define how a symbolic transition system may be constructed from a LOTOS process behaviours. The resulting transition system is typically a cyclic graph (if recursive processes are involved) and is always of finite width (since only a finite number of branches may be described in a LOTOS behaviour).

In order to describe abstract properties of symbolic transition systems a modal logic called Full LOTOS Logic (FULL) was developed. Earlier work [1, 2] describes the logic in more detail; here we give an overview.

FULL is based on the classic modal logic for concurrent systems known as Hennessy Milner Logic (HML). We use the version as presented in [8]. In addition to the usual operators of $\langle\ \rangle$ and $[\ ]$, which can be thought of as existential and universal quantification over transitions, we add quantifiers over data which may be communicated by those transitions. We refer to these as the quantified modal operators. For example, if a process P satisfies $\langle \exists x\ g\rangle\Phi$ then there is a transition labelled by gv such that Φ with v substituted for x holds with respect to the new state. The value v may be a specific value in the transition system, or a symbolic one. Further, the formula $[\exists x\ g]\Phi$ expresses that for some value v, every transition that can perform a gv action must lead to a state in which Φ with v substituted for x holds. The other operators combine the universal quantifier over data with the modal operators: $\langle \forall x\ g\rangle$ and $[\forall x\ g]$.

3 ERGO

Ergo 5 [9] is an interactive proof tool, designed and implemented at the SVRC. Ergo 5 has many useful features, but the ones most relevant here are:

- A generalised sequent calculus proof style.
- User-definable heuristics (tactics) for automation of proof steps.
- Many (50+) supplied tactics that encode common proof patterns including support for transformational reasoning (window inference).
- An Emacs graphical user interface for proof browsing and construction.
- Support for storing, browsing and replaying proofs.

Ergo 5 is implemented in Qu-Prolog, which is a high-level language designed primarily for rapid prototyping of interactive theorem provers. The object-level terms of Qu-Prolog include quantified terms, substitutions and object variables. Qu-Prolog can also be used for writing user tactics and prover extensions.

4 Implementing STS and FULL in Ergo

We give a few examples of the representation of the modal logic FULL using Ergo. A complete description of the representation will be given in the full paper.

In Ergo we define a new theory for STSs and start by declaring the required constants and primitive inference rules. An example of a declaration of a constant is

```
declare models(_,_)
```

which declares `models` as a two place function symbol whose semantics are as follows.

```
models(State,STSForm)
```

is true if and only if the STS formula `STSForm` is true at `State`.

For example,

```
models(s1, dia(g, X, X > 0))
```

is true if and only if there is a `g` transition from state `s1` with data value `X` satisfying `X > 0`.

After the declarations of the required constants we declare the primitive inference rules (called rules in Ergo). The following three rules come from the STS theory. First, the obvious rule for logical "and".

```
rule models_and_iff ===
    models(State, Exp1 and Exp2)
    <=> models(State, Exp1) and models(State, Exp2).
```

Secondly, a two step rule which extracts first the data variable of the transition (which can be dealt with by standard rules for `ex`, existential quantification), and then deals with the existence of a suitable transition.

```
rule models_ex_dia_iff(x) ===
    models(State, ex x:Type dia(Trans, V, Form))
    <=>
    ex x:Type models(State, dia(Trans, V, Form))
       provided x not_free_in Type,
                x not_free_in State,
                x not_free_in Trans.

rule models_dia_iff ===
    models(State, dia(Trans, V, Form))
    <=>
    ex x is_trans(State, Trans, x)
           and formula(x, V) and models(to(x), Form)
       provided x not_free_in State,
                x not_free_in Trans,
                x not_free_in Form,
                x not_free_in V.
```

These rules are a relatively direct translation of the mathematical definition of FULL. For example, the rule for the last operator defined above $\langle \exists x\ g \rangle$, or `ex x:Type dia(Trans, V, Form)` is

$$t \models \langle \exists x\ g \rangle \Phi \;=\; \begin{array}{l} \text{for some value } v \text{, either for some } t',\ t \xrightarrow{\mathrm{tt}\quad gv} t' \text{ and } t' \models \Phi[v/x] \\ \text{or for some } t',\ t \xrightarrow{b\quad gz} t' \text{ and } b[v/z] \equiv \mathrm{tt} \text{ and } t'_{[v/z]} \models \Phi[v/x] \end{array}$$

In fact, the Ergo representation is more compact since the two cases of data value and data variable do not need to be separated. Also, the representation of the operator (with data before gate) allows separate rules to be applied for dealing with the data and the gate, whereas in the FULL semantics we are forced to deal with them together.

All the STS rules are stated as equivalences so that they can be used as part of a rewrite system using transformational reasoning. After carrying out some example proofs by hand, tactics were developed that use the STS rules as well as inherited classic logic rules such as one-point rules and formula simplification rules. These tactics automatically transform `models` formulae into classical formulae involving the datatypes appearing on transitions.

The next step is to incorporate rules for the data type as sepcified by the ACT ONE equational specification part of the LOTOS behaviour, and develop tactics to simplify terms involving data types. Then the entire proof can be carried out automatically. This ability to incorporate theories for both the processses and the data is the advantage of using a theorem prover to prototype our model checker.

Further work includes refinement of tactics and use of Qu-Prolog to implement tactics more efficiently. A problem yet to be addressed is the representation of behaviours with infinite depth which depend on data parameters.

References

[1] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS based on Symbolic Transition Systems. *The Computer Journal*, 2001. In press.

[2] M. Calder, S. Maharaj, and C. Shankland. An Adequate Logic for Full LOTOS. To appear in Proceedings of Formal Methods Europe'01, LNCS series, Springer-Verlag, 2001.

[3] M. Calder and C. Shankland. A Symbolic Semantics and Bisimulation for Full LOTOS. Technical Report CSM-159, University of Stirling, 2000.

[4] M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138:353–389, 1995.

[5] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[6] C. Pecheur. Using LOTOS for specifying the CHORUS distributed operating system kernel. *Computer Communications*, 15(2):93–102, March 1992.

[7] M. Sighireanu and R. Mateescu. Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): an Experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, Dec. 1998.

[8] C. Stirling. Temporal Logics for CCS. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 660–672. Springer-Verlag, 1989. REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1988.

[9] M. Utting. The Ergo 5 Generic Proof Engine. Technical Report 97-44, Software Verification Research Centre, 1997.

Agents and Roles: Refinement in Alternating-Time Temporal Logic

Pierre-Yves Schobbens
Institut d'Informatique
Facultés Universitaires de Namur
Rue Grandgagnage 21
5000 Namur, Belgium
www.info.fundp.ac.be/~pys
pys@info.fundp.ac.be

Mark Ryan
School of Computer Science
University of Birmingham
Edgbaston
Birmingham B15 2TT, UK
www.cs.bham.ac.uk/~mdr
M.D.Ryan@cs.bham.ac.uk

April 5, 2001

Abstract

We present a notion of refinement between agent-oriented systems defined using alternating-time temporal logic (ATL). The refinement relation provides a framework for defining *roles* in a society of interacting agents, and formalising a relation of *conformance* between agents and roles. The refinement relation also allows us to construct abstractions in order to make verification more tractable.

1 Introduction

In order to make descriptions of reactive systems more tractable, programming languages and specification languages usually have a modularisation construct which allows the decomposition of the system under consideration into objects (sometimes called agents, to emphasise their autonomy). This horizonal decomposition of a system should be integrated with a vertical decomposition, in which we refine abstract descriptions of objects or agents into more concrete ones during the course of the system development. The abstract descriptions can be thought of as roles, which are fulfilled by the concrete agents.

Verifying that a set of agents fulfills its role must be done in the context of the other agents. This idea has lead to the assume/guarantee paradigm: the specification of an agent guarantees its behaviour will be correct, assuming that its environment (formed in part by other agents) is correct. Our goal is to integrate this idea with a framework for agent refinement.

To formalise these intuitions, we propose to use Alternating-time Temporal Logic [3] (ATL), which allows us to describe precisely the properties of the different agents involved in a system, and the *strategies* they have for achieving their goals. ATL is a branching temporal logic based on game theory. It contains the usual temporal operators (next, always, until) plus cooperation modalities $\langle\langle A\rangle\rangle\phi$, where A is a set of players (also called agents). This modality quantifies over the set of behaviours and means that the agents in A have a collective strategy to enforce ϕ, whatever the choices of the other agents. ATL generalises CTL, and similarly ATL* generalises CTL*, μ-ATL generalises the μ-calculus. These logics can be model-checked by generalising the techniques of CTL, often with the same complexity.

We define a very general notion of refinement between alternating-time transition systems (ATS, the models of ATL), which allows splitting or coalescing sets of agents so that a role may be fulfilled by several agents, or more generally several roles fulfilled by several agents. As such, our refinement relation generalises that of [2]. As well as providing a framework for describing agents and the roles they conform to, the refinement relation can be used to construct abstractions that allow us to verify systems efficiently in the manner of Cadence SMV [6].

Outline. In this preliminary version we present our early results. In section 2 we recall the basic concepts of ATL and ATL* and their semantics on ATSs. Section 3 defines refinement and shows some of its properties, and some examples. Finally in section 4 we discuss some directions for future work.

2 Alternating-time temporal logic

Alternating-time temporal logic (ATL) is based on CTL. Let us first recall a few facts about CTL. CTL [4] is a branching-time temporal logic in which we can express properties of reactive systems. For example, properties of cache-coherence protocols [7], telephone systems [8], and communication protocols have been expressed in CTL. One problem with CTL is that it does not distinguish between different sources of non-determinism. In a telephone system, for example, the different sources include individual users, the environment, and internal non-determinism in the telephone exchange. CTL provides the A quantifier to talk about all paths, and the E quantifier to assert the existence of a path. $\mathrm{A}\psi$ means that, no matter how the non-determinism is resolved, ψ will be true of the resulting path. $\mathrm{E}\psi$ asserts that, for at least one way of resolving the non-determinism, ψ will hold. But because CTL does not distinguish between different types of non-determinism, the A quantifier is often too strong, and the E quantifier too weak. For example, if we want to say about a telephone system that *user i can converse with user j*, CTL allows us to write the formulas $A\Diamond talking(i,j)$, $E\Diamond talking(i,j)$ The first one says that in all paths, somewhere along the path there is a state in which i is talking to j, and is clearly much stronger than the intention. The second formula says that there is a path along which i is eventually talking j. This formula is weaker than the intention, because to obtain that path we may have to make choices on behalf of all the components of the system that behave non-deterministically. What we wanted to say is that users i and j can resolve their non-deterministic choices in such a way that, no matter how the other users or the system or the environment behaves, all the resulting paths will eventually have a state in which i is talking j. These subtle differences in expressing the properties we want to check can be captured accurately with ATL.

Alternating-time temporal logic (ATL) [3] generalises CTL by introducing *agents*, which represent different sources of non-determinism. In ATL the A and E path quantifiers are replaced by a path quantifier $\langle\langle A\rangle\rangle$, indexed by a subset A of the set of agents. The formula $\langle\langle A\rangle\rangle\psi$ means that the agents in A can resolve their non-deterministic choices such that, no matter how the other agents resolve their choices, the resulting paths satisfy ψ. We can express the property that user i has the power, or capability, of talking to j by the ATL formula[1]

$\langle\langle i\rangle\rangle\Diamond\mathtt{talking}(i,j)$.

We read $\langle\langle A\rangle\rangle\psi$ as saying that the agents in A can, by cooperating together, force the system to execute a path satisfying ψ. If A is the empty set of agents, $\langle\langle A\rangle\rangle\psi$ says that the system will execute a path satisfying ψ without the cooperation of any agents at all; in other words, $\langle\langle\emptyset\rangle\rangle\psi$ is equivalent to $\mathrm{A}\psi$ in CTL. Dually, $\langle\langle\Sigma\rangle\rangle\psi$ (where Σ is the entire set of agents) is a weak assertion, saying that if all the agents conspire together they may enforce ψ, which is equivalent to $\mathrm{E}\psi$ in CTL.

2.1 ATL and ATL*

Let P be a set of atomic propositions and Σ a set of agents. The syntax of ATL is given by

$$\phi ::= p \mid \top \mid \neg\phi_1 \mid \phi_1 \vee \phi_2 \mid \langle\langle A\rangle\rangle[\phi_1 \,\mathrm{U}\, \phi_2] \mid \langle\langle A\rangle\rangle\Box\phi_1 \mid \langle\langle A\rangle\rangle\bigcirc\phi_1$$

where $p \in P$ and $A \subseteq \Sigma$. We use the usual abbreviations for $\rightarrow$, $\wedge$ in terms of $\neg$, $\vee$. The operator $\langle\langle\ \rangle\rangle$ is a path quantifier, and $\bigcirc$ (*next*), $\Box$ (*always*) and U (*until*) are temporal operators. The logic ATL is similar to the branching-time logic CTL, except that path quantifiers are parameterised

[1] We write $\langle\langle i\rangle\rangle\psi$ instead of $\langle\langle\{i\}\rangle\rangle\psi$.

by sets of agents. As in CTL, we write $\langle\langle A\rangle\rangle\Diamond\phi$ for $\langle\langle A\rangle\rangle[\top \text{ U } \phi]$, and we define the weak-until: $\langle\langle A\rangle\rangle[\phi \text{ W } \psi] = \neg[[A]][\neg\psi \text{ U } (\neg\phi \wedge \neg\psi)]$.

While the formula $\langle\langle A\rangle\rangle\psi$ means that the agents in A can cooperate to make ψ true (they can "enforce" ψ), the dual formula $[[A]]\psi$ means that the agents in A cannot cooperate to make ψ false (they cannot "avoid" ψ). The formulas $[[A]]\Diamond\phi$, $[[A]]\Box\phi$, and $[[A]]\bigcirc\phi$ stand for $\neg\langle\langle A\rangle\rangle\Box\neg\phi$, $\neg\langle\langle A\rangle\rangle\Diamond\neg\phi$, and $\neg\langle\langle A\rangle\rangle\bigcirc\neg\phi$.

The CTL path quantifiers A (all paths) and E (some path) can be recovered in ATL as $\langle\langle\emptyset\rangle\rangle$ and $\langle\langle\Sigma\rangle\rangle$. The logic ATL* generalises ATL in the same way that CTL* generalises CTL, namely by allowing path quantifiers and temporal operators to be nested arbitrarily.

For a subset $A \subseteq \Sigma$ of agents, the fragment $\langle\langle A\rangle\rangle$-ATL of ATL consists of ATL formulas whose only modality is $\langle\langle A\rangle\rangle$, and that does not occur within the scope of a negation. The $\langle\langle A\rangle\rangle$-ATL* fragment of ATL* is defined similarly.

2.2 Alternating transitions systems

Whereas the semantics of CTL is given in terms of transition systems, the semantics of ATL is given in terms of *alternating transition systems* (ATSs). An ATS S over a set of atomic propositions P and a set of agents Σ is composed of (Q, π, δ, I, o), where Q is a set of states and $\pi : Q \to 2^P$ maps each state to the set of propositions that are true in it, and

$$\delta : Q \times \Sigma \to 2^{2^Q}$$

is a transition function which maps a state and an agent to a non-empty set of *choices*, where each choice is a non-empty set of possible next states. If the system is in a state q, each agent a chooses a set $Q_a \in \delta(q, a)$; the system will move to a state which is in $\bigcap_{a\in\Sigma} Q_a$. We require that the system is non-blocking and that the agents together choose a unique next state; that is, for every q and every tuple $(Q_a)_{a\in\Sigma}$ of choices $Q_a \in \delta(q, a)$, we require that $\bigcap_{a\in\Sigma} Q_a$ is a singleton. Similarly, the initial state is specified by $I : \Sigma \to 2^{2^Q}$. I maps each agent to a set of choices. The agents together choose a single initial state: for each tuple $(Q_a)_{a\in\Sigma}$ of choices $Q_a \in I(a)$, we require that $\bigcap_{a\in\Sigma} Q_a$ is a singleton.

In the sequel, we will consider ATS of a particular form: the choices of the agents are expressed by choosing the value of their variables. This is also the choice made in the language Mocha [1]. We therefore stipulate a function $o : P \to \Sigma$ that, for each propositional variable, gives its owner. We define $P_a = \{p \in P \mid o(p) = a\}$ to designate the propositional variables belonging to agent a, and $P_A = \{p \in P \mid o(p) \in A\}$ to a group A. The choices of an agent thus determine the value of its variables, and let the other vary freely:

$$\forall a \in \Sigma, q \in Q, Q_a \in \delta(q, a), \forall X \subseteq P_{\neq a}, \exists q \in Q_a : \pi(q) \cap P_{\neq a} = X$$

where $P_{\neq a} = \{p \in P | o(p) \neq a\}$. A signature is defined as this function o, together with its domain P and codomain Σ.

For two states q and q', we say that q' is a *successor* of q if, for each $a \in \Sigma$, there exists $Q' \in \delta(q, a)$ such that $q' \in Q'$. We write $\delta(q)$ for the set of successors of q; thus,

$$\delta(q) = \bigcap_{a\in\Sigma} \bigcup_{Q\in\delta(q,a)} Q$$

A computation of S is an infinite sequence $\lambda = q_0, q_1, q_2 \ldots$ of states such that (for each i) q_{i+1} is a successor of q_i. We write $\lambda[0, i]$ for the finite prefix $q_0, q_1, q_2, \ldots, q_i$.

Often, we are interested in the cooperation of a subset $A \subseteq \Sigma$ of agents. Given A, we define $\delta(q, A) = \{\bigcap_{a\in A} Q_a \mid Q_a \in \delta(q, a)\}$. Intuitively, when the system is in state q, the agents in A can choose a set $T \in \delta(q, A)$ such that, no matter what the other agents do, the next state of the system is in T. Note that $\delta(q, \{a\})$ is just $\delta(q, a)$, and $\delta(q, \Sigma)$ is the set of singleton successors of q.

Example 1 ([3]) Consider a system with two agents "user" u and "telephone exchange" e. The user may lift the handset, represented as assigning value true to the boolean variable "offhook".

The exchange may then send a tone, represented by assigning value true to the boolean variable "tone". Initially, both variables are false. Clearly, obtaining a tone requires collaboration of both agents.

We model this as an ATS $S = (Q, \pi, \delta, I)$ over the agents $\Sigma = \{u, e\}$ and propositions $P = \{\text{offhook,tone}\}$. Let $Q = \{00, 01, 10, 11\}$. 00 is the state in which both are false, 01 the state in which "offhook" is false and "tone" is true, etc. (thus, $\pi(00) = \emptyset$, $\pi(01) = \{tone\}$, etc.). The transition function δ and initial states I are as indicated in the figure.

$\delta(q,a)$	u	e
00	$\{\{00,01\},\{10,11\}\}$	$\{\{00,10\}\}$
10	$\{\{10,11\}\}$	$\{\{00,10\},\{01,11\}\}$
01	$\{\{00,01\},\{10,11\}\}$	$\{\{01,11\}\}$
11	$\{\{10,11\}\}$	$\{\{01,11\}\}$
I	$\{\{00,01\}\}$	$\{\{00,10\}\}$

Figure 1: The transition function of the ATS.

2.3 Semantics

The semantics of ATL uses the notion of *strategy*. A *strategy* for an agent $a \in \Sigma$ is a mapping $f_a : Q^+ \to 2^Q$ such that $f_a(\lambda \cdot q) \in \delta(q, a)$ with $\lambda \in Q^*$. In other words, the strategy is a recipe for a to make its choices. Given a state q, a set A of agents, and a family $F_A = \{f_a \mid a \in A\}$ of strategies, the *outcomes* of F_A from q are the set $out(q, F_A)$ of all computations from q where agents in A follow their strategies, that is,

$$out(q_0, F_A) = \{\lambda = q_0, q_1, q_2, \cdots \mid \forall i,\ q_{i+1} \in \delta(q_i) \cap \big(\bigcap_{a \in A} f_a(\lambda[0,i])\big)\}.$$

If $A = \emptyset$, then $out(q, F_A)$ is the set of all computations, while if $A = \Sigma$ then it consists of precisely one computation.

The semantics of ATL* is as CTL*, with the addition of:

- $q \models \langle\!\langle A \rangle\!\rangle \psi$ if there exists a set F_A of strategies, one for each agent in A, such that for all computations $\lambda \in out(q, F_A)$ we have $\lambda \models \psi$.

Remark 2 To help understand the ideas of ATL, we state below some validities, and more surprising non-validities.

1. If $A \subseteq B$, then $\langle\!\langle A \rangle\!\rangle \psi \to \langle\!\langle B \rangle\!\rangle \psi$, and $[[B]]\psi \to [[A]]\psi$. Intuitively, anything that A can enforce can also be enforced by a superset B; and if anything that B is powerless to prevent cannot be prevented by a subset of B.

2. In CTL*, A distributes over $\wedge$. But in general in ATL*, $\langle\!\langle A \rangle\!\rangle(\psi_1 \wedge \psi_2)$ only implies $(\langle\!\langle A \rangle\!\rangle \psi_1) \wedge (\langle\!\langle A \rangle\!\rangle \psi_2)$. The first formula asserts that A can enforce $\psi_1 \wedge \psi_2$, while the second is weaker, asserting that A has a way to enforce ψ_1 and another, possibly incompatible, way to enforce ψ_2. Similarly, $\langle\!\langle A \rangle\!\rangle(\psi_1 \vee \psi_2)$ and $\langle\!\langle A \rangle\!\rangle \psi_1 \vee \langle\!\langle A \rangle\!\rangle \psi_2$ are different (for $A \neq \Sigma$). The first one asserts that A can enforce $\psi_1 \vee \psi_2$, but which of the two is true might be chosen by others. This is weaker than the second formula, which asserts that A can guarantee ψ_1, or A can guarantee ψ_2, but nobody can choose which. The strongest variant where A can choose, is expressed as: $\langle\!\langle A \rangle\!\rangle(\psi_1 \wedge \neg\psi_2) \wedge \langle\!\langle A \rangle\!\rangle(\psi_2 \wedge \neg\psi_1)$.

3. The cooperation operator can often be repeated inside a temporal operator, changing slightly the meaning of the formula, for instance: $\langle\!\langle A \rangle\!\rangle \Box\Diamond\phi \to \langle\!\langle A \rangle\!\rangle \Box \langle\!\langle A \rangle\!\rangle \Diamond\phi$. Here, this is because the strategies F_A that A use in the outer modality may be adapted for the inner modality, by shifting its time: each $f'_a(x)$ is simply $f_a(\lambda \cdot x)$, where λ is the path linking the points of evaluation of the two modalities. (Note the CTL* validities $\mathrm{E}\Box\Diamond\phi \to \mathrm{E}\Box\mathrm{E}\Diamond\phi$ and $\mathrm{A}\Box\Diamond\phi \leftrightarrow \mathrm{A}\Box\mathrm{A}\Diamond\phi$.)

3 Refinement of agents

In this section we introduce refinement between ATSs. Our aim is to present a framework which provides

- A method for constructing systems by assembling components. The system is specified as a collection of roles, and the component agents are shown to conform to the roles.
- A method for verifying agents efficiently, by abstracting their environment (similar to [6], but generalising from LTL to ATL). An agent is verified under the guarantees made by the roles of its environment. This addresses the state explosion problem, since it is simpler to verify against the roles of the environment than against the agents implementing the roles.

Example 3 The sliding-window protocol (SWP) is a communications protocol designed to guarantee communication over an unreliable channel. The channel may loose, duplicate, and re-order messages. We assume, however, that the channel is not completely unreasonable; thus,

- Although it might loose some messages, it does not loose all of them. More precisely, if a given message is sent often enough, it will eventually get through.
- Although it might re-order messages, the degree to which it re-orders is bounded. The channel will not swap a given message with other messages more than N times.

The agents involved in the abstract specification of the SWP are depicted in Figure 2. Here is a

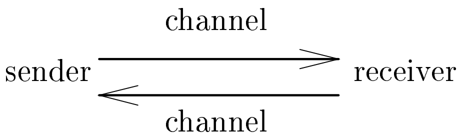


Figure 2: Roles for the sliding-window protocol

very abstract account of the SWP. The message to be sent is split into packets, and the packets are passed in order to the sender. In order to cope with the deficiencies of the channels, the sender may send individual packets multiple times, until an acknowledgment is received from the receiver. Since the acknowledgments are also sent along an unreliable channel, they might need to be sent several times too. In order to cope with re-ordering, the sender labels messages with a number in the set $\{0, \ldots, M\}$, where $M > N$. When all the numbers are used up, the sender starts again from 0. The sender maintains an interval $[i, i + w]$, known as its window, of size $w < N$. Initially $i = 0$. The sender non-deterministically picks a packet whose sequence number is in the window, and sends it. It records any acknowledgments it receives; when it has received an acknowledgment for the ith message, it may advance its window by incrementing i. The receiver, whose job is to assemble the incoming message and send out acknowledgments, may be described in similar abstract terms.

This account is highly non-deterministic; it does not specify what order the sender should send the packets in its window, or what order the receiver should send the acknowledgments; this is inefficient. It allows the sender to resend messages which have already been acknowledged, which is wasteful. A more concrete specification of the SWP is given in [5], where the sender is split into three processes: the *trans*, which sends the packets initially and in order; the *ack-recv*, which records which messages have been acknowledged, and the *re-trans*, which re-transmits packets for which an acknowledgment has not been received before a given timeout. The reciever is also split into several components.

Trans, ack-rcvr, and re-trans jointly play the role of sender, and accept and ack-send jointly play the role of receiver (Figure 3).

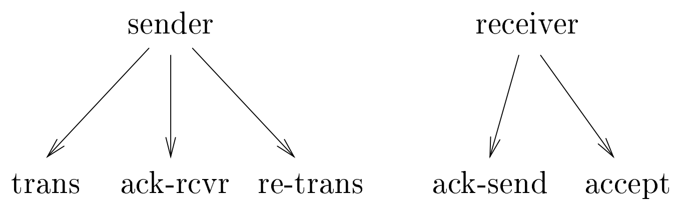


Figure 3: Agents and Roles for the sliding-window protocol

When proving properties of the concrete agents, we assume only the guarantees made by the roles of the other agents. This is a weaker assumption than the guarantees actually provided by the other agents, but it is leads to more efficient verification. Indeed, in a multi-layered example, while proving the properties of one agent we would assume the guarantees of the other agents in the most abstract layer possible.

This example helps us motivate the definition of ATS refinement: a set of agents is refined, while the complement set of agents (forming the refinement) may be abstracted.

The formal definition of refinement is split into two parts. Signature refinement deals with the symbols in the vocabulary. Next, ATS refinement extends a signature refinement, and deals with behaviours.

Definition 4 Let Σ and Σ' be disjoint sets of agents with variables P and P' respectively. A *signature refinement from* $A \subseteq \Sigma$ *to* $A' \subseteq \Sigma'$ is: a relation $f \subseteq \Sigma \times \Sigma'$ between agents in both societies, and a relation $g \subseteq P \times P'$, such that:

- f links agents in A to agents in A', and agents outside A to agents outside A'. To define this formally, we extend f to sets of agents in the usual way: $f(B) = \{a' \mid \exists a \in B f(a,a')\}$ and $f^{-1}(B') = \{a \mid \exists a' \in B' f(a,a')\}$. A set of agents $B \in \Sigma$ *corresponds to* $B' \in \Sigma'$ iff $B' = f(B)$ and $f^{-1}(B') = B$. Note that some sets of agents will not have a correspondent. We require that A corresponds to A' and $\bar{A}$ to $\bar{A}'$ (where $\bar{A} = \Sigma \setminus A$ and $\bar{A}' = \Sigma' \setminus A'$).

- g respects ownership: $g(p,p')$ implies $f(o(p), o(p'))$. Additionally, the more detailed description of the component should at least include the variables required in its role, thus g is a function from $P_A = \{p \mid o(p) \in A\}$, and conversely, g^{-1} is a function on $P_{\bar{A}'}$.

Example 5 $A = \{\text{sender}\} \subseteq \Sigma = \{\text{sender}, \text{receiver}\}$. $A' = \{\text{trans, ack-rcvr, re-trans}\} \subseteq \Sigma' = \{\text{trans, ack-rcvr, re-trans, receiver}\}$ refining the sender. In this case, f is shown as the left three arrows of figure 3.

Signature refinements have the following properties:

Theorem 6 1. *The identity is a signature refinement;*

2. *Signature refinements compose: if (f_1, g_1) is a signature refinement from $A_1 \subseteq \Sigma_1$ to $A_2 \subseteq \Sigma_2$, and (f_2, g_2) is a signature refinement from $A_2 \subseteq \Sigma_2$ to $A_3 \subseteq \Sigma_3$, then $(f_1 \circ f_2, g_1 \circ g_2)$ is a signature refinement from $A_1 \subseteq \Sigma_1$ to $A_3 \subseteq \Sigma_3$.*

Given such a signature refinement, we can now consider how to translate formulae. Not all formulae can be translated: they may involve agents or variables that have no correspondent. The translation T along f, g of a formula is thus: $T(p) = g(p)$ assuming g is functional on p; $T(\langle\langle A \rangle\rangle \phi) = \langle\langle f(A) \rangle\rangle T(\phi)$ assuming $f(A)$ corresponds to A; and the other logical operators translate unchanged. If the assumptions above are not fulfilled, the formula cannot be translated.

A refinement between signatures can sometimes be extended to a refinement between ATS $S = (\Sigma, Q, \pi, \delta, I)$ and $S' = (\Sigma', Q', \pi', \delta', I')$. We then say that the agents A' are conformant to the role A.

Definition 7 An *ATS refinement* r extending a signature refinement $s = (f, g)$ from $A \subseteq \Sigma$ to $A' \subseteq \Sigma'$is a triple (H, C, D), where:

1. $H \subset Q \times Q'$ is a relation between states such that corresponding variables have the same value, formally: if $H(q, q'), g(v, v')$, then $v \in \pi(q)$ iff $v' \in \pi'(q')$),
2. $C : H \to \delta'(q', A') \to \delta(q, A)$: C gives for any pair (q, q') in the relation H and for any choice possible for the component A' in q', a corresponding choice for the role A in q
3. conversely $D : H \to \delta(q, \bar{A}) \to \delta'(q', \bar{A}')$,

such that $\forall(q, q') \in H, \forall T \in \delta(q, \bar{A}), \forall R' \in \delta'(q', A') : (T \cap C(q, q')(R')) \times (D(q, q')(T) \cap R') \subseteq H$.

H relates states in the abstract system with those of the refined system. The intuition for C and D is this: any choices A' makes in the refined system must correspond to choices A makes in the original system. C maps A''s choices to A's choices. The environment goes the other way; in S' it is more abstract, so D maps $\bar{A}$'s choices to $\bar{A}'$'s choices. The 'such that' part of the definition guarantees that, after these choices have been resolved, the two systems are again in H-related states. Note that since T and $C(q, q')(R)$ together define the choices of all agents, their intersection is a singleton, and similarly for $D(q, q')(T) \cap R$.

If the agents A of the abstract system S are viewed as a role, then this definition ensures that the agents A' of S' conform to that role. Note that again the notion of conformance goes in opposite directions for the system and for its environment.

Note that this definition is similar to the definition of alternating refinement in [2]. Our definition is more general than theirs, since it allows the agents and the variables to be different in the two systems. Our definition implies their definition:

Remark 8 If $r = (f, g, H, C, D)$ is an ATS refinement from $S = (\Sigma, Q, \pi, \delta, I)$ to $S' = (\Sigma', Q', \pi', \delta', I')$ and $\Sigma = \Sigma'$, $P = P'$, and f, g are identity, then $S \leq_{\bar{A}} S'$ in the sense of [2].

Proof The definition of $S \leq_{\bar{A}} S'$ is that there exists $H \subseteq Q \times Q'$ such that $H(q, q')$ implies $\pi(q) = \pi'(q')$ and $\forall T \in \delta(q, \bar{A})\, \exists T' \in \delta'(q', \bar{A})\, \forall R' \in \delta'(q', A)\, \exists R \in \delta(q, A)\, (T \cap R) \times (T' \cap R') \subseteq H$. As R may depend on T as well as R' in this definition, while $C(q, q')(R')$ depends on R' but not on T in ours, their definition is weaker.

Now ATS refinements have the properties:

Theorem 9 *1. The identity is an ATS refinement;*

2. ATS refinements compose vertically: if $r_1 = (f_1, g_1, H_1, C_1, D_1)$ is a ATS $S_1 \to S_2$ refinement from $A_1 \subseteq \Sigma_1$ to $A_2 \subseteq \Sigma_2$, and $r_2 = (f_2, g_2, H_2, C_2, D_2)$ is a $S_2 \to S_3$ refinement from $A_2 \subseteq \Sigma_2$ to $A_3 \subseteq \Sigma_3$, then there is a $S_1 \to S_3$ refinement r_3 from A_1 to A_3, which may also be written $r_1; r_2$, given by $(f_1 \circ f_2, g_1 \circ g_2, H_1 \circ H_2, C_1 \circ C_2, D_1 \circ D_2)$, where $\circ$ is the relation composition: $H_1 \circ H_2 = \{(q_1, q_3) \mid \exists q_2 (q_1, q_2) \in H_1 \wedge (q_2, q_3) \in H_2\}$.

Theorem 10 *If there is an ATS refinement from $A \subseteq \Sigma$ to $A' \subseteq \Sigma'$, then any translatable $\langle\langle A'\rangle\rangle, [[\bar{A}']]$-ATL formula valid on the ATS S' is also valid on S when translated; conversely, any translatable $\langle\langle \bar{A}\rangle\rangle, [[A]]$-ATL formula valid on S is also valid on S' when translated.*

Proof Use remark 8 and theorem 6 of [2].

Horizontal composition of refinements We use the refinement from $A \subseteq \Sigma$ to $A' \subseteq \Sigma'$ when we want to prove that the agents in A' satisfy the role A. The agents in A' are more *concrete* than those in A. As already noted, the definition of refinement means that the agents in $\bar{A}'$ may be more *abstract* than those in $\bar{A}$.

Now suppose we have two refinements, $r_1 = (f_1, g_1, H_1, C_1, D_1)$ and $r_2 = (f_2, g_2, H_2, C_2, D_2)$, such that r_1 is an $S \to S_1$ refinement from $A \subseteq \Sigma$ to $A_1 \subseteq \Sigma_1$, and a $S \to S_2$ refinement from $B \subseteq \Sigma$ to $B_2 \subseteq \Sigma_2$. Refinement r_1 refines the agents in A, and r_2 refines those in B. We would like to be able to put these refinements together, to form a refinement on $A \cup B$. This would allow us to implement the roles A and B independently, and verify the implementations independently. Putting the refinements together means that we obtain a correct refinement of $A \cup B$ with no additional verification. Recall that r_1 refines A, but may abstract B; and r_2, which refines B, may abstract A. Therefore, the composition of r_1 and r_2, noted $r_1 \| r_2$ in Figure 4, has to combine the refinement of A by r_1 and the refinement of B by r_2. We stipulate that $A \cap B$ is empty, in order to avoid the possibility that these refinements are incompatible on their common part.

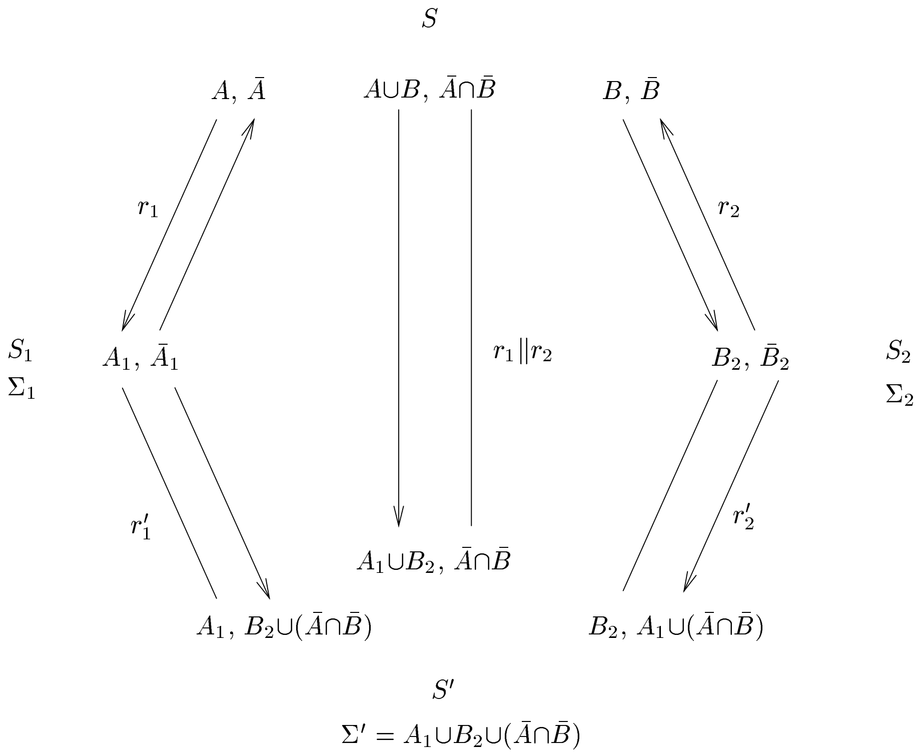


Figure 4: Horizontal composition of refinements.

Example 11 (continued) Let S be the abstract description of the sliding-window protocol (SWP), having agents $\Sigma = \{$sender, receiver, channel1, channel2$\}$. We decide to implement the sender and the receiver separately (Figure 5). The set $A = \{$sender$\}$ is implemented as $A_1 = \{$trans, ack-rcvr, re-trans$\}$. In this example, $\bar{A}_1 = \bar{A} = \{$channel1, channel2, receiver$\}$. We check that this implementation, S_1, is a refinement of S from A to A_1.

The receiver is also implemented. $B_2 = \{$receiver$\}$ is refined into $B_2' = \{$accept, ack-send$\}$.

In the general framework of Figure 4 and Theorem 12 below, $\bar{A}_1$ is allowed to be an abstraction of $\bar{A}$. This is useful for model checing, because S_1 will be more abstract and so checking the refinement will be computationally cheaper.

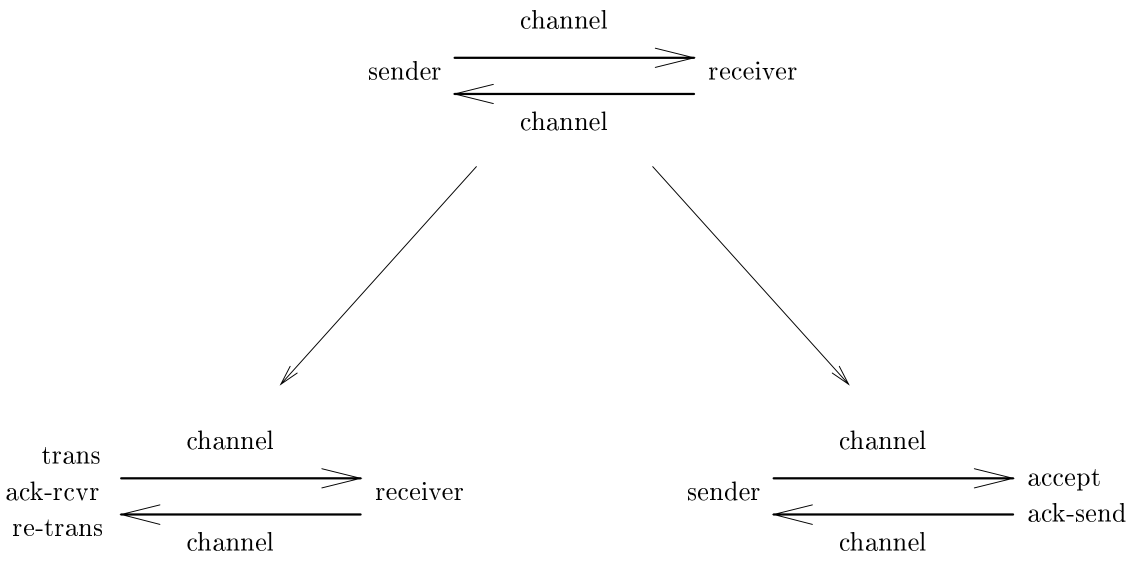


Figure 5: Independently implementing the sender and the receiver.

Theorem 12 *ATS refinements compose horizontally: if $r_1 = (f_1, g_1, H_1, C_1, D_1)$ is an $S \to S_1$ refinement from $A \subseteq \Sigma$ to $A_1 \subseteq \Sigma_1$, and $r_2 = (f_2, g_2, H_2, C_2, D_2)$ is an $S \to S_2$ refinement from $B \subseteq \Sigma$ to $B_2 \subseteq \Sigma_2$ such that $A \cap B = \emptyset$, then we can build r, also noted $r_1 \| r_2$, an $S \to S'$ refinement from $A \cup B \subseteq \Sigma$ to $A_1 \cup B_2 \subseteq \Sigma'$, such that there are refinements:*

1. *r_1', an $S_1 \to S'$ refinement from $\bar{A}_1 \subseteq \Sigma_1$ to $B_2 \cup (\bar{A} \cap \bar{B}) \subseteq \Sigma'$, and*
2. *r_2', an $S_2 \to S'$ refinement from $\bar{B}_2 \subseteq \Sigma_2$ to $A_1 \cup (\bar{A} \cap \bar{B}) \subseteq \Sigma'$,*

and any other such $S \to S'$ refinement r' from $A \cup B$ to C can be decomposed through r: there is r'' such that $r' = r; r''$. The situation is shown in figure 4.

Example 13 (continued) This shows that the two implementations can be put together, and that the result refines the original specification. Moreover, we do not have to check this (which would be computationally expensive, because of the size of S'); we simply check the refinements r_1 and r_2, and the theorem guarantees the refinement $r_1 \| r_2$.

Proof We construct $S' = (\Sigma', Q', \pi', \delta', I')$ and the $S \to S'$ refinement $r = (f, g, H, C, D)$:

- $\Sigma' = A_1 \cup B_2 \cup (\bar{A} \cap \bar{B})$
- $Q' = \{(q, q_1, q_2) \mid H_1(q_1, q) \wedge H_2(q_2, q)\}$
- $v \in \pi'(q, q_1, q_2)$ if $o(v) \in A_1$ and $v \in \pi_1(q_1)$; or $o(v) \in B_2$ and $v \in \pi_2(q_2)$; or $o(v) \in \bar{A} \cap \bar{B}$ and $v \in \pi(q)$.
- $\delta'(a, (q, q_1, q_2)) =$
$$\begin{cases} \{C_1(q_1, q)(T_1) \times T_1 \times D_2(q, q_2)(C_1(q_1, q)(T_1)) \cap Q' \mid T_1 \in \delta_1(a, q_1)\} & \text{if } a \in A_1 \\ \ldots\text{(similar)} & \text{if } a \in B_2 \\ \{T \times D_1(q, q_1)(T) \times D_2(q, q_2)(T) \cap Q' \mid T \in \delta(q, a)\} & \text{if } a \in \bar{A} \cap \bar{B} \end{cases}$$
- $f(a, a')$ if $a \in A$ and $f_1(a, a')$; or $a \in B$ and $f_2(a, a')$; or $a \in \bar{A} \cap \bar{B}$ and $a = a'$.

- $g(v, v')$ if $o(v) \in A$ and $g_1(v, v')$; or $o(v) \in B$ and $g_2(v, v')$; or $o(v) \in \bar{A} \cap \bar{B}$ and $v = v'$.
- $H(q', (q, q_1, q_2)) \Leftrightarrow q = q'$
- $C(q, (q, q_1, q_2))(T \times T_1 \times T_2) = T$
- $D(q, (q, q_1, q_2))(T) = T \times D_1(q, q_1)(T) \times D_2(q, q_2)(T)$

It is straightforward to prove that S' is an ATS (the main part of which is to show that $\delta'(\Sigma', q)$ is a singleton). We prove that $r : S \to S'$ is a refinement:

1. f links $A \cup B$ to some $C' \in \Sigma'$; in fact $C' = A_1 \cup B_2$.
2. g respects ownership: if $g(v, v')$ then, for instance, $o(v) \in A$ and $g_1(v, v')$; the second conjunct implies $f_1(o(v), o(v'))$, which together with $o(v) \in A$ implies $f(o(v), o(v'))$
3. g is functional on $P_{A\cup B}$, since g_1 is functional on P_A and g_2 functional on P_B.
4. We show $H(q', (q, q_1, q_2))$ and $g(v, v')$ imply $v \in \pi(q')$ iff $v' \in \pi'(q, q_1, q_2)$. Since $H(q', (q, q_1, q_2))$, $q = q'$. Now we distinguish between the usual three cases: $o(v)$ may be in A, B, or $\bar{A} \cap \bar{B}$. For example, if $o(v) \in A$ then $g(v, v')$ implies $g_1(v, v')$, therefore $o'(v') \in A_1$. Since $(q, q_1, q_2) \in Q'$, $H_1(q, q_1)$. Therefore, $v \in \pi(q')$ iff $v' \in \pi_1(q_1)$. The other cases are similar.
5. Finally, we need to show that $H(q, (q, q_1, q_2))$, $T \in \delta(q, A \cup B)$, $R \in \delta'((q, q_1, q_2), \bar{A} \cap \bar{B})$: $(T \cap C(q, (q, q_1, q_2))(R)) \times (D(q, (q, q_1, q_2))(T) \cap R) \subseteq H$. Note that $T = \bigcap_{a \in A\cup B} Q_a$ for some choices $Q_a \in \delta(q, a)$, $a \in A \cup B$, and $R = \bigcap_{a \in \bar{A}\cap\bar{B}} (Q_a \times D_1(q, q_1)(Q_a) \times D_2(q, q_2)(Q_a)) \cap Q'$, for again some choices $Q_a \in \delta(q, a)$, $a \in \bar{A} \cap \bar{B}$. Let us fix these choices of Q_a, $a \in \Sigma'$. Suppose $(s, (s', s_1, s_2) \in (T \cap C(q, (q, q_1, q_2))(R)) \times (D(q, (q, q_1, q_2))(T) \cap R)$. Then $\{s\} = \bigcap_{a \in \Sigma'} Q_a$ and $\{(s', s_1, s_2)\} = \bigcap_{a \in \Sigma'} (Q_a \times D_1(q, q_1)(Q_a) \times D_2(q, q_2)(Q_a)) \cap Q'$. Therefore, $s = s'$.

The refinement $r_1' : S_1 \to S'$ is given by

- $f_1' \subseteq \Sigma' \times \Sigma_1$ given by: $f_1'(a', a_1)$ iff $a_1 = a' \in A_1$; or $a_1 \notin A_1$ and $a' \in B_2$ and $(a_1, a') \in f_2 \circ f_1$; or $a_1 \notin A_1$ and $a' \in \bar{A} \cap \bar{B}$ and $(a_1, a') \in f_1$.
- $H_1'(q_1', (q, q_1, q_2)) \Leftrightarrow q_1 = q_1'$
- $C_1'(q_1, (q, q_1, q_2))(T_1) = C_1(q_1, q)(T_1) \times T_1 \times D_2(q, q_2)(C_1(q_1, q)(T_1))$
- $D_1'(q_1, (q, q_1, q_2))(T \times T_1 \times T_2) = T_1$.

The refinement $r_2' : S' \to S_2$ is defined similarly, and we must again check that r_1 and r_2 are a refinement (omitted).

Finally, we show that any other such $S \to S'$ refinement r' from $A \cup B$ to C can be decomposed through r: there is r'' such that $r' = r; r''$: . . .

4 Conclusions

Our refinement framework allows the system developer to implement various components of the system independently. The results can be put together automatically to form an implementation of the whole. This guaranteed interoperability: in the example, any implementation of the sender is compatible with any implementation of the receiver.

This paper is preliminary. Showing that the roles A and B may be refined independently is an important step in building a refinement calculus suitable for making abstractions to address the state explosion problem in ATL model checking. We intend to continue this work by efficiently computing refinements and developing examples.

References

[1] R. Alur, H. Anand, R. Grosu, F. Ivancic, M. Kang, M. McDougall, B.-Y. Wang, L. de Alfaro, T. Henzinger, B. Horowitz, R. Majumdar, F. Mang, C. Meyer, M. Minea, S. Qadeer, S. Rajamani, and J.-F. Raskin. *Mocha User Manual.* University of California, Berkeley. `www.eecs.berkeley.edu/~mocha`.

[2] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 163–178. Springer-Verlag, 1998.

[3] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 100–109. IEEE Computer Society Press, 1997.

[4] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In D. Kozen, editor, *Logic of Programs Workshop*, number 131 in LNCS. Springer Verlag, 1981.

[5] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1990.

[6] K. McMillan. The SMV language. Available from www-cad.eecs.berkeley.edu/~kenmcmil, June 1998.

[7] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[8] M. C. Plath and M. D. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 2001. In print.

Compositional Verification of an Object-Based Model for Reactive Systems

M. Sirjani*, A. Movaghar*, M.R. Mousavi*

Department of Computer Engineering
Sharif University of Technology
Tehran,Iran

1 Introduction

Reactive systems have an ongoing interaction with their environment, accepting requests and producing responses. Correct and reliable construction of reactive systems is particularly important and challenging. So, using a formal method to establish mathematical proof of correctness of such systems is necessary.

A formal method consists of three major components: a model for describing the behavior of the system, a specification language to embody correctness requirements, and an analysis method to verify the behavior against the correctness requirements [8]. We choose an object-based model as the modelling language to describe the system, linear temporal logic as the specification language, and compositional verification for verifying that the model meets its specification.

In system verification, a main obstacle to the use of automatic methods is the state-explosion problem, which is the exponential increase in the number of system states caused by a linear increase in the number of system components or variables [6]. Compositional verification attempts to overcome the state-explosion problem by exploiting the modular structure that is naturally present in many system designs.

Compositional verification can decrease the complexity of problem when the model is naturally decomposable [10]. So, a model consisting of inherently independent modules is suitable for compositional verification. Object-based modelling is based on abstraction, encapsulation, and information hiding, using modules with high independence. Therefore, we choose an object-based model.

*emails: msirjani@mehr.sharif.ac.ir, { movaghar,mousavi}@sharif.ac.ir

2 Alecs Model

The model proposed here is similar to the actor model [2, 3, 4] in its independent active objects, asynchronous message passing, dynamically changing topology, and unlimited buffer for messages. This model does not support dynamic creation of active objects to decrease the complexity of the model and hence make the verification process easier. We call our model Alecs[1] and each active object alec. In Alecs one has to first define the classes and then introduce alecs as instances of that class. Often, we have several similar agents in a system that act concurrently and interact with one another. This feature allows us to reuse the class code in the model. Verification of a module specification and then composing the specifications is the basis of compositional verification; thus, in a model like ours, verification of a class property can be generalized to all of its instances. Therefore, we have reusability in the verification approach as well.

Independence of the alecs is important in compositional verification as well as in modelling. Alecs directly supports encapsulation and information hiding by separating the interface and the body of each alec's class definition. The interface part includes all the information necessary for interacting with other alecs and also for verification. Therefore, it is not just a modelling concern but a verification concern as well. This partitioning improves the readability too. Unlike other tasks in the verification of finite-state systems, which have been largely automated, current modular verification techniques still rely on user guidance [5]. As a result, readability is an important feature.

To overcome a common criticism on the actor model [4], an alec's semantics is defined such that the composition of two alecs is another alec.

The computation takes place by message passing and the execution of message servers. For each alec, there is an unlimited FIFO buffer called mailbox for arriving messages . There is a server (i.e. method) for each arrived message. When a message reaches the head of the mailbox its server is invoked, and the message is omitted from the mailbox and passed to the server. Each alec can send messages to its known alecs which are in the known alecs list. This list can be updated by sending alec's names in the messages. The topology of the model is based on this known alecs list.

The definition of the system includes the classes and the alecs in the running system. There is not a complete inheritance hierarchy, the classes act like templates for definition of the alecs. Alecs are instantiated from the classes and are composed in parallel to make the whole reactive system. The init server is invoked to initialize the system. Alecs are executed in parallel; each, having a single thread of execution, taking a message from the mailbox and invoking the server. If there is no message waiting in the mailbox then the alec will wait idle. Message passing is asynchronous and the sender alec continues its execution after sending the message. A part of computation and nondeterminism is modelled by asynchrony in message passing.

[1] Alive objects

3 Compositional Verification of Alecs

A necessary condition for compositional verification is that an alec's variable values shouldn't change when it is composed with other alecs, i.e. concurrent execution of alecs shouldn't change their specifications [1].

Executing message servers is the only way to change alecs' variable values. So, first we verify an alec property considering all possible environment conditions. It makes a state space presenting all the state transitions of the alec that can take place in the running system. Then it is obvious that safety property of an alec holds even if it is composed with another alec. So, because of alecs independence and encapsulation there is no need for explicit conditions in compositional verification of them.

We can suggest a strategy for compositional verification of safety properties. In this strategy, if P and Q are two alecs, and $\varphi_{P||Q}$ is a safety property of $P||Q$, we can show that $\varphi_{P||Q}$ is a property of compound alec, if we find safety properties for P and Q, such that

1. φ_P is a safety property of P,
2. φ_Q is a safety property of Q, and
3. $(\varphi_P \wedge \varphi_Q) \Rightarrow \varphi_{P||Q}$.

Also, we can use compositional minimization by finding the interface alec Q' corresponding to P. Then, we may model check a property on parallel composition of P and Q' $(P||Q')$, and conclude that the property is valid for $P||Q$ [9].

4 An Example

To show the power of Alecs model, we used it to specify a railroad control system, which has been used as a typical verification example [5, 7]. The system is made up of a bridge controller class(*BridgeController*) and a train class(*Train*). The bridge controller class prevents collisions between the two trains by ensuring the train safety requirement and the train class is a template for defining trains moving toward the bridge in different directions. Running system of this model consists of two train alecs(Train1 and Train2) and one controller alec(theController) which run concurrently as depicted in figure 1.

The system safety requirement, is that at any given time at most only one train is on the bridge. It is specified in temporal logic as follows:

$$\varphi_{sys} = [](\neg(Train1.OnTheBridge \wedge Train2.OnTheBridge))$$

We proved the correctness of above statement using minimization rule on safety properties of $BridgeController$ and $(BridgeController||Train)$. The state space of the system, with traditional verification methods, consists of 288 states. However, applying compositional verification results in just 56 states.

```
ActiveClass BridgeController
{
 Interface :
   KnownObjects :
    T[ 1..2] : Train;
   MsgServers :
    Arrive(sender),
    Leave(sender);
   Observational :
    Signal[ 1..2]: (green, red) ;

 Body :
  theOtherOne: Intege r;
   isWaiting[ 1..2]: Boolean;
   Arrive(sender)
   {
    theOtherOne := (sender mod 2) + 1;
     if Signal[theOtherOne] := red then
     {
       Signal[sender]:= green;
       send (sender, YouMayPass);
      else
         isWaiting[sender] := true;
      }

   }

   Leave(sender)
   {
     theOtherOne := (sender mod 2) + 1;
     Signal[sender] := red;
      if isWaiting[theOtherOne] then
      {
           send (T[theOtherOne],
                        YouMayPass);
       }
   }
   Init()
   {
    Signal[ 1] := red; Signal[ 2] := red;
    isWaiting[ 1] := false;
    isWaiting[ 2] := false;
   }
}
```

```
ActiveClass Train
{
 Interface :
   KnownObjects :
     Controller : BridgeController;
   MsgServers :
     YouMayPass(),
      ReachBridge(),Passed() ;
   Observational :
     OnTheBridge : Boolean;
  Body :
    ReachBridge()
    {
     send(Controller, Arrive(self));
     }

    YouMayPass()
    {
     OnTheBridge:= true;
      send (self, Passed)
    }

    Passed()
   {
    OnTheBridge:=false;
    send(Controller,Leave(self));
    send (self, ReachBridge);
   }

    Init()
    {
      send(self, Passed);
    }
}

ActiveObjects :
   Train 1, Train 2 :Train;
   theController : BridgeController;
Composition :
     Train 1 || theController || Train 2
Init()
{
Train 1.Controller := theController;
Train 2.Controller := t heController;
theController.T[ 1] := Train 1;
theController.T[ 2] := Train 2;
}
```

Figure 1: A railroad control system in Alecs

5 Concluding Remarks

In the future, we plan to add real-time constraints to the model, increase the reusability of the model, and add the possibility of dynamic creation of alecs. Since decomposing the system into modules, and specifying the properties are currently the designer's responsibility, we shall make them as automatic as possible. The first step is to present some guidelines and algorithms in this area. In this paper, we focus on safety properties only, verification of progress properties may also be considered. Induction methods and special optimization algorithms for model checking of the Alecs will also be needed in the future.

References

[1] Abadi L. and Lamport L., Composing Specifications, in *ACM Transactions on Programming Languages and Systems*, vol. 15, No. 1, pp. 73-133, January 1993.

[2] Agha G. and Hewitt C., Concurrent Programming Using Actors, in Yonezawa A., Tokoro M. Eds.,*Object-Oriented Programming*, MIT Press, pp. 37-53, 1988.

[3] Agha G., The Structure and Semantics of Actor Languages, *Foundations of Object-Oriented Languages: REX School Workshop*, LNCS 489, pp. 1-59, 1991.

[4] Agha G., Mason I., Smith S. and Talcott C., A Foundation for Actor Computation, *Journal of Functional Programming*, No. 7, pp. 1-72, 1997.

[5] Alur R. and Henzinger T.A., *Computer Aided Verification*, Draft, 1999.

[6] Alur R., de Alfaro L., Henzinger T.A. and Mang F.Y.C., Automating Modular Verification, in *Proceedings of the Tenth International Conference on Concurrency Theory*, LNCS 1664, Springer-Verlag, pp. 83-97, 1999.

[7] Bjorner N.S., Manna Z., Sipma H.B. and Uribe T.E., Deductive Verification of Real-time Systems Using Step, in *Proceedings of ARTS-97*, LNCS 1231, Springer-Verlag, pp. 22-43, 1997.

[8] Manna Z. and Pnueli A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[9] Peng H. and Tahar S., A Survey on Compositional Verification, *Technical Report Dept of ECE*, Concordia University, 1998.

[10] Roever W.P., The need for Compositional Proof Systems: A Survey, in Roever W.P., Langmaack H. and Pnueli A., Eds., *Compositionality: The Significant Difference*, LNCS 1536, Springer-Verlag, pp. 1-22, 1998.