PSL Model Checking and Run-Time Verification Via Testers

A. Pnueli and A. Zaks

New York University, New York
{amir, zaks}@cs.nyu.edu

Abstract. The paper introduces the construct of *temporal testers* as a compositional basis for the construction of automata corresponding to temporal formulas in the PSL logic. Temporal testers can be viewed as (non-deterministic) transducers that, at any point, output a boolean value which is 1 iff the corresponding temporal formula holds starting at the current position.

The main advantage of testers, compared to acceptors (such as Büchi automata) is that they are compositional. Namely, a tester for a compound formula can be constructed out of the testers for its sub-formulas. In this paper, we extend the application of the testers method from LTL to the logic PSL.

Besides providing the construction of testers for PSL, we indicate how the symbolic representation of the testers can be directly utilized for efficient model checking and run-time monitoring.

1 Introduction

The standard way of model checking an LTL property φ over a finite-state system S, represented by the automaton M_S, is based on the construction of an ω-automaton $\mathcal{A}_{\neg\varphi}$ that accepts all sequences that violate the property φ. Having both the system and its specification represented by automata, we may form the product automaton $M_S \times \mathcal{A}_{\neg\varphi}$ and check that it accepts the empty language, implying that there exists no computation of S which refutes φ [14].

Usually, the automaton $\mathcal{A}_{\neg\varphi}$ is a non-deterministic Büchi automaton, which is constructed using an explicit-state representation. In order to employ it in a symbolic (BDD-based) model checker, it is necessary to encode the automaton by the introduction of auxiliary variables. Another drawback of the normal (tableau-based) construction is that it is not compositional. That is, having constructed automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ for LTL formulas φ and ψ, there is no simple recipe for constructing the automaton for a compound formula which combines φ and ψ, such as $\varphi\ U\ \psi$.

The article [9] introduces a compositional approach to the construction of automata corresponding to LTL formulas. This construction is based on the notion of a *temporal tester* that has been introduced first in [8]. A tester for an LTL formula φ can be viewed as a *transducer* that keeps observing a state sequence σ and, at every position $j \geq 0$, outputs a boolean value which equals 1 iff $(\sigma, j) \models \varphi$. While acceptors, such as the Büchi automaton $\mathcal{A}_\varphi$, do not compose, transducers do. In Fig. 1, we show how transducers for the formulas φ, ψ, and $p\ U\ q$ can be composed into a transducer for the formula $\varphi\ U\ \psi$.

J. Misra, T. Nipkow, and E. Sekerinski (Eds.): FM 2006, LNCS 4085, pp. 573–586, 2006.

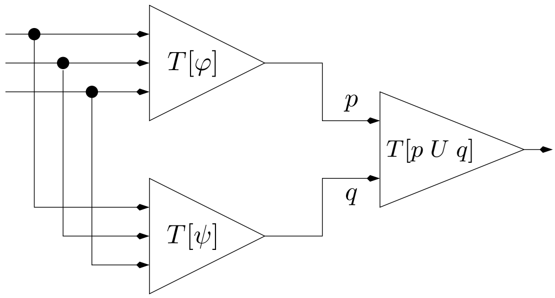

Fig. 1. Composition of transducers to form $T[\varphi\ U\ \psi]$

There are several important advantages to the use of temporal testers as the basis for the construction of automata for temporal formulas:

- The construction is compositional. Therefore, it is sufficient to specify testers for the basic temporal formulas: $X!p$ and $p\ U\ q$, where p and q are assertions (state formulas). Testers for more complex formulas can be derived by composition as in Fig. 1 .
- The testers for the basic formulas are naturally symbolic. Thus, a general tester, which is a synchronous parallel composition (automata product) of symbolic modules can also be easily represented symbolically.
- As shown below, the basic processes of model checking and run-time monitoring can be performed directly on the symbolic representation of the testers. There is no need for determinization or reduction to explicit state representation.

In the work presented in this paper, we generalize the temporal tester approach to the more expressive logic PSL, recently introduced as a new standard logic for specifying hardware properties [1]. Due to compositionality, it is only necessary to provide the construction of testers for the basic operators introduced by PSL.

In addition, we show how to construct an optimal symbolic run-time monitor. By optimality, we mean that the monitor extracts as much information as possible from the observed trace. In particular, an optimal monitor stops as soon it can be deduced that the specification is violated or satisfied, regardless of the possible continuations of the observed trace.

2 Accellera PSL

In this paper, we only consider a subset of PSL. For brevity, we omit the discussions of OBE (Optional Branching Extension) formulas that are based on CTL. Note that using testers we can obtain a model checking algorithm even for CTL* branching formulas by combining PSL testers with the work in [9]. Regarding run-time monitoring, which together with model checking is the primary motivation for our work, branching formulas are not applicable at all. In addition, we do not consider clocked formulas and formulas with *abort* operator. This is not a severe limitation since none of the above add any expressive power to PSL. One can find a rewriting scheme for the @ operator (clock operator) in [6] and for the *abort* operator in [12]. The rewriting rules produce a semantically equivalent formula not containing the operators, which is linear in the size of the original.

Due to lack of space, we do not formally define logic PSL but follow very closely the definitions from [6]. The only exceptions are the one mentioned above, and, for convenience, we define one additional operator $\langle r\rangle\varphi$ as

$$v \models \langle r\rangle\varphi \iff \exists j < |v| \text{ s.t. } \bar{v}^{0..j} \models\!\!\!\equiv r,\ v^{j..} \models \varphi.$$

3 Computational Model

3.1 Fair Discrete Systems with Finite Computations

We take a *just discrete system*(JDS), which is a variant of *fair transition system* [10], as our computational model. Under this model, a system $\mathcal{D} : \langle V, \Theta, R, \mathcal{J}, F\rangle$ consists of the following components:

- V: A finite set of *system variables*. A *state* of the system $\mathcal{D}$ provides a type-consistent interpretation of the system variables V. For a state s and a system variable $v \in V$, we denote the value assigned to v by the state s by $s[v]$.
- Θ: The *initial condition*. This is an assertion (state formula) characterizing the initial states. A state is defined to be *initial* if it satisfies Θ.
- $R(V, V')$: The *transition relation*, which is an assertion that relates the values of the variables in V interpreted by a state s to the values of the variables V' in an R*-successor* state s'.
- $\mathcal{J}$: A set of *justice* (*weak fairness*) requirements. Each justice requirement is an assertion. An infinite computation must include infinitely many states satisfying the assertion.
- F: The *termination condition*, which is an assertion specifying the set of *final* states. Each finite computation must end in a final state.

A *computation* of an JDS $\mathcal{D}$ is a non-empty sequence of states $\sigma : s_0, s_1, s_2, ...$, satisfying the requirements:

- *Initiality*: s_0 is initial.
- *Consecution*: For each $i \in [0, |\sigma|)$, the state s_{i+1} is a R-successor of state s_i. That is, $\langle s_i, s_{i+1}\rangle \in R(V, V')$ where, for each $v \in V$, we interpret v as $s_i[v]$ and v' as $s_{i+1}[v]$.
- *Justice*: If σ is infinite, then for every $J \in \mathcal{J}$, σ contains infinitely many occurrences of J-states.
- *Termination*: If $\sigma = s_0, s_1, s_2, ..., s_k$ is finite, then s_k must satisfy F.

A sequence of states $\sigma : s_0, s_1, s_2, ...$ that only satisfies all conditions for being a computation except initiality is called an *uninitialized computation*.

Given two JDS's, $\mathcal{D}_1$ and $\mathcal{D}_2$, their *synchronous parallel composition*, $\mathcal{D}_1 \ ||| \ \mathcal{D}_2$, is the JDS whose sets of variables and justice requirements are the unions of the corresponding sets in the two systems, whose initial and termination conditions are the conjunctions of the corresponding assertions, and whose transition relation is a conjunction of the two transition relations. Thus, a step in an execution of the composed system is a joint step of the systems $\mathcal{D}_1$ and $\mathcal{D}_2$.

3.2 Interpretation of PSL Formulas over JDS

We assume that the set of atomic propositions P is a subset of the variables V, so we can easily evaluate all the propositions at a given state of a JDS. We say that a letter $l \in 2^P$ *corresponds* to a state s if $p \in l$ iff $s[p] = 1$. Similarly, we define a correspondence between words and computations. We say, that a computation σ models (or satisfies) PSL formula φ, denoted $\sigma \models \varphi$, if the corresponding word v satisfies PSL formula φ.

4 Temporal Testers

One of the main problems in constructing a Büchi automaton for a PSL formula (or for that matter any ω-regular language) is that the conventional construction is not compositional. In particular, given Büchi automata $\mathcal{A}_\varphi$ and $\mathcal{A}_\psi$ for formulas φ and ψ, it is not trivial to build an automaton for $\varphi \ U \ \psi$. Compositionality is an important consideration, especially in the context of PSL. It is expected that specifications are written in a modular way, and PSL has several language constructs to facilitate that. For example, any property can be given a name, and a more complex property can be built by simply using a named sub-property instead of an atomic proposition.

One way to achieve compositionality with Büchi automata is to use alternation [3]. Nothing special is required from the Büchi automata to be composed in such manner, but the presence of universal branching in the resulting automaton is undesirable. Though most model checkers can deal with existential non-determinism directly and efficiently, universal branching is usually preprocessed at exponential cost.

Our approach is based on the observation that while the main property of Büchi automata (as well as any other automata) is to correctly identify a language membership of a given sequence of letters, starting from the very first letter; it turns out that for composition it is also very useful to know whether a word is in the language starting from an arbitrary position i. We refer to this new class of objects as *testers*. Essentially, testers are transducers that at each step output whether the suffix of the input sequence is in the language. Of course, the suffix is not known by the time the decision has to be made, so the testers are inherently non-deterministic.

Formally, a *tester* for a PSL formula φ is a JDS T_φ, which has a distinguished boolean variable x_φ, such that:

- **Soundness:** For every computation $\sigma : s_0, s_1, s_2, ...$ of T_φ , $s_i[x_\varphi] = 1$ iff $\sigma^{i..} \models \varphi$
- **Completeness:** For every sequence of states $\sigma' : s'_0, s'_1, s'_2, ...$, there is a matching computation $\sigma : s_0, s_1, s_2, ...$ such that for each i, s_i and s'_i agree on the interpretation of φ-variables.

Intuitively, the second condition requires that a tester must be able to correctly interpret x_φ for an arbitrary input sequence. Otherwise, the first condition can be trivially satisfied by a JDS that has no computations.

5 LTL Testers

We are going to continue the presentation of testers by considering two very important PSL operators, namely $X!$(next) and U(until). First, we show how to build testers for two *basic formulas* $X!b$ and $b_1\ U\ b_2$, where b, b_1, and b_2 are boolean expressions. Then, we demonstrate high compositionality of the testers by easily extending the result to cover full LTL. Note that our construction for LTL operators is very similar to the one presented in [8].

5.1 A Tester for $\varphi = X!b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. The components of T_φ are defined as follows:

$$T(X!b) : \begin{cases} V_\varphi : P \cup x_\varphi, \text{ where } P \text{ is a set of atomic propositions} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : x_\varphi = b' \\ \mathcal{J}_\varphi : \emptyset \\ F_\varphi : \neg x_\varphi \end{cases}$$

It almost immediately follows from the construction that $T(X!b)$ is indeed a good tester for $X!b$. The soundness of the $T(X!b)$ is guaranteed by the transition relation with the exception that we still have a freedom to incorrectly interpret x_φ at the very last state. This case is handled separately by insisting that every final state must interpret x_φ as false. The completeness follows from the fact that we do not restrict P variables by the transition relation, and we can always interpret x_φ properly, by either matching b' or setting it to false in the last state.

5.2 A Tester for $\varphi = b_1\ U\ b_2$

The components of T_φ are defined as follows:

$$T(b_1\ U\ b_2) : \begin{cases} V_\varphi : P \cup x_\varphi \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : x_\varphi = b_2 \vee (b_1 \wedge x'_\varphi) \\ \mathcal{J}_\varphi : b_2 \vee \neg x_\varphi \\ F_\varphi : b_2 \vee \neg x_\varphi \end{cases}$$

Unlike the previous tester, $T(b_1\ U\ b_2)$ has a non-empty justice set. A technical reason is that the transition relation allows x_φ to be continuously set to true without having a single state that actually satisfies b_2. The situation is ruled out by the justice requirement. Another way to look at the problem is that R_φ represents an expansion formula for the U(strong until) operator, namely $b_1\ U\ b_2 \iff b_2 \vee (b_1 \wedge X![b_1\ U\ b_2])$. In general, starting with an expansion formula is a good first step when building a tester. However, the expansion formula alone is usually not sufficient for a proper tester. Indeed, consider the operator $\mathcal{W}$(weak until), defined as $b_1\ \mathcal{W}\ b_2 \equiv \neg(\mathit{true}\ U\ \neg b_1) \vee b_1\ U\ b_2$, which has exactly the same expansion formula, namely $b_1\ \mathcal{W}\ b_2 \iff b_2 \vee (b_1 \wedge X![b_1\ \mathcal{W}\ b_2])$. We use justice to differentiate between the two operators.

6 Tester Composition

In Fig. 2, we present a recursive algorithm that builds a tester for an arbitrary LTL formula φ. In Example 1, we illustrate the algorithm by applying the tester construction for the formula $\varphi = \mathit{true}\ U\ \big(X![b_1\ U\ b_2] \vee (b_3\ U\ [b_1\ U\ b_2])\big)$.

- **Base Case**: If φ is a basic formula (i.e., $\varphi = X!b$ or $\varphi = b_1\ U\ b_2$), use construction from Section 5. For a trivial case, when the formula φ does not contain any temporal operators, we can use a tester for $\mathit{false}\ U\ \varphi$.
- **Induction Step**: Let ψ be an innermost basic sub-formula of φ, then $T_\varphi = T_{\varphi[\psi/x_\psi]}\ |||\ T_\psi$, where $\varphi[\psi/x_\psi]$ denotes the formula φ in which each occurrence of the sub-formula ψ is replaced with x_ψ.

Fig. 2. Tester construction for an arbitrary LTL formula φ

Example 1. *Tester Construction for* $\varphi = \mathit{true}\ U\ \big(X![b_1\ U\ b_2] \vee [b_1\ U\ b_2]\big)$

We start by identifying $b_1\ U\ b_2$ to be the innermost basic sub-formula and building the corresponding tester, $T_{b_1 U b_2}$ with the output variable y. Let $\alpha = \varphi[b_1\ U\ b_2/y]$; after the substitution $\alpha = \mathit{true}\ U\ \big(X!z \vee y\big)$. Note that we performed the substitution twice, but there is no need for two testers, which can result in significant savings. We proceed in similar fashion and build one more tester $T_{X!y}$ with the output variable x. After the substitutions, we obtain $\beta = \mathit{true}\ U\ [x \vee y]$, which satisfies the conditions of the base case. The final result can be expressed as:

$$T_\varphi = T_\beta\ |||\ T_{X!y}\ |||\ T_{b_1 U b_2}.$$

Though we have assumed φ is an LTL formula, the algorithm can be extended to PSL by considering additional basic formulas.

7 Associating a Regular Grammar with a SERE

Following [7], a grammar $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ consists of the following:

- $\mathcal{V}$: A finite set of *variables*.
- $\mathcal{T}$: A finite set of *terminals*. We assume that $\mathcal{V}$ and $\mathcal{T}$ are disjoint. In our framework, $\mathcal{T}$ consists of boolean expressions and a special terminal ϵ.
- $\mathcal{P}$: A finite set of *productions*. We only consider right-linear grammars, so all productions are of the form $V \to aW$ or $V \to a$, where a is a terminal, and V and W are variables.
- $\mathcal{S}$: A special variable called a *start symbol*.

We say a grammar $\mathcal{G}$ is *associated* with a SERE r if, intuitively, they both define the same language. While this definition is not accurate, we show a precise construction of an associated grammar for a given SERE in [12]. For example, we associate the following grammar $\mathcal{G}$ with SERE $r = (a_1 b_1)[*]$ && $(a_2 b_2)[*]$

$$\begin{aligned} V_1 &\to \epsilon \mid (a_1 \wedge a_2) V_2 \\ V_2 &\to \quad\; (b_1 \wedge b_2) V_1 \end{aligned}$$

Theorem 1. *For every SERE r of length n, there exists an associated grammar $\mathcal{G}$ with the number of productions $O(2^n)$. If we restrict SERE's to the three traditional operators: concatenation (;), union (|), and Kleene closure ([∗]), the number of productions becomes linear in the size of r.*

8 PSL Testers

There are only two additional basic formulas that we need to consider to handle full PSL, namely $\varphi = \langle r \rangle b$ and $\varphi = r$, where r is a SERE and b is a boolean expression. All other PSL temporal operators can be expressed using those two and the LTL operators, $X!$ and U. For example, $r! \equiv \langle r \rangle true$, and $r \mapsto b \equiv \neg(\langle r \rangle \neg\varphi)$.

8.1 A Tester for $\varphi = \langle r \rangle b$

Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S} \rangle$ be a grammar associated with r. Without the loss of generality, we assume $\mathcal{G}$ has variables $V_1, \ldots, V_n$ with V_1 being the start symbol. In addition, each variable V_i, has derivations of the form:

$$V_i \to \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n$$

where $\alpha_1, \ldots, \alpha_m, \beta_1, \ldots, \beta_n$ are boolean expressions. The case that variable V_i does not have a particular derivation $V_i \to \beta_j V_j$ or $V_i \to \alpha_k$, is covered by having $\beta_j = \mathit{false}$, and similarly, $\alpha_k = \mathit{false}$. Note that by insisting on this specific form, which does not allow ϵ productions, we can not express whether an empty string is in the language. However, since, by definition of $\langle\rangle$ operator, a prefix that satisfies r must be non-empty, we do not need to consider this. The tester T_φ is given by:

$$T_\varphi : \begin{cases} V_\varphi : P \cup x_\varphi \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\ \Theta_\varphi : 1 \\ R_\varphi : \text{Each derivation } V_i \to \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \qquad \text{contributes to } \rho \text{ the conjunct} \\ \qquad X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\ \qquad \text{and the conjunct} \\ \qquad Y_i \to (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \\ \qquad \text{the output variable is constrained by the conjunct} \\ \qquad x_\varphi = X_1 \\ \mathcal{J}_\varphi : \{\neg Y_1 \wedge \cdots \wedge \neg Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\ F_\varphi : \text{Each derivation } V_i \to \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \qquad \text{contributes to } F \text{ the conjunct} \\ \qquad X_i = (\alpha_1 \wedge b) \vee \cdots \vee (\alpha_m \wedge b) \end{cases}$$

Example 2. *A Tester for* $\varphi = \langle\{pq\}[*]\rangle b$.

To illustrate the construction, consider formula $\langle\{pq\}[*]\rangle b$. Following the algorithm from [12] and removing ϵ productions, the associated right-linear grammar for the SERE $\{pq\}[*]$ is given by

$$\begin{array}{l} V_1 \rightarrow \quad pV_2 \\ V_2 \rightarrow q \mid qV_1 \end{array}$$

Consequently, a tester for $\langle\{pq\}[*]\rangle b$ is given by

$$T(\langle\{pq\}[*]\rangle b) : \begin{cases} \quad V_\varphi : P \cup x_\varphi \cup \{X_1, X_2, Y_1, Y_2\} \\ \quad \Theta_\varphi : 1 \\ R_\varphi(V, V') : \begin{pmatrix} (X_1 = (p \wedge X_2')) & \wedge \\ (X_2 = (q \wedge b) \vee (q \wedge X_1')) & \wedge \\ (Y_1 \rightarrow (p \wedge Y_2')) & \wedge \\ (Y_2 \rightarrow (q \wedge b) \vee (q \wedge Y_1')) & \wedge \\ x_\varphi = X_1 & \end{pmatrix} \\ \quad \mathcal{J}_\varphi : \{\neg Y_1 \wedge \neg Y_2, \quad X_1 = Y_1 \wedge X_2 = Y_2\} \\ \quad F_\varphi : (X_1 = \mathit{false}) \wedge (X_2 = q \wedge b) \end{cases}$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix satisfying the SERE r. Thus, the subsequence $s_j, \ldots, s_k, \ldots \models \langle r\rangle b$ iff there exists a generation sequence $V^j = V_1, V^{j+1}, \ldots, V^k$, such that for each $i, j \leq i < k$, there exists a grammar rule $V^i \rightarrow \beta V^{i+1}$, where $s_i \Vert\!\models \beta, V^k \rightarrow \alpha$, and $s_k \Vert\!\models (\alpha \wedge b)$.

The generation sequence is represented in a run of the tester by a sequence of true valuations for the variables $Z^j = Z_1, Z^{j+1}, \ldots, Z^k$ where $Z^i \in \{X^i, Y^i\}$ for each $i \in [j..k]$. An important element in this checking is to make sure that any such generation sequence is finite. This is accomplished through the double representation of each V_i by X_i and Y_i. The justice requirement $(X_1 = Y_1) \wedge \cdots \wedge (X_n = Y_n)$ guarantees that that any true X_i is eventually copied into Y_i. The justice requirement $\neg Y_1 \wedge \cdots \wedge \neg Y_n$ guarantees that all true Y_i's are eventually falsified. Together, they guarantee that there exists no infinite generation sequence. The double representation approach was introduced in [11].

8.2 A Tester for $\varphi = r$

We start the construction exactly the same way as we did for $\varphi = \langle r\rangle b$, in Section 8.1. Let $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi\rangle$ be the tester we wish to construct. Assume that x_φ is the output variable. Let $\mathcal{G} = \langle \mathcal{V}, \mathcal{T}, \mathcal{P}, \mathcal{S}\rangle$ be a grammar associated with r.

The tester T_φ is given by:

$$T_\varphi : \begin{cases} V_\varphi : P \cup x_\varphi \cup \{X_1, \ldots, X_n, Y_1, \ldots, Y_n\} \\ \Theta_\varphi : 1 \\ R_\varphi(V, V') : \text{Each derivation } V_i \to \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \quad \text{contributes to } \rho \text{ the conjunct} \\ \quad X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge X_1') \vee \cdots \vee (\beta_n \wedge X_n') \\ \quad \text{and the conjunct} \\ \quad \alpha_1 \vee \cdots \vee \alpha_m \vee (\beta_1 \wedge Y_1') \vee \cdots \vee (\beta_n \wedge Y_n') \to Y_i \\ \quad \text{the output variable is constrained by the conjunct} \\ \quad x_\varphi = X_1 \\ \mathcal{J}_\varphi : \{Y_1 \wedge \cdots \wedge Y_n, \quad X_1 = Y_1 \wedge \cdots \wedge X_n = Y_n\} \\ F_\varphi : \text{Each derivation } V_i \to \alpha_1 \mid \cdots \mid \alpha_m \mid \beta_1 V_1 \mid \cdots \mid \beta_n V_n \\ \quad \text{contributes to } F \text{ the conjunct} \\ \quad X_i = \alpha_1 \vee \cdots \vee \alpha_m \vee \beta_1 \vee \cdots \vee \beta_n \end{cases}$$

The variables $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ are expected to check that the rest of the sequence from now on has a prefix that does not violate SERE r. We follow a similar approach as for the tester $\varphi = \langle r \rangle b$. However, now we are more concerned with false values of the variables $X_1 \ldots X_n$. The duality comes from the fact that, now, we are trying to prevent postponing the violation of the formula r forever.

8.3 Complexity of the Construction

Theorem 2. *For every* PSL *formula φ of length n, there exists a tester with $O(2^n)$ variables. If we restrict SERE's to three traditional operators: concatenation* (;)*, union* (|)*, and Kleene closure* ([*])*, the number of variables is linear in the size of φ.*

To justify the result, we can just count the fresh variables introduced at each step of the tester construction. There is only linear number of sub-formulas, so there is a linear number of output variables. The only other variables introduced are the ones that are used to handle SERE's. According to Theorem 1, the associated grammars contain at most $O(2^n)$ non-terminals ($O(n)$ - for the restricted case). We conclude by observing that testers for the formulas $\varphi = \langle r \rangle b$ and $\varphi = r$ introduce exactly two variables, X_i and Y_i, for each non-terminal V_i.

9 Using Testers for Model Checking

One of the main advantages of our construction is that all the steps, as well as the final result – the tester itself, can be represented symbolically. That is particularly handy if one is to use symbolic model checking [2]. Assume that the formula under consideration is φ, and $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi \rangle$ is the corresponding tester. Let JDS $\mathcal{D}$ represent the system we wish to model check.

We are going to use traditional automata theoretic approach based on synchronous composition, as in [2]. We perform the following steps:

- Compose $\mathcal{D}$ with T_φ to obtain $\mathcal{D} \mid\mid\mid T_\varphi$.
- Check if $\mathcal{D} \mid\mid\mid T_\varphi$ has a (fair) computation, such that $s_0[x_\varphi] = 0$. $\mathcal{D} \mid\mid\mid T_\varphi$ has such a computation iff $\mathcal{D}$ does not satisfy φ.

As you can see, a tester can be used anywhere instead of an automaton. Indeed, we can always obtain an automaton from a tester by restricting the initial state to interpret x_φ as $true$.

10 Run-Time Monitoring with Testers

The problem of *run-time monitoring* can be described as follows. Assume a reactive system $\mathcal{D}$ and a PSL formula φ, which formalizes a property that $\mathcal{D}$ should satisfy. In order to test the conjecture that $\mathcal{D}$ satisfies φ, we construct a program M, to which we refer as a *monitor*, that observes individual behaviors of $\mathcal{D}$. Behaviors of $\mathcal{D}$ are fed to the monitor state by state. After observing the finite sequence $\sigma : s_0, \ldots, s_k$ for some $k \geq 0$, we expect the monitor to be able to answer a subset of the following questions:

1. Does σ satisfy the formula φ?
2. Is φ *negatively determined* by σ? That is, is it the case that $\sigma \cdot \eta \not\models \varphi$ for all finite or infinite completions η.
3. Is φ *positively determined* by σ? That is, is it the case that $\sigma \cdot \eta \models \varphi$ for all finite or infinite completions η?
4. Is φ $\sigma-$*monitorable*? That is, is it the case that there exists a finite η such that φ is positively or negatively determined by $\sigma \cdot \eta$. If $\mathcal{D}$ is expected to run forever then it is useless to continue monitoring after observing σ such that φ is not $\sigma-$*monitorable*.

Solving the above questions leads to a creation of an *optimal* monitor - a monitor that extracts as much information as possible from the observation σ. In particular, an optimal monitor detects a violation of the property as early as possible. Of course, a monitor can do better if we supply it with some implementation details of the system $\mathcal{D}$, which may allow to deduce a violation even earlier [13]. In the extreme case, when a monitor knows everything about $\mathcal{D}$ the monitoring problem is reduced to model checking.

10.1 Monitoring with Testers

Let $\mathcal{D} : \langle P, \Theta, R, \mathcal{J}, F\rangle$ be a reactive system with observable variables P, and let φ be a PSL formula over P, which validity with respect to $\mathcal{D}$ we wish to test. Assume that $T_\varphi = \langle V_\varphi, \Theta_\varphi, R_\varphi, \mathcal{J}_\varphi, F_\varphi\rangle$ is the tester for φ, where the variables $V_\varphi = P \cup A$ are partitioned into the variables of $\mathcal{D}$ and additional auxiliary variables A. Let x_φ be the distinguished output variable of the tester T.

For an assertion (state formula) α, we define the R_φ-*predecessor* and R_φ-*successor* of α by

- $R_\varphi \diamond \alpha \;=\; \exists V'_\varphi : R_\varphi(V_\varphi, V'_\varphi) \wedge \alpha'$
- $\alpha \diamond R_\varphi \;=\; \textit{unprime}(\exists V_\varphi : R_\varphi(V_\varphi, V'_\varphi) \wedge \alpha)$

where $unprime$ simply replaces all next state variables with current state variable. Remember that the transition relation R_φ has two copies of each variable, one representing a current state and the other copy (a primed one) the next state.

Let $\sigma : s_0, s_1, \ldots, s_k$ be a finite observation produced by system $\mathcal{D}$. That is, a sequence of evaluations of the variables P. We define the *symbolic monitoring trace* $\mathcal{M} = \alpha_0, \alpha_1, \ldots, \alpha_k$ as the sequence of assertions given by

$$\alpha_0 = \Theta_\varphi \wedge x_\varphi \wedge (P{=}s_0), \text{ and } \alpha_{i+1} = (\alpha_i \diamond R_\varphi) \wedge (P{=}s_{i+1}),\ i \in [0, k),$$

where $P = s$ stands for $\bigwedge\limits_{v \in P} v = s[v]$.

Essentially, α_i represents a "current" state of the monitor, which is more precisely just a set of states of the tester T_φ. Whenever, the system makes a step from s_i to s_{i+1}, a monitor takes the corresponding step from α_i to α_{i+1} according to the transition relation R_φ and the interpretation of the propositions by the state s_{i+1}. The whole process can be described as, on the fly, synchronous, composition of the system and the tester, in which the later is determinized using classical subset construction. Note that we only need to worry about the existential non-determinism, A similar approach, but for alternating automata was also used for a so called breadth-first traversal in [**?**]. The monitoring sequence can be used to answer the first of the monitoring questions as stated by the following claim:

Claim 1 (Finitary satisfaction). *For a* PSL *formula φ, the finite sequence $\sigma : s_0, s_1, \ldots, s_k$ satisfies φ, i.e., $\sigma \models \varphi$, iff the formula $\alpha_k \ \wedge\ F_\varphi$ is satisfiable.*

The correctness of the claim results from the following observations. The tester T_φ can be interpreted as a non-deterministic automaton for acceptance of sequences satisfying φ if we insist that x_φ is $true$ in the initial state. Furthermore, the assertion α_k represents all the automaton (tester) states which can be reached after reading the input σ. If any such evaluation is consistent with the assertion F_φ, which represents the set of final states, then this points to an accepting run of the automaton.

10.2 Deciding Negative Determinacy

Claim 1 has settled the first monitoring task. Next we consider one of the remaining tasks. Namely, we show how to decide whether, for a given σ, $\sigma \cdot \eta \not\models \varphi$ for all infinite or finite completions η.

In order to do this, we have to perform some offline calculations as a preparation. We generalize the notion of a single-step predecessor to an *eventual* predecessor by defining

$$R^*_\varphi \diamond \alpha = \alpha \vee R_\varphi \diamond \alpha \vee R_\varphi \diamond (R_\varphi \diamond \alpha) \vee \cdots$$

Consider the fix-point expression presented in Equation (1).

$$\mathit{feas} = [\mu X : (R_\varphi \diamond X) \vee F_\varphi] \quad \bigvee \quad [\nu Y : R_\varphi \diamond Y \ \wedge \bigwedge_{J \in \mathcal{J}} R^*_\varphi \diamond (Y \wedge J_\varphi)] \qquad (1)$$

The first expression captures all the states that have a path to a final state. The second expression captures a maximal set of tester states Y such that every non-final state $s \in Y$ has an Y-successor and, for every justice requirement J, s has a Y-path leading to some Y-state which also satisfies J. The following can be proven:

Claim 2 (Feasible states). *The set feas characterizes the set of all states which originate an uninitialized computation.*

Assuming that we have precomputed the assertion *feas*, the following claim tells us how to decide whether a finite observation σ is sufficient in order to negatively determine φ:

Claim 3 (Negative Determinacy). *The* PSL *formula φ is negatively determined by the finite observation $\sigma = s_0, s_1, \ldots, s_k$ iff $\alpha_k \wedge$ feas is unsatisfiable.*

The claim is justified by the observation that $\alpha_k \wedge$ *feas* being unsatisfiable means that there is no way to complete the finite observation σ into a finite or infinite observation which will satisfy φ.

10.3 Deciding Positive Determinacy

In order to decide positive determinacy, we need to monitor the incoming observations not only by assertion sequences which attempt to validate φ but also by an assertion sequence which attempts to refute φ. Consequently, we define the *negative symbolic monitoring trace* $\mathcal{M}^- = \beta_0, \beta_1, \ldots, \beta_k$ by

$$\beta_0 = \Theta_\varphi \wedge \neg x_\varphi \wedge (P{=}s_0), \text{ and } \beta_{i+1} = (\beta_i \diamond R_\varphi) \wedge (P{=}s_{i+1}),\ i \in [0, k)$$

Claim 4 (Positive Determinacy). *The* PSL *formula φ is positively determined by the finite observation $\sigma = s_0, s_1, \ldots, s_k$ iff $\beta_k \wedge$ feas is unsatisfiable.*

10.4 Detecting Non-monitorable Prefixes

Unfortunately, not all properties can be effectively monitored. Consider a property $\square \diamondsuit p$, which is not σ-monitorable for any σ prefix. No useful information can be gained after observing a finite prefix if the property only depends on the things that must happen infinitely often. A good monitor should be able to detect such situations and alert the user. Next, we show how to decide whether φ is σ-monitorable, for a given σ.

Let $\mathcal{M} = \alpha_0, \alpha_1, \ldots, \alpha_k$ and $\mathcal{M}^- = \beta_0, \beta_1, \ldots, \beta_k$ be the positive and negative symbolic monitoring traces that correspond to σ. Let Γ represent a set of assertions. We define the *R_φ-successor* and *eventual R_φ-successor* of Γ by

- $\Gamma \diamondsuit R_\varphi \ = \ \{(\gamma \diamond R_\varphi) \wedge (P = s) \mid \gamma \in \Gamma, s$ is some state of the system $\mathcal{D}\}$
- $\Gamma \diamondsuit R^*_\varphi = \Gamma \vee R_\varphi \diamondsuit \Gamma \vee R_\varphi \diamondsuit (R_\varphi \diamondsuit \Gamma) \vee \cdots$

Claim 5 (Monitorability). *A* PSL *formula φ is $\sigma-$monitorable, where $\sigma{=}s_0, s_1, \ldots,$ s_k, iff there exists an assertion γ such that either $\gamma \in (\alpha_k \diamondsuit R^*_\varphi)$ or $\gamma \in (\beta_k \diamondsuit R^*_\varphi)$, and $(\gamma \wedge$ feas$)$ is unsatisfiable.*

The claim almost immediately follows from the definition of $\sigma-$monitorable properties, Claim 3, and Claim 4. Note that the algorithm can be very inefficient due to the double-exponential complexity. One way to cope with the problem is to consider each state in α_k and β_k individually. The idea is very similar to never-violate states introduced in [5]. A state of a Büchi automaton is called *never violate* if, on any input letter, there is a transition to another *never-violate* state. Similarly, we can define *never-satisfy* states and obtain a reasonable approximation to the problem of monitorability. Note that the complexity of this solution is exponential, which hopefully can be managed using BDD's. In addition, the never-violate and never-satisfy states can be pre-computed before the monitoring starts. However, it remains to be seen whether the approximation works well in practice.

11 Related Work

It is very interesting to compare our approach to the one suggested in [4], which uses alternating automata. We have already mentioned some high-level distinctions between testers and alternating automata in Section 4. However, the question remains about which construction is better. It turns out that both approaches yield very similar results, assuming universal non-determinism is removed from the alternating automata. Although that is a somewhat unexpected conclusion, it is not hard to justify it.

Without going into the details of algorithm described in [4], it is enough to mention that each state in the alternating automaton is essentially labeled with a sub-formula. To remove universal non-determinism, we follow classical subset construction. In particular, we assign a boolean variable x for each sub-formula φ to represent whether the corresponding state is in the subset. One can easily verify that x is nothing more but the output variable of the tester T_φ and follows the same transition relation.

To finish the partial determinization and define the final states in the new automata, the authors of [4] use the same trick with double representation as we do. At this step, the automata obtained after the subset construction is composed with itself via a cartesian product. This step is conceptually the same as introducing Y variables in the tester construction. However, we only introduce the extra variables when dealing with SERE's. For the LTL portion of the formula, the tester construction avoids the quadratic blow out associated with the cartesian product by essentially building a generalized Büchi with multiple acceptance sets (i.e., multiple justice requirements). If one to insist on a single acceptance set, our approach would yield an automaton identical to the one obtained in [4]. Note that, for symbolic model checking, using a generalized Büchi automaton might be more efficient then the corresponding Büchi automaton.

While our approach may not necessarily yield a better automaton, it never performs worse, and there are several significant benefits. Since model checking is very expensive, we expect that, in practice, automata for commonly occurring sub-properties will be hand-tuned. In such a case, it is more beneficial to work with testers since an alternating automaton requires an exponential blow-up due to universal non-determinism that cannot be locally optimized.

Another important advantage is that PSL testers can be used anywhere instead of LTL testers. For example, if one were to extend CTL* with PSL operators, our approach combined with [9] immediately gives a model checking algorithm for the new logic.

12 Conclusion

In this paper, we have shown a new approach towards model checking logic PSL, recently introduced as a new standard for specifying hardware properties. Our approach is based on testers that, unlike automata, are highly compositional, which is very advantageous in the context of PSL.

In addition, we have described a framework for symbolic run-time monitoring. In particular, we have identified some of the major questions that a good monitor should be able to answer and shown how to answer those questions using symbolic algorithms.

Acknowledgement

The authors wish to thank the anonymous reviewers as well as Cindy Eisner for their helpful comments.

References

1. Accellera Organization, Inc. *Property Specification Language Reference Manual, Version 1.01*, 2003. http://www.accellera.org/.
2. E.M. Clarke andO. Grumberg and D.A. Peled. *Model checking*. MIT Press, 2000.
3. A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of ACM*, 28(1):114–133, 1981.
4. Bustan D., Fisman D., and Havlicek J. Automata Construction for PSL. 2005. http://www.wisdom.weizmann.ac.il/d̃ana/publicat/automta_constructionTR.pdf.
5. Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In *CAV*, pages 364–378, 2005.
6. Cindy Eisner, Dana Fisman, John Havlicek, Michael Gordon, Anthony McIsaac, and David Van Campenhout. Formal Syntax and Semantics of PSL. 2003. http://www.wisdom.weizmann.ac.il/d̃ana/publicat/formal_semantics_standalone.pdf.
7. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Reading, Massachussetts, 1979.
8. Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloq. Aut. Lang. Prog.*, volume 1443 of *Lect. Notes in Comp. Sci.*, pages 1–16, 1998.
9. Yonit Kesten and Amir Pnueli. A compositional approach to CTL* verification. *Theoretical Computer Science*, 331:397–428, 2005.
10. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
11. Satoru Miyano and Takeshi Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.
12. Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. Technical Report, Dept. of Computer Science, New York University, 2006.
13. Amir Pnueli, Aleksandr Zaks, and Lenore Zuck. Monitoring interfaces for faults. In H. Barringer, B. Finkbeiner, Y. Gurevich, and H. Sipma, editors, *Fifth International Workshop on Run-time Verification (RV)*, July 2005. Edinburgh, Scotland, UK.
14. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First IEEE Symp. Logic in Comp. Sci.*, pages 332–344, 1986.

Formal Methods for Security: Lightweight Plug-In or New Engineering Discipline

Werner Stephan

German Research Center for Artificial Intelligence (DFKI GmbH)
Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
Werner.Stephan@dfki.de

Abstract. This contribution discusses two main lines of developments concerning the use of formal methods in security engineering. Fully automated and highly specialized methods that hide most of the formal theory from its users are compared to formal security models centered around explicit formal system models. It is argued that only the latter offer the perspective to comprehensively control the development process with its various security aspects and phases. In putting more emphasis on the combination of theories, fragmentation could be overcome by an integration of the specialized methods that are presently still applied in isolation.

1 Introduction

For most of the questions concerning the industrial perspective of formal techniques including the topics addressed at this I-Day there will be no simple answers. The current situation in formal methods is characterized by a vast amount of approaches most often pursued in isolation and difficult to overlook even by experts. To assess obstacles, potential benefits, and costs for an industrial use, first of all it is therefore necessary to roughly classify the main lines of development.

In the author's view, approaches like the AbsInt tool for execution time analysis [2] and the ongoing Boogie development [6], are very likely to be adopted on a broad basis by industrial users. There is no reason not to believe that in security engineering program analysis to detect covert channels, following approaches developed in [1] and specialized tools for analyzing security protocols, like the AVISPA tool [3] are on the right track toward commercial applications.

Such approaches *hide* a deep and complex theory and highly sophisticated implementation[1] behind a front end that seamlessly fits to existing, well established programming environments and notations. They are easy to use since the formal analysis is carried out *automatically* with results that can be interpreted directly in the given context. Hiding the internal structure of the underlying model also provides protection against *inadequate usage*. Finally, the *extra costs* for the acquisition and application of these tools make no real difference, neglecting the expenses for research that was necessary to develop them.

[1] From this point of view to call them "‘lightweight plugins"’ is much too disrespectful.

J. Misra, T. Nipkow, and E. Sekerinski (Eds.): FM 2006, LNCS 4085, pp. 587–591, 2006.

However, even if support for error prone and tedious activities is offered at a high level of expertise beyond that of typical human experts, and even if the approaches scale up to the size of real world applications, their isolated use will limit the impact of formal methods on the overall development process. Fragmentation seems to be a consequence of the advantages mentioned above.

The strength in automatic problem solving comes along with restrictions to certain aspects, like protocol correctness, and development phases, like coding. Hiding the underlying formal models and theory makes it difficult to adapt these methods to new scenarios and, above all, *combine* them into an integrated approach necessary for a comprehensive treatment of complex developments. A multi-applicative smart card, for example, might use access control to protect the applications (against each other), protocols to communicate with the outside world, in particular to download new applications, and non-generic but security critical mechanisms inside the applications.

What seems to be missing is an explicit, sufficiently rich system model at various levels of abstraction that is shared by highly specialized analysis methods for generic mechanisms like access control and cryptographic protocols.

The author is not completely confident that we will see formal methods used to *control*, *record*, and *assess* comprehensive developments on a scientifically objectified basis in the near future. Nevertheless, after arguing that the program indicated above seems to suit the spirit of the Common Criteria framework, some encouraging own experiences will be mentioned as a basis for a brief discussion of steps necessary to keep this vision on the agenda.

2 Formal Methods in the Common Criteria

To a large extent the engineering practice for critical systems will (and in the opinion of the author has to be) shaped by mandatory guidelines like the Common Criteria (CC) for IT security. Formal methods will never leave the state of an unsystematic, accidental, and largely incompatible use without such frameworks. Most of the (commercial) formal activities of the DFKI group related to IT security were part of developments intended to meet the CC requirements.

Although not stipulating a particular development method or life cycle model, the CC insists on laying down requirements and tracking them through appropriately documented design stages connected by well defined relationships.

In our work we adopt the view that the *formal* Security Model (as required by the CC) should consist of an (abstract) specification of the relevant parts of the system together with certain security mechanisms (or measures). Although this is not explicitly requested we then proved that the intended security properties (or principles) are actually satisfied. This is the place where the specific techniques for modeling and proving security properties, like protocol analysis [3,7] and information flow analysis at the specification level [5], come into play. In this context specialized automated techniques, if integrated into the overall formal model, will lower the burden of proof work. However, in this setting one has to be prepared to establish a formal relationship to the system model and to possibly integrate several security issues.

To allow for such an integration an explicit formal system model is needed. This has to be given by the developer using a specification language which is expressive and general enough to cover all necessary aspects. In most cases this means that in the general case we have to carry out interactive proofs.

Whether or not one day automatic techniques used *in this setting* will reduce the interactive proof work to some residue that can be neglected remains an open question.

The formal model also serves as a starting point for a subsequent refinement to technical solutions, for example on platforms like smart cards. This includes the relationship to non-formal technical documents. In cases where the specification technique did not guarantee (logical) consistency, refinements were used for model construction by prototypic implementations.

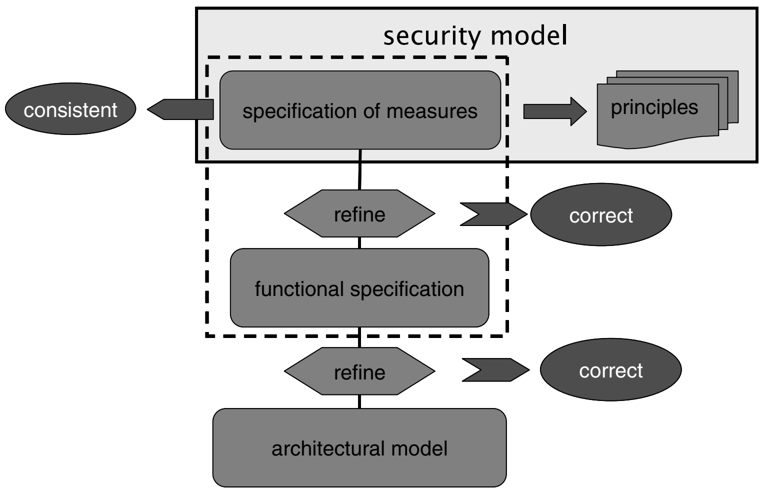

Fig. 1. CC-Components for Formal Development

During the design stage formal debugging techniques are useful since interactive proofs still are very sensible to changes despite reasonable progress in (proof) reuse.

Yet it should be stated that the use of formal methods advocated here can be ”trivialized” (intendedly or unintendedly). This is due to the open nature of formal modeling which is difficult to restrict by guidelines or official CC interpretations. For example, a pure restatement of, say, some access control rules without relation to a system model and its refinement or the proof of information flow properties realized by them will not offer much benefit except that the formal[2] requirements of the CC are (possibly) met. Such a use of formal techniques will sooner or later discredit the whole community.

[2] In the legal sense.

3 Experience

The group at DFKI developed several security models as part of industrial consortia following the scheme above. Specifications and proofs were carried out in the Verification Support Environment (VSE) [4].

In all cases the formal model covered the critical part of the system under consideration. Moreover, the models contained all details and cases mentioned in the non-formal design documents.

In particular, when discussing the possible benefits of formal methods, this working use of formal methods should well be distinguished from the idealized and simplified models often appearing in case studies performed to *demonstrate* the principal theory behind some method.

The formal specifications were readable and understandable by most of the other (technical) team members after they had been briefly introduced to VSE. However, this does not hold for the theory used to formulate and prove the desired security properties. Whenever the formal reflection (using that theory) revealed problems they were discussed using the formal specification as a reference. The majority of problems was detected (by our experts) while writing down the specification. However, due to the complexity of the system without the proofs no one felt really confident that there were no *further* vulnerabilities left.

Again note that it is not enough to show that some solution is secure *in principle*. This is sufficient for mathematical results discussed in the scientific community. Here we use formal methods to control developments in all their detail using tools that guarantee sound reasoning.

The formal work was carried out as part of the ongoing project including revisions. It caused no critical delays. Redoing proofs turned out to be critical. When in some cases we monitored our work it turned out that up to 50% of the proof work resulted from (our own) specification errors and revisions.

The use of formal methods as outlined above increased the costs considerably. However, due to our estimates the additional costs were comparable to those of other expert teams.

Taking all this as an indication that formal modeling and analysis along the lines briefly described above is not just a mere vision, what are the necessary steps on the road to a comprehensive use of formal methods in security engineering?

- All of our developments were basically built from scratch. Except for the basic modeling techniques (abstract data types, state based systems, concurrency, information flow, protocol traces) we could not build on any formal (security) engineering experiences manifested in guidelines or even generic models (or parts thereof). To develop such patterns for formal modeling on top of the existing theories and to extend these where necessary from an application oriented point of view seems to be the most important task.
- In particular, notions of refinement for security mechanisms (analyzed in a formal way) have to be developed.
- Combination of theories is another critical issue.
- In the academic community there seems not to be enough appreciation for the kind of work indicated above.

– On the other hand collaboration with industrial partners (for example in application oriented research projects) is absolutely necessary to obtain the appropriate domain knowledge.
– The academic community should take part in and (try to) influence the further development of criteria like the CC.

References

1. Andrei Sabelfeld and David Sands. A Per Model of Secure Information Flow in Sequential Programs. In *European Symposium on Programming*, pages 40–58, 1999
2. AbsInt: Advanced Compiler Technology for Embedded Systems. http://www.absint.com
3. The AVISPA project. http://www.avispa-project.org/
4. Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, Werner Stephan, and Andreas Wolpers. Verification support environment (vse). *High Integrity Systems*, 1(6): 523–530, 1996.
5. Heiko Mantel. Information Flow Control and Applications – Bridging a Gap. In Jose Nuno Olivera and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, pages 153–172, LNCS 2021, Springer, Berlin, Germany, 2001
6. Leino, K. R. M. and Müller, P. Modular verification of static class invariants. In Fitzgerald, J. and Hayes, I. and Tarlecki, A., editors, *Formal Methods (FM)*, pages 26 – 42, LNCS 3582, Springer-Verlag, 2005
7. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.