Temporal Logic and Toolset for Applet Verification:

Compositional Reasoning, Model Checking, Abstract Interpretation.

VeriFiCard Deliverable 4.1.

Gilles Barthe, Pierre Courtieu, Guillaume Dufay,
Marieke Huisman, Simão Melo de Sousa[1],
Gennady Chugunov, Lars-Åke Fredlund, and Dilian Gurov[2]

[1] INRIA Sophia-Antipolis, France
[2] Swedish Institute of Computer Science

1 Introduction

One of the VerifiCard objectives is to provide methods for the specification and verification of smart card applets, and tools to support these. Within WP4, we have explored algorithmic techniques for verifying JavaCard applets. Two families of properties have been identified and studied:

- generic safety properties such as type safety, non-interference, object initialisation, etc. Such properties are best enforced by type systems and thus the preferred techniques for the verification of such properties are enhanced bytecode verifiers. Most of our effort in this direction has consisted in developing adequate tools to provide effective support for designing and validating novel type systems for the JavaCard Virtual Machine. Preliminary experiments suggest that our tools shall reduce substantially the cost of specifying and verifying type systems for JavaCard (as an indication, validating an existing type system for JavaCard in WP3 represents an effort of approximately 100 man*monthes), and the level of expertise required for carrying such formal specifications and verifications.
- application-specific properties which deal with the behavior of programs. These include safety properties such as "Applet A can only perform an operation after having been successfully authenticated by Applet B", but also liveness properties. Such properties are best expressed in a temporal logic and verified using techniques of model-checking, abstract interpretation and compositional reasoning. Most of our effort in this direction has consisted in integrating existing tools into an effective toolset for verifying temporal properties of open smartcard systems. A distinctive feature of our toolset is its ability to deal with open systems. Indeed, by reducing global properties

of the global system to local properties of components, verifications on the global system can be performed on the basis of the specifications (and not implementations) of its components.

All the work in this workpackage could have been done at either byte or source code level. We chose to do it at bytecode level, because this is what any industrial user will prefer, as bytecode level is closest to actual execution of an applet on a physical smart card.

This document describes the principles, methods and tools used for the verification of control-flow properties (Task WP4.1) and of (generic) safety properties (Task WP4.2).

2 Verification of Control Flow Properties

2.1 Summary of Results

Within Task WP 4.1 the following has been produced:

- a compositional program model to represent JavaCard programs [5, 4], together with results justifying the compositional operational semantics [4];
- a specification logic to express properties about the compositional program model [4];
- a proof system for verifying property decompositions [6];
- an embedding of this proof system into the EVT framework, see `http:/www.sics.se/fdt/vericode/vcpt.html`;
- a toolset for model checking individual applets, see [12] and `http://www.sics.se/fdt/vericode/jcave.html`.

2.2 Motivation

Concerning applet verification, we have focused on the verification of behavioural properties of applet interactions, describing the required control flow of the program. Within this section, we briefly describe the different ingredients of this approach – an abstract program model, a specification language and appropriate verification techniques – and give references for further information.

Program model First of all, we propose an abstract program model for JavaCard programs. This program model abstracts away all data and focuses on the control flow of the program. A program is represented as the union of a set of applets (components), and applets are represented by an internal transfer graph and a call graph. The granularity of the decomposition can be varied, according to needs: it can vary from a single method to the whole system. We actually prove that the composition of two components can also be considered as a single component. The program model is developed to be general, so that it can be applied for different programming languages. Its instantiation to JavaCard programs is straightforward.

Specification language Secondly, we have developed a set of specification patterns which can be used to specify relevant behavioural properties about JavaCard applets. The specification patterns have a semantics in the modal μ-calculus [18]. Since many behavioural properties of applet interaction can be reduced to syntactic (that is, structural) properties of the program graphs which can be checked far more efficiently by simple inspection of the transfer and call graphs, we also propose a specification language for syntactic properties.

Verification techniques Finally, for the verification of the specifications *w.r.t* the program model, we identify two verification tasks:

1. formulate a global specification and prove that the composed system satisfies this global specification, relative to the component specifications; and
2. verify that each component implementation satisfies its specification.

Ad. 1, it is important that this is done completely at specification level: it is not necessary to have a (model of an) implementation to verify that the decomposition is correct. We developed appropriate proof rules to show the correctness of the composition. The correctness proofs for composition amount to showing the correctness of the following sequent (where ϕ, ψ and θ are specifications, x and y are components of the system and $x\,|\,y$ denotes their composition).

$$x : \phi, y : \psi \vdash x\,|\,y : \theta$$

Ad. 2, we show how this can be done by casting the program model into a push down automaton, and by using existing model checkers for push down automata.

2.3 Program model

Our initial proposal and motivation for the program model has been presented in [5]. The program model has been inspired by the work of Jensen *et al.* [17] on verification of security properties for monolithic systems. However, our first proposal was too abstract, and we improved it by keeping the notion of method. The complete formal description of the framework can be found in [4]; here we present the main ideas.

Definition 1 (Program, Applet, Method Graph). *A* program *consists of a set of applets $\mathcal{A}$ over a set of methods M. An* applet *$a \in \mathcal{A}$ is a nonempty subset of M. A* method graph *is a tuple*

$$m \triangleq (V_m, \rightarrow_m, \lambda_m, \mu_m)$$

such that:

(i) V_m *is the finite set of* program points *of* m*,*
(ii) $\rightarrow_m \subseteq V_m \times V_m$ *are the* transfer edges *of* m*, and*

(iii) $\lambda_m : V_m \to T$ *designates to each program point of* m *a* program point type *from the set* $T \stackrel{\Delta}{=} \{\mathsf{entry}, \mathsf{seq}, \mathsf{call}, \mathsf{return}\}$.

(iv) $\mu_m : V_m \to \wp(M)$ *designates to each program point of type* call *of* m *a non-empty set of methods.*

The behaviour of a program is described by a labelled transition system (LTS). To define this, we first define the notion of applet state.

Definition 2 (Applet State). *A state of an applet* a *is a pair* $\langle c, \sigma \rangle$ *consisting of a* control point $c \in V_a \cup \{\star\}$ *and an* applet stack $\sigma \in (V_a \cup (M - a))^\star$. *The state* $\langle \star, \epsilon \rangle$ *is the* initial state *of* a.

Applets induce LTSs through a set of applet transition rules.

Definition 3 (Induced LTS). *An applet* a induces *an LTS*

$$\mathcal{T}_a \stackrel{\Delta}{=} (\mathcal{S}_a, \mathcal{L}_a, \to_a)$$

as follows:

(i) $\mathcal{S}_a$ *is the set of states of applet* a,

(ii) $\mathcal{L}_a$ *consists of the* silent action τ *and the set of triples*

$$\begin{array}{r} \{m\,l\,m' \,|\, m, m' \in a\} \cup \\ \{m\,l!\,m' \,|\, m \in a, m' \in M - a\} \cup \\ \{m\,l?\,m' \,|\, m \in M - a, m' \in a\} \end{array}$$

where $l \in \{\mathsf{call}, \mathsf{ret}\}$;

(iii) $\to_a$ *is the least transition relation on* $\mathcal{S}_a$ *closed under the transition rules of Fig. 1. We write* $s \xrightarrow{\alpha}_a s'$ *for* $(s, \alpha, s') \in \to_a$, *and by convention omit* τ, *writing* $s \to_a s'$ *for* $s \xrightarrow{\tau}_a s'$, *and the subscript* a *if clear from the context.*

We identify two kinds of applet composition: structural and behavioural. Structural composition is the composition of applets on the syntactic, graph level, which amounts simply to set union. Behavioural composition denotes the composition of applets on the semantic, behavioural level, which requires the notion of composed applet state.

Definition 4 (Composed Applet State). *A* composed applet state $\langle c_1, \sigma_1 \rangle_{a_1} \,|\, \langle c_2, \sigma_2 \rangle_{a_2} \,|\, \ldots \,|\, \langle c_n, \sigma_n \rangle_{a_n}$ *of* $\mathcal{A}$ *is a mapping from* $\mathcal{A}$ *to applet states, such that at most one applet control point is different from* $\star$; *this is then the* composed control point. $\langle \star, \epsilon \rangle_{a_1} \,|\, \langle \star, \epsilon \rangle_{a_2} \,|\, \ldots \,|\, \langle \star, \epsilon \rangle_{a_n}$ *is the* initial composed state *of* $\mathcal{A}$.

We use here that JavaCard programs are sequential, and thus that there is at most one active applet, *i.e.* at most one applet whose control point is different from $\star$.

The composed behaviour of a set of applets is again given in terms of LTSs by composing the behaviours of the individual applets through simple handshake communication (in the style of CCS), where input actions and output actions synchronize to form perfect actions.

$$(\mathsf{localtransfer})\ \frac{v : (\mathsf{entry} \vee \mathsf{seq}) \quad v : \mathsf{loc}\ m \quad m : \mathsf{loc}\ a \quad v \to_m v'}{\langle v, \sigma \rangle \to \langle v', \sigma \rangle}$$

$$(\mathsf{localcall})\ \frac{\begin{array}{l} v_1 : \mathsf{call}\ m_2 \quad v_1 : \mathsf{loc}\ m_1 \quad m_1 : \mathsf{loc}\ a \quad v_1 \to_{m_1} v_1' \\ v_2 : \mathsf{entry} \quad\ \ v_2 : \mathsf{loc}\ m_2 \quad m_2 : \mathsf{loc}\ a \end{array}}{\langle v_1, \sigma \rangle \xrightarrow{m_1\ \mathsf{call}\ m_2} \langle v_2, v_1' \cdot \sigma \rangle}$$

$$(\mathsf{localreturn})\ \frac{\begin{array}{r} v_1 : \mathsf{loc}\ m_1 \quad m_1 : \mathsf{loc}\ a \\ v_2 : \mathsf{return} \quad v_2 : \mathsf{loc}\ m_2 \quad m_2 : \mathsf{loc}\ a \end{array}}{\langle v_2, v_1 \cdot \sigma \rangle \xrightarrow{m_2\ \mathsf{ret}\ m_1} \langle v_1, \sigma \rangle}$$

$$(\mathsf{sendcall})\ \frac{\begin{array}{c} v : \mathsf{call}\ m_2 \quad v : \mathsf{loc}\ m_1 \quad m_1 : \mathsf{loc}\ a \quad v \to_{m_1} v' \\ \neg m_2 : \mathsf{loc}\ a \end{array}}{\langle v, \sigma \rangle \xrightarrow{m_1\ \mathsf{call!}\ m_2} \langle \star, v' \cdot \sigma \rangle}$$

$$(\mathsf{sendreturn})\ \frac{\begin{array}{r} \neg m_1 : \mathsf{loc}\ a \\ v : \mathsf{return} \quad v : \mathsf{loc}\ m_2 \quad m_2 : \mathsf{loc}\ a \end{array}}{\langle v, m_1 \cdot \sigma \rangle \xrightarrow{m_2\ \mathsf{ret!}\ m_1} \langle \star, \sigma \rangle}$$

$$(\mathsf{receivecall})\ \frac{\begin{array}{r} \neg m_1 : \mathsf{loc}\ a \\ v : \mathsf{entry} \quad v : \mathsf{loc}\ m_2 \quad m_2 : \mathsf{loc}\ a \end{array}}{\langle \star, \sigma \rangle \xrightarrow{m_1\ \mathsf{call?}\ m_2} \langle v, m_1 \cdot \sigma \rangle}$$

$$(\mathsf{receivereturn})\ \frac{\begin{array}{r} v : \mathsf{loc}\ m_1 \quad m_1 : \mathsf{loc}\ a \\ \neg m_2 : \mathsf{loc}\ a \end{array}}{\langle \star, v \cdot \sigma \rangle \xrightarrow{m_2\ \mathsf{ret?}\ m_1} \langle v, \sigma \rangle}$$

Fig. 1. Applet Transition Rules.

Definition 5 (Induced LTS). *A set of applets $\mathcal{A}$* induces *an LTS*

$$\mathcal{T}_{\mathcal{A}} \triangleq (\mathcal{S}_{\mathcal{A}}, \mathcal{L}_{\mathcal{A}}, \to_{\mathcal{A}})$$

as follows:

(i) *$\mathcal{S}_{\mathcal{A}}$ is the set of composed applet states of $\mathcal{A}$;*
(ii) *$\mathcal{L}_{\mathcal{A}}$ is as in Def. 3 (ii);*
(iii) *$\to_{\mathcal{A}}$ is the least transition relation on $\mathcal{S}_{\mathcal{A}}$ closed under the rules for applet states of Fig. 1, extended with the rules for composed states of Fig. 2, where $\mathcal{A}_1$ and $\mathcal{A}_2$ partition $\mathcal{A}$, $l \in \{\mathit{call}, \mathit{ret}\}$, and where $m : \mathsf{loc}\ \mathcal{A}'$ is defined as: for some $a \in \mathcal{A}'$, $m : \mathsf{loc}\ a$.*

We have shown that for closed systems, *e.g.* systems in which all components are available, structural and behavioural composition coincide [4]. As a result, in such cases, single component verification techniques can be used.

As mentioned before, the program model is inspired by work of Jensen *et al.* at INRIA Rennes. Part of the work done in Rennes is the development of a

$$[\mathsf{tau}]\ \frac{\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \longrightarrow_{\mathcal{A}_1} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1}}{\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2} \longrightarrow_{\mathcal{A}} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2}}$$

$$[\mathsf{sync}]\ \frac{\begin{array}{c}\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \xrightarrow{m_1\ l?\ m_2}_{\mathcal{A}_1} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \\ \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2} \xrightarrow{m_1\ l!\ m_2}_{\mathcal{A}_2} \langle c'_{\mathcal{A}_2}, \sigma'_{\mathcal{A}_2}\rangle^{\mathcal{A}_2}\end{array}}{\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2} \xrightarrow{m_1\ l\ m_2}_{\mathcal{A}} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c'_{\mathcal{A}_2}, \sigma'_{\mathcal{A}_2}\rangle^{\mathcal{A}_2}}$$

$$[\mathsf{prop}]\ \frac{\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \xrightarrow{m_1\ l?!\ m_2}_{\mathcal{A}_1} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \qquad \neg m_1 : \mathsf{loc}\ \mathcal{A}_2 \qquad \neg m_2 : \mathsf{loc}\ \mathcal{A}_2}{\langle c_{\mathcal{A}_1}, \sigma_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2} \xrightarrow{m_1\ l?!\ m_2}_{\mathcal{A}} \langle c'_{\mathcal{A}_1}, \sigma'_{\mathcal{A}_1}\rangle^{\mathcal{A}_1} \mid \langle c_{\mathcal{A}_2}, \sigma_{\mathcal{A}_2}\rangle^{\mathcal{A}_2}}$$

Fig. 2. Transition Rules for Composed Applet States.

tool for extracting the program model from a Java byte code program. We have adapted this tool to extract program models that fit in our framework.

2.4 Specification language

As a specification language, we use Kozen's modal μ-calculus [18], extended with control point predicates as model-specific atomic propositions.

Let m range over methods, t over the program point types T, as introduced in Def. 1, and let α range over transition labels.

Definition 6 (Syntax of behavioural properties). *Atomic propositions* p *and* formulae ϕ *are generated by the following grammar, where* X *ranges over propositional variables:*

$$\begin{aligned} p ::= {} & t \mid \mathsf{loc}\ m \mid \mathsf{loc}\ a \mid \mathsf{call}\ m \mid m\ \mathsf{call}\ m \mid \\ & \mathsf{entry}\ m \mid \mathsf{return}\ m \mid \mathsf{external} \mid \mathsf{loc}\ m \end{aligned}$$

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid [\alpha]\,\phi \mid X \mid \nu X.\phi$$

To make the specifications easier to understand, we introduce several standard abbreviations for formulae. Here we mention some (where M denotes a set of methods).

$$\begin{aligned} \textsc{Always}\ \phi \triangleq \nu X.\ \phi \wedge {} & [\tau]\,X \\ & \wedge \forall m_1.\ \forall m_2.\ [m_1\ \mathsf{call}\ m_2]\,X \\ & \qquad \wedge [m_1\ \mathsf{call?}\ m_2]\,X \\ & \qquad \wedge [m_1\ \mathsf{call!}\ m_2]\,X \\ & \qquad \wedge [m_1\ \mathsf{ret}\ m_2]\,X \\ & \qquad \wedge [m_1\ \mathsf{ret?}\ m_2]\,X \\ & \qquad \wedge [m_1\ \mathsf{ret!}\ m_2]\,X \end{aligned}$$

$$\textsc{Never}\ \phi \ \stackrel{\Delta}{=}\ \textsc{Always}\ \neg\phi$$

$$\begin{aligned}\textsc{Within}\ m\ \phi \ \stackrel{\Delta}{=}\ & \forall m_1.\ [m_1\ \mathsf{call}\ m]\ \textsc{Always}_{-\mathsf{ret}\ m_1}\phi \\ & \wedge [m_1\ \mathsf{call?}\ m]\ \textsc{Always}_{-\mathsf{ret!}\ m_1}\phi\end{aligned}$$

$$\textsc{CallsExtOnly}\ M \ \stackrel{\Delta}{=}\ \forall m_1.\ \forall m_2.\ [m_1\ \mathsf{call!}\ m_2]\ m_2 \in M$$

Using these abbreviations, we can express properties such as:

– m_1 never triggers m_2

$$\textsc{Within}\ m_1\ (\textsc{Never}\ \mathsf{call}\ m_2)$$

– Within m_1, m_2 is the only external method that can be called

$$\textsc{Within}\ m_1\ (\textsc{CallsExtOnly}\ m_2)$$

Many useful behavioural properties of applet interactions can be enforced by requiring the program graph to have a certain structure. Therefore, we also propose a language for structural properties. These properties can be checked efficiently over the program graph. The corresponding behavioural properties can then be shown as a logical consequence of the structural properties of the program graphs.

To formulate the structural properties, we use the same syntax, but with the difference that the [ ] modality is not labelled and is interpreted over the transfer edges rather than over transitions. In this logic, we can encode structural properties such as $\mathsf{path}\, p \,\hat{=} \mu X.p \vee \langle\ \rangle X$, meaning there is a path (within the transfer graph) to a control point p.

Using the structural property language, one can express for example, that within a particular method graph there is no path to a call node. It can be shown that this structural property is sufficient to imply the semantical property that within this method no external calls will be made. In the future, we plan to develop techniques for inferring behavioural properties from structural ones.

2.5 Compositional reasoning

To facilitate compositional reasoning we have developed a proof system [6]. This proof system is a Gentzen-style proof system based on the compositional approach advocated by Simpson [24]. This approach has been successfully used for the compositional verification of CCS programs [13], and even of complex telecommunications software written in the Erlang programming language [15].

The proof system uses Gentzen style sequents, *i.e.* proof judgments of the form $\phi_1, \ldots, \phi_n \ \vdash\ \psi_1, \ldots, \psi_n$. The intuitive interpretation of such a sequent is that the conjunction of the antecedents implies the disjunction of the consequents, *i.e.* $\phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi_1 \vee \ldots \vee \psi_n$. Formally, we define the building blocks of our proof system as follows.

Definition 7 (Assertion, Sequent).

(i) *An* assertion γ *is either a* satisfaction assertion $\mathcal{A} : \phi$, *where* ϕ *is a propositionally closed formula, a* transition assertion $\mathcal{A}_1 \xrightarrow{\alpha} \mathcal{A}_2$, *an* atomic formula assertion σ, *a* transfer-edge assertion $v \rightarrow^T v'$, *a* call-edge assertion $v \rightarrow^C v'$, *or a* wellformedness assertion $\mathsf{wf}(\mathcal{A})$.

(ii) *Assertion* $\mathcal{A} : \phi$ *is* valid *for program model* $\mathcal{M}$ *and environment* ρ *if* $\mathcal{A}\rho \in \|\phi\|^{\mathcal{M}}_{\rho}$. $\mathcal{A}_1 \xrightarrow{\alpha} \mathcal{A}_2$ *is valid for* $\mathcal{M}$ *and* ρ *if* $\mathcal{A}_1\rho \xrightarrow{\alpha\rho}_{\mathcal{M}} \mathcal{A}_2\rho$. σ *is valid for* $\mathcal{M}$ *and* ρ *if* $[\![\sigma]\!]^{\mathcal{M}}_{\rho}$. $v \rightarrow^T v'$ *is valid for* $\mathcal{M}$ *and* ρ *if* $v\rho \rightarrow^T v'\rho$ *in* $\mathcal{M}$. $v \rightarrow^C v'$ *is valid for* $\mathcal{M}$ *and* ρ *if* $v\rho \rightarrow^C v'\rho$ *in* $\mathcal{M}$. $\mathsf{wf}(\mathcal{A})$ *is valid for* $\mathcal{M}$ *and* ρ *if* $\mathcal{A}\rho$ *is wellformed in* $\mathcal{M}$.

(iii) *A* sequent *is a proof judgment of the form* $\Gamma \vdash \Delta$, *where* Γ *and* Δ *are sets of assertions.*

(iv) *Sequent* $\Gamma \vdash \Delta$ *is* valid *if, for all program models* $\mathcal{M}$ *and environments* ρ, *whenever all assertions in* Γ *are valid for* $\mathcal{M}$ *and* ρ *then also some assertion in* Δ *is valid for* $\mathcal{M}$ *and* ρ.

Note that wellformedness of program states does not lift to program-state terms in a way which can be captured purely syntactically, and therefore has to be dealt with explicitly in the proof system.

The proof rules can be divided into groups, dealing with the different kind of assertions. First of all, we have the logical and structural rules which can be applied to satisfaction assertions. Application of the rules for the modalities give rise to transition assertions. These transition assertions follow directly from the transition rules in the program model, we refer to [6] for a detailed explanation. Finally, we have several rules dealing with the wellformedness assertions, which are used to eliminate the *impossible* program-state terms.

2.6 Model checking

As explained above, once the system is closed, *i.e.* all components are available, the specifications can be checked using model checking. In order to do this, we define a mapping from our program model into push down automata.

Definition 8 (PDA). *A* pushdown automaton (PDA) *is a tuple*

$$\mathcal{P} \stackrel{\Delta}{=} (Q, \Gamma, \Sigma, P, \alpha_0)$$

where:

(i) Q *is a finite set of* control states*;*

(ii) Γ *is a finite set of* stack symbols*;*

(iii) Σ *is a finite set of* labels*;*

(iv) $P \subseteq (Q \times \Gamma) \times \Sigma \times (Q \times \Gamma^*)$ *is a finite set of* rewrite rules *of the shape* $\langle p, A\rangle \xrightarrow{a} \langle q, \beta\rangle$*;*

(v) $\alpha_0 \in Q \times \Gamma^*$ *is the* initial state.

The operational semantics of PDAs is defined via LTSs in a standard fashion. Finally, we can define how an applet induces a PDA.

Definition 9 (Induced PDA). *An applet a* induces *a PDA*

$$\mathcal{P}_a \triangleq (Q_a, \Gamma_a, \Sigma_a, P_a, \alpha_{0_a})$$

as follows:

(i) Q_a *is the set* $V_a \cup \{\star\}$ *of control points of* a*;*
(ii) Γ_a *is the set* $V_a \cup (M - a) \cup \perp$ *of applet stack symbols of* a*, extended with the bottom-of-stack symbol* $\perp$*;*
(iii) Σ_a *is the set* $\mathcal{L}_a$ *of transition labels of* a*;*
(iv) P_a *is defined through the same set of rules as those in Fig. 1, but with* σ *replaced by* ϵ *in the three return rules and by* s *ranging over* Γ_a *in the remaining rules;*
(v) α_{0_a} *is the state* $\langle \star, \perp \rangle$*.*

The LTS induced by this PDA can be related with the LTS induced directly by the applet via the transition rules of Fig. 1 [4].

Model checking algorithms for push down automata exist [10], and these can thus be used to check the properties of the PDA induced by the applet. In [12] the full verification process is demonstrated on an electronic purse example: first the program model is extracted using the (adapted) tool from INRIA Rennes, then this program model is translated into a push down automata. Finally, the Moped tool [14] is used to check the properties. In this case the properties are specified in LTL, because for the moment we are not aware of a suitable tool for model checking modal μ-calculus properties of pushdown systems.

2.7 Future work

We plan to investigate further the relationship between structural and semantical properties.

Another point of on-going research is whether there are fragments of the logic for which these sequents are decidable, *i.e.* for which there exists a terminating decision procedure, or semi-decidable, *i.e.* for which there exists a decision procedure which is sound but not complete. We will do this by extending the compositional reasoning framework with simulation preorders and maximal model generation. Expected results include:

- Definition of a simulation preorder on program models supporting compositional reasoning and an algorithm for preorder checking.
- Definition of maximal program models for compositional verification, following the approach of Grumberg and Long [16]. This approach allows the problem of proving property decompositions correct to be reduced to model checking.
- Algorithm for maximal program model construction.

– Tool support for the above algorithms.
– Establishing completeness results for fragments of the logic, which will give us an algorithm to decide compositionality proofs.

The work on model checking will continue on developing the toolset, and trying it out in case studies. The current version supports verifying a closed-card scenario (where all applets are known) but not yet an open-card scenario (checking individual applets) which is necessary to properly interface with the work on developing tool support for the compositional proof system.

3 Abstract Interpretation

3.1 Summary of results

Within WP 4.2, the Jakarta toolset has been produced. It contains:

– the Jakarta Specification Language (JSL), a front-end for producing highly readable executable specifications. It is equipped with a type checker;
– the Jakarta Transformation Kit (JTK), a program to manipulate and transform JSL specifications. For example, it can be used to build abstractions of specifications;
– the Jakarta Prover Interface (JPI), a compiler that translates JSL specifications into proof assistants like Coq, Isabelle, PVS, Spike and programming languages like OCaml;
– the Jakarta Automation Kit (JAK), a toolset to support reasoning about executable specifications within proof assistants.

3.2 Motivation

Platform verification is one of the aspects of formal verification for the *JavaCard platform*. It aims at establishing that a platform for smartcard is correct in the sense that it adheres to some acceptable security policy. Such a security policy is usually formulated as a set of constraints like "a reference cannot be cast to an integer", or "the virtual machine cannot jump to an arbitrary address", and in the case of the JavaCard platform should include constraints such as type safety and applet isolation. The purpose of platform verification is to prove formally that these constraints are enforced by well-identified security functions. One important such security function is bytecode verification (it guarantees properties such as type correctness, code containment, proper object initialization, absence of stack overflows); proving its correctness is a prerequisite for Common Criteria evaluations at highest levels, and *Jakarta* has been developed in order to automate the verification.

The correctness of bytecode verification is stated as a correspondence between two flavors of virtual machines [11, 19, 22]:

– a defensive JCVM, which manipulates typed values and performs structural and type checking at run-time. The defensive JCVM is close to Sun's specification, but is inefficient in terms of execution speed and memory consumption;
– an offensive JCVM, which manipulates untyped values and relies on successful bytecode verification to eliminate structural and type checking at run-time. The offensive JCVM is close to an implementation;

The correctness statement can be summarized by the slogan: "the offensive and defensive JCVMs coincide on programs that pass bytecode verification". Formulating precisely and proving the statement requires:

– modeling the defensive and offensive JCVMs and formalizing the notion of correspondence between the two virtual machines in terms of an abstraction function α_{do} that maps states of the defensive machine to states of the offensive machine;
– showing that both machines coincide on programs whose execution on the defensive JCVM does not raise a type error;
– modeling the BCV as a dataflow analysis of an abstract JCVM that only manipulates types and showing that the defensive JCVM does not raise a type error for programs that pass bytecode verification.

In the context of Common Criteria evaluations at the highest levels, such models and proofs are required to be formal and are best developed with the help of proof assistants.

The objective of Jakarta is to provide an interactive environment for transforming virtual machines and for verifying the correctness of these transformations in proof assistants. Examples of such transformations include refinements and optimizations, but to date our work focuses on abstractions. In what follows, we present the abstraction mechanism of Jakarta, and describe how it has been used on the JavaCard virtual machine.

The principle of Jakarta is to provide automatic support for the construction and cross-validation of the virtual machines verifying an arbitrary property P. Concretely, the Jakarta toolset takes as input:

– a P-defensive virtual machine `dexec: dstate → dstate` that manipulates typed values (the notion of type being determined by P) and enforces P through run-time type-checks;
– two abstraction functions:

  ```
  alpha_da : dstate → astate
  alpha_do : dstate → ostate
  ```

  where `astate` and `ostate` respectively denote some suitably defined types of abstract and offensive states.
– transformation scripts which provide instructions on how to conduct the abstraction of the P-defensive virtual machine into a P-abstract (resp. P-offensive) one;

and produces as output a P-offensive and a P-abstract virtual machines:

```
oexec : ostate → orstate
aexec : astate → (list arstate)
```

together with the proofs that they constitute sound transformations of the P-defensive virtual machine. More precisely, the proof of the following properties:

```
∀s:dstate. alpha_da(dexec s) = (aexec (alpha_da s))
∀s:dstate. noerror(s) ⇒ alpha_da(dexec s) = (aexec (alpha_da s))
```

Where `noerror(s)` means that a suitable *bytecode verifier* has accepted the initial state `s`.

3.3 An overview of Jakarta

Jakarta is a set of different tools, that altogether aim at providing efficient means to build, transform and reason about specifications. Jakarta is composed of three main modules.

The Jakarta Specification Language JSL is a small language for describing virtual machines in a neutral mathematical style. JSL is a polymorphically typed language whose execution model is based on term rewriting [2, 8]. A short description of its syntax is given in next section.

The Jakarta Prover Interface provides a means to translate JSL specifications to proof assistants and theorem provers. It is described in section 3.4.

The Jakarta Transformation Kit JTK provides a means to perform automatic abstractions on JSL specifications. It is described in section 3.5.

The Jakarta Specification Language A JSL theory is given by a set of declarations that introduce first-order, polymorphic datatypes and their constructors, and by a set of function definitions that introduce function symbols and fix their computational meaning via conditional rewrite rules. More precisely:

- JSL types are first-order polymorphic types;
- JSL expressions are standard first-order algebraic terms;
- JSL functions are defined by conditional rewrite rules.

Having a restricted framework is useful to abstract or refine specifications and to compile and reason about them in proof assistants.

JSL expressions JSL expressions are first-order terms built from variables and constant symbols. The latter are either constructor symbols, introduced by datatype declarations, or defined symbols, introduced by function definitions. Formally, expressions are defined as follows.

Definition 10.

- *A* signature Σ *consists of a set* $\mathcal{C}$ *of* constructor symbols, $\mathcal{D}$ *of* defined symbols *and an* arity *function* $\mathsf{ar} : (\mathcal{C} \cup \mathcal{D}) \to \mathbb{N}$.
- *Let* Σ *be a signature and let* $\mathcal{V}$ *be a fixed set of* variables. *The set* $\mathcal{E}$ *of* expressions *(over* Σ*) is defined by the abstract syntax:*

  $$\mathcal{E} := \mathcal{V} \mid \mathcal{E} == \mathcal{E} \mid c\ \boldsymbol{\mathcal{E}} \mid f\ \boldsymbol{\mathcal{E}}$$

  where c *ranges over* $\mathcal{C}$, f *ranges over* $\mathcal{D}$, *and in the last two clauses it is assumed that* $\boldsymbol{\mathcal{E}}$ *is of length* $\mathsf{ar}(c)$ *and* $\mathsf{ar}(f)$ *respectively.*
  For every $f \in \mathcal{D}$, *we let* $\mathcal{E}_d(f) \subset \mathcal{E}$ *be the set of expressions with head symbol* f, i.e. *we let* $\mathcal{E}_d(f) = \{f\ \boldsymbol{t} \mid \boldsymbol{t} \in \boldsymbol{\mathcal{E}}\}$. *Moreover, we let* $\mathcal{E}_d$ *denote* $\mathcal{V} \cup (\bigcup_{f \in \mathcal{D}} \mathcal{E}_d(f))$.

JSL provides concrete syntax for records. Internally they are represented as inductive datatypes with a single constructor, as is done in Coq, thus no abstract syntax has to be defined for them explicitly.

JSL rules Defined function symbols are specified by rewrite rules. In term rewriting jargon, our formalism is a restriction of constructor-based oriented conditional rewriting with extra variables. More precisely, the semantics of a defined function symbol is described by a set of oriented conditional rewrite rules of the form

$$l_1 \twoheadrightarrow r_1,\ \ldots\ , l_n \twoheadrightarrow r_n \Rightarrow g \to d$$

A novelty of JSL is to require that the right-hand sides of the conditions, i.e the r_i s, should be patterns with fresh variables (recall that a pattern is a linear term built from variables and constructors). Such a format of rewriting seems unusual but is closely related to pattern-matching in functional programming and proof assistants.

We give below the formal definition of a syntactically correct JSL rewrite rule:

Definition 11 (Rewrite rule). *Let* Σ *be a signature.*

- *The set* $\mathcal{P}$ *of patterns is defined by the abstract syntax*

  $$\mathcal{P} := \mathcal{V} \mid c\ \boldsymbol{\mathcal{P}}$$

  where in the second clause it is assumed that patterns have pairwise disjoint sets of variables (in other words, patterns are required to be linear).
- *An* f-rewrite rule *is a compound* R *of the form*

  $$l_1 \twoheadrightarrow r_1,\ \ldots\ ,\ l_n \twoheadrightarrow r_n \Rightarrow g \to d$$

  where:
  - $r_1\ \ldots\ r_n \in \mathcal{P}$, $l_1\ \ldots\ l_n \in \mathcal{E}_d$ *and* $d \in \mathcal{E}$*;*
  - $g = f\ \boldsymbol{x}$ *where* $\boldsymbol{x} \in \mathcal{V}$ *are pairwise distinct;*

- *for* $1 \le i \le n$, $\mathsf{var}(l_i) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup \ \ldots \ \cup \mathsf{var}(r_{i-1})$*;*
- *for* $1 \le i, j \le n$ *with* $i \neq j$, $\mathsf{var}(r_i) \cap \mathsf{var}(g) = \emptyset$ *and* $\mathsf{var}(r_i) \cap \mathsf{var}(r_j) = \emptyset$*;*
- $\mathsf{var}(d) \subseteq \mathsf{var}(g) \cup \mathsf{var}(r_1) \cup \ \ldots \ \cup \mathsf{var}(r_n)$*.*

– *The* domain $\mathsf{dom}(R)$ *of a rewrite rule* R *is defined as*

$$\mathsf{var}(g) \cup \bigcup_{1 \le i \le n} \mathsf{var}(r_i)$$

– *The set of rewrite rules is denoted by* $\mathcal{R}$*, so* $\mathcal{R} \subseteq \mathsf{List}\ (\mathcal{E}_d \times \mathcal{P}) \times (\mathcal{E}_d \times \mathcal{E})$*.*
– *The set of rewrite systems is denoted by* $\mathbb{R}$*, so* $\mathbb{R} = \mathsf{PowFin}\ \mathcal{R}$*, where* $\mathsf{PowFin}\ E$ *stands for the set of the finite subsets of* E*.*

See Fig. 3 and 4 for examples of JSL specification.

```
data nat = Zero | Succ of nat.

function plus : nat -> nat -> nat :=
begin
  m -> Zero   => plus m n -> n;
  m -> Succ y => plus m n -> Succ (plus y n);
end.
```

Fig. 3. Example of JSL specification: the function plus

3.4 The Jakarta Prover Interface

The Jakarta Prover Interface provides a means to translate JSL specifications to proof assistants and theorem provers. Currently, we support translations from JSL to Coq, Isabelle [21], PVS [23] and Spike [9]. These translations may produce programs which are considered ill-formed by the target system (proof assistants often reject partial, non-deterministic or non-terminating functions), but it has not turned out to be a problem in practice.

Furthermore, we provide a translation from Coq to JSL so as to reuse existing Coq developments and a translation from JSL to OCaml [20] for an efficient execution of JSL specifications. This translation makes use of the *extraction* feature of Coq that allows to extract type and function definition into OCaml code automatically.

3.5 The Jakarta Transformation Kit

The abstraction $\hat{\mathtt{S}}$ of a JSL theory $\mathtt{S}$ is constructed in three successive steps:

1. the user provides for every datatype $\mathtt{T}$ of $\mathtt{S}$ a corresponding abstract datatype $\hat{\mathtt{T}}$ together with an abstraction function $\alpha_{\mathtt{T}}$ from $\mathtt{T}$ to $\hat{\mathtt{T}}$; abstraction functions are written in JSL, but are only allowed to use unconditional rewriting;

2. the user builds a transformation script that guides the abstraction process in places where automatic procedures are inaccurate; the purpose and effect of this script is detailed below.
3. based on the inputs of steps 1 and 2 above, the JTK constructs automatically for every function $\mathtt{f}\colon \mathtt{T}_1 \to \ldots \to \mathtt{T}_n \to \mathtt{U}$ of `S` its abstract counterpart $\hat{\mathtt{f}}\colon \hat{\mathtt{T}}_1 \to \ldots \to \hat{\mathtt{T}}_n \to \hat{\mathtt{U}}$. Unless indicated otherwise by the abstraction script, the function $\hat{\mathtt{f}}$ is constructed by a syntactic translation on the rewrite rules defining `f`, followed by a cleaning phase where vacuous conditions and rules are removed.

Most of the abstraction process is handled automatically but there are however situations where user interaction is required. It is the purpose of the transformation script to facilitate such interactions through a carefully designed set of commands. Each command is of the general form:

$$\langle command\rangle ::= \langle action\rangle\ \langle target\rangle list\ [\mathbf{by}\ \langle expr\rangle]\ [\mathbf{in}\ \langle scope\rangle list]$$

$$\begin{array}{lll}
\langle target\rangle & ::= & \langle f\rangle \mid \langle f\rangle.\langle loc\rangle \mid \langle f\rangle\mathtt{@}\langle nat\rangle \mid \langle expr\rangle \\
\langle scope\rangle & ::= & \langle f\rangle \mid \langle f\rangle.\langle loc\rangle \\
\langle loc\rangle & ::= & \langle nat\rangle \mid \langle nat\rangle.\langle loc\rangle \\
\langle action\rangle & ::= & \mathtt{drop} \mid \mathtt{delete} \mid \mathtt{replace} \mid \mathtt{select} \mid \ldots
\end{array}$$

We illustrate the meaning of target and scope by the following examples (the meaning of actions is illustrated in Fig. 5):

- `f.2` denotes the second rewriting rule of the definition of the function `f`;
- `f.2.3` denotes the third condition of `f.2`. By convention, `f.2.0` stands for the conclusion of `f.2`;
- `f.2.3.1` denotes the left hand side of `f.2.3` and `f.2.3.2` stands for the right hand side of the same;
- `f@i` denotes the `i`th argument of `f`.

3.6 Automating cross-validation of virtual machines

Finally, the *Jakarta Automation Kit* aims at facilitating the process of cross-validating virtual machines in proof assistants. In order to automate the proofs of cross-validation of the three JCVMs in Coq, we have designed specific reasoning tactics that automatically generate induction principles for reasoning about complex recursive functions. For example for the function `ifnull` defined in Fig. 4, we generate an induction principle that decomposes a lemma into five sub-lemmas, one for each rewrite rule defining `ifnull`. These tactics yield drastic improvements; in particular, proof scripts are significantly easier to construct, and their size is reduced by 90% [3].

In a different line of work, we have been experimenting cross-validation of the JCVMs using Spike [9, 7], an automatic theorem prover based on implicit induction. Currently Spike has been used to establish automatically the commutation properties for over two thirds of the instruction set, but we expect that further optimizations on Spike should allow us to handle even more instructions. This experience suggests that first-order theorem proving techniques could be advantageously exploited in the context of platform verification for smartcards.

Application: JSL specification of the JavaCard Virtual Machine To illustrate JSL, we take the formalisation of the JCVM as an example. This application has been the initial motivation for the work on Jakarta. The JSL specification of the JCVM (which is about 10 000 lines long) is derived automatically from the CertiCartes formalisation of the JCVM in Coq.

Basically, the JCVM is described as a state transformer: execution of an instruction modifies the internal state space. The virtual machine takes as input the initial state space, and a program (in cap-format) and executes the appropriate code. The state that is produced by an instruction is tagged with a label `Normal` or `Abnormal`. If the state is tagged with `Abnormal`, this means that something bad has happened: either an exception has been thrown (and not been caught), or there is an error in the bytecode, noticed by the virtual machine. These errors in the bytecode normally are found by a bytecode verifier. Checking for these kind of errors makes our virtual machine defensive, *i.e.* we do not expect the input program to be certified. By making different abstractions of a defensive virtual machine, one can construct different bytecode verifiers.

Fig. 4 gives the JSL specification of the `ifnull` bytecode of the defensive JCVM. `(ifnull branch state)` executes a conditional jump to `branch` (by updating `state`) if the current top value of the operand stack is null, and simply go to the next instruction otherwise. The value tested (`v`) is deleted from the stack. Notice the three rules dealing with errors.

Example of abstraction script: Synthesis of the offensive JCVM

Fig. 5 contains the script used to generate from the defensive JCVM (which is over 10000 lines long) the offensive JCVM (5000 lines). The script begins with a preamble, which specifies the function to abstract, the name of the resulting function (at line 3) and the abstraction functions (at line 4). Then the remaining of the script is divided in two blocks of discarding commands (between lines 8 and 17) and substitution commands (between lines 21 and 26); post-processing commands, which form the last available group of command are not used for this abstraction.

For example, the block between lines 8 and 17 uses the discarding commands **`delete`** and **`drop`**.

- The command **`delete`** is used to suppress all occurrences of the target within the specified scope. For example at line 8, the script forces type verification operations to be removed everywhere in the specification. By propagation, these commands may lead to rules being deleted, in particular because rules whose conclusion has been removed are also removed.
- The command **`drop`** is used to remove some arguments from the signature of a function. Such a command is needed to clean up code that is made vacuous by the abstraction. For example, the function `getfield_obj` takes as third argument some typing information which becomes vacuous in the offensive JCVM. Therefore we instruct the argument to be removed at line 17.

```
data valu_prim = VReturnAddress nat | VBoolean z |
                  VByte z| VShort z | VInt z.

data valu_ref = VRef_null | VRef_array type0 heap_idx |
                VRef_instance cap_class_idx heap_idx |
                VRef_interface cap_interf_idx heap_idx .

data valu = VPrim valu_prim | VRef valu_ref.

data list 'a = Nil | Cons 'a (list 'a).

data exc 'a = Value 'a | Error.

function  ifnull  : bytecode_idx → dstate → rdstate :=
  (stack_f state) → Nil
     ⇒ (ifnull branch state) → (abortCode State_error state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VPrim v0)
     ⇒ (ifnull branch state) → (abortCode Type_error state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VRef vr),
  (res_null vr) → True
     ⇒ (ifnull branch state)
         → (update_frame (update_pc branch
                           (update_opstack (tail (opstack h)) h)) state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → (Value v),
  v → (VRef vr),
  (res_null vr) → False
     ⇒ (ifnull branch state)
         → (update_frame (update_pc (S (p_count h))
                       (update_opstack (tail (opstack h)) h)) state);

  (stack_f state) → (Cons h lf),
  (head (opstack h)) → Error
     ⇒ (ifnull branch state) → (abortCode Opstack_error state).
```

Fig. 4. JSL SPECIFICATION OF THE BYTECODE IFNULL

3.7 Summary

We have described the general structure of the Jakarta toolset, which provides a tool-independent front-end to specify and reason about the JavaCard platform. In particular, the toolset provides support to construct abstractions, in order to derive certified bytecode verifiers and correct offensive virtual machines from the formalization of the JavaCard defensive virtual machine.

Concerning cross validation of JCVMs, a significant level of automation have been reached. We expect however that further automation is possible, and in particular that equational reasoning can be automated by adapting existing connections between Coq and external tools, such as Elan [1]. A current line of work is also to exploit the information available during the abstraction process to generate intermediate lemmas and proofs in order to lift the overall level of automation of the cross validation process.

```
(* Preamble *)

build oexec from dexec using
abstraction functions alpha_do_prim alpha_do ...

(* Discarding commands *)

delete type_verification signature_verification Type_error
       types_match test_exception_aastore assignment_compatible

delete test_exception_putfield_ref.2 test_exception_astore.3
       test_exception_checkcast.4 test_exception_checkcast.5
       test_exception_putstatic_ref.2

delete t0 in putstatic putfield_obj

drop getfield_obj@3 getstatic@1 tableswitch@1 lookupswitch@1

(* Substitution commands *)

replace get_inst_from_heap by oget_inst_from_heap
replace v2t by (Ref Ref_null)
replace (trhi2vr (vr2tr vr) (absolu (v2z v))) by v in putfield_obj.36.0

select vp2z@1 v2z@1 vr2z@1 vr2hi@1 extr_from_opstack@2
select tpz2vp@2 in adr_aload res_opstack getfield_obj putfield_obj ...
```

Fig. 5. Offensive abstraction script

References

1. C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in COQ. In J. Despeyroux, editor, *Proceedings of LFM'00*, 2000. Rapport Technique INRIA.
2. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In C. Muñoz and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
4. G. Barthe, D. Gurov, and M. Huisman. Formal specification of applet interactions. Manuscript.
5. G. Barthe, D. Gurov, and M. Huisman. Compositional specification and verification of control flow based security properties of multi-application programs. In *Proceedings of Workshop on Formal Techniques for Java Programs (FTfJP)*, 2001.
6. G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In *Proc. FASE'02*, number 2306 in LNCS, pages 15–32. Springer, 2002.
7. G. Barthe and S. Stratulat. Automated first order theorem proving techniques for JavaCard. 2002.
8. M. Bezem, J. W. Klop, and R. de Vrijer, editors. *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2002.
9. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, January 1997.
10. O. Burkart and B. Steffen. Model checking the full modal mu-calculus for infinite sequential processes. In *Proceedings of ICALP'97*, number 1256 in LNCS, pages 419–429, 1997.

11. L. Casset and J.-L. Lanet. A Formal Specification of the Java Byte Code Semantics using the B Method. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proceedings of Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999.
12. G. Chugunov, L.-å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *Cardis '02*, 2002. To appear.
13. M. Dam and D. Gurov. Compositional verification of CCS processes. In D. Bjørner, M. Broy, and A.V. Zamulin, editors, *Proceedings of PSI'99*, number 1755 in LNCS, pages 247–256, 1999.
14. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Computer Aided Verification (CAV '00)*, number 1855 in LNCS, pages 232–247. Springer, 2000.
15. L.-å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *Software Tools for Technology Transfer (STTT)*, 2002. To appear.
16. O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. on Prog. Lang. & Syst.*, 16(3):843–871, 1994.
17. T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.
18. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
19. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
20. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, release 3.00*, 2000.
21. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
22. A. Requet. A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of FMICS'00*, pages 29–46, 2000.
23. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
24. A. Simpson. Compositionality via cut-elimination: Hennesy-Milner logic for an arbitrary GSOS. In *Proceedings of the Tenth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 420–430, 1995.