Compositional specification and verification of control flow based security properties of multi-application programs

Gilles Barthe[1], Dilian Gurov[2], Marieke Huisman[1]

[1] INRIA Sophia-Antipolis, France
{Gilles.Barthe,Marieke.Huisman}@sophia.inria.fr
[2] SICS, Sweden dilian@sics.se

Abstract. Jensen *et al.* present a simple and elegant program model, within a specification and verification framework for checking control flow based security properties by model checking techniques. We generalise this model and framework to allow for compositional specification and verification of security properties of multi-application programs. The framework contains a program model for multi-application programs, and a temporal logic to specify security properties about such programs.

1 Introduction

Formal verification of security properties becomes more and more important. An important and interesting class of security properties are control flow based security properties. Jensen *et al.* [7] present a simple and elegant program model which is used to check these kind of properties (using finite-state model checking). This program model is language independent, but it can easily be instantiated for Java or JavaCard. With the program model, also a language to specify security properties is presented. A drawback of this approach is that to check "real world" programs, the state space can become very large.

Verification of security properties is in particular important for the new generation multi-application smart cards. Typical for such multi-application smart cards is that applets can be loaded post-issuance, *i.e.* after initialisation of the card. Therefore, one would like to do the verifications in a compositional way, stating which properties should be satisfied by the components of the system, to ensure the global correctness of the system. When issuing a new applet on the card, one has to check that this new applet satisfies these required properties, in order to know that other applets can safely cooperate with it.

In this paper, we present a framework for compositional verification of multi-application programs, which is a generalisation of the model presented by Jensen *et al.* The framework, which consists of a program model and a specification language, is language-independent, but can easily be instantiated for JavaCard (as in [7]). The framework enables specification and reasoning in a compositional style, and is thus more suited to verify security properties for multi-application smart cards. The program model is designed to be as abstract as possible, while

it still accurately describes the method call behaviour. Further we propose a set of temporal logic patterns which can be used to specify properties over these programs. The temporal logic patterns can be translated into different logics, including the modal μ-calculus [8]. For this logic, a proof system is under development which will allow one to decompose system properties into properties over the individual applets. This verification method fits in well with the nature of smart cards, where applets can be loaded post-issuance, and it makes verification more manageable by reducing the state space. This paper focuses on appropriate specifications of multi-application programs, and on how to specify properties over such programs in such a way that compositional verification can be achieved.

The model and the logic enable us to reason about smart cards at a behavioural level, *i.e.* at the level of method calls. We feel that this is the right level to talk about applet interaction: for the global correctness of the system it is important to know that the components have a certain interface behaviour, and it does not matter how this behaviour is achieved. Only when showing that an applet satisfies the required properties, one has to look at its implementation.

Example: electronic purse To illustrate our approach we discuss an example from [1], which presents a typical verification problem for smart cards. An electronic purse is presented, which contains three applets: a Purse applet `P`, and two loyalty applets: AirFrance `AF`, and RentACar `RaC`. The owner of an electronic purse smart card can decide to join a loyalty program of some company, and load the appropriate applet on his card. The loyalty applets need to be informed about the purchases done with the card, in order to compute the loyalty points.

For efficiency reasons, the electronic purse keeps a log table of bounded size of all credit and debit transactions, and the loyalty applets can request the information stored in this table. For example, if the user wishes to know how many loyalty points he/she has, the loyalty applet will update its local balance first, before returning an answer. Updating the local balance of a loyalty applet consists of two phases: asking the entries of the log table of the purse, and asking the balances of loyalty partners (to compute an extended balance).

In order to ensure that loyalties do not miss any of the logged transactions (if the log table is full, entries will be replaced by new transactions), they can subscribe to the so-called `logFull` service. This service signals all subscribed applets that the log will be emptied soon, and that they should thus update their local balance. In the example, the AirFrance applet is subscribed to this service, but the RentACar applet is not. However, RentACar might be able to implicitly deduce that the log is full, from the fact that AirFrance asks RentACar for its balance information, every time AirFrance gets the `logFull` message. A malicious implementation of the RentACar loyalty applet might therefore request the information stored in the log table, before returning the value of its local balance to AirFrance. This is unwanted, because it might be the case that applets pay for the `logFull` service, and the owner of the purse applet would not want other applets to get this information for free.

Thus, one would like to specify and verify that only applets that are subscribed to the `logFull` service update their balance, until the log is emptied; in particular one would like to specify that the bad scenario, depicted as a message sequence chart in Fig. 1 (where the solid lines indicate method invocations and the dashed lines indicate method returns) can not happen.

The property depicted in Fig 1 can be formulated as: *an invocation of* `AF.logFull` *in the AirFrance applet should not trigger an invocation of* `P.getTrs` *in the Purse applet by the RentACar applet* `RaC`. Below, in Section 2, we will specify this property formally, and we will also show that to establish that this property holds for the system, it is sufficient to show that `AF.logFull` only calls `P.getTrs` and `RaC.getBalance`, while these methods do not call other methods (hence `RaC` never calls `P.getTrs` when `AF.logFull` is called).

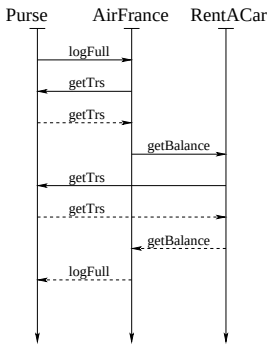

Fig. 1. Electronic purse: bad scenario

The remainder of this paper is organised as follows. Section 2 introduces the temporal logic patterns and show how these are used to specify properties. It also discusses the decomposition theorem. Section 3 discusses the compositional program model, which extends the model of Jensen *et al.* Finally, Section 4 concludes and discusses future work. Throughout the paper, the case study described above will serve as a motivating example.

2 Specifying properties for multi-application programs

Typical properties that are of interest for multi-application programs can often be expressed as temporal logic formulae, stating *e.g.* that a particular event only occurs after some other event has happened. We take the following approach to specification. First we specify the global property (as a temporal logic formula) that should be satisfied by the program. Then we specify which properties should hold for the individual applets (or components) of the program, and we prove formally that if the components satisfy these properties, the global program satisfies the global specification.

The specifications of the global system and the applets are described using temporal specifications patterns, following the approach taken for the Bandera specification language [3]. These patterns have proven useful to specify properties, and can easily be translated into formulae in a particular logic. Typical example patterns that we use are ALWAYS ϕ, WITHIN m ϕ, where m is a method, and A CALLS $\mathcal{M}$, where A is an applet, and $\mathcal{M}$ a set of methods. The temporal logic framework is rich enough to express security properties like the absence of bad scenarios as illustrated above, and it allows a wide range of other important behavioural correctness properties of multi-application programs to be specified.

Using these temporal logic patterns we can specify correctness properties for the electronic purse. As mentioned above we want that an invocation of

`AF.logFull` in the purse does not trigger a call from `RaC` to `P.getTrs`. Formally, we can specify this as $\mathsf{SPEC}_{\mathtt{EP}}(\mathtt{P}, \mathtt{AF}, \mathtt{RaC})$, where:

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{EP}}(X, Y, Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\; Y.\mathtt{logFull}\,. \\ \quad\quad\quad \mathsf{NOT}(Z\;\mathsf{CALLS}\;\{X.\mathtt{getTrs}\})\end{array}$$

where X, Y, Z are variables ranging over applets. This specification states that for any (reachable) state in which the method $Y.\mathtt{logFull}$ has been invoked, but not been finished, there should be no call from the Z applet to $X.\mathtt{getTrs}$.

Based on this specification, we give specifications per applet in such a way that it is sufficient to prove for each applet that it satisfies its local specification, in order to deduce that the global system satisfies the global specification. Finding the local specification requires insight into the system. We specify the purse applet as $\mathsf{SPEC}_{\mathtt{P}}(\mathtt{P})$, the AirFrance applet as $\mathsf{SPEC}_{\mathtt{AF}}(\mathtt{AF}, \mathtt{P}, \mathtt{RaC})$, and the RentACar applet as $\mathsf{SPEC}_{\mathtt{RaC}}(\mathtt{RaC})$, where $\mathsf{SPEC}_{\mathtt{P}}$, $\mathsf{SPEC}_{\mathtt{AF}}$, and $\mathsf{SPEC}_{\mathtt{RaC}}$ are defined as follows.

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{P}}(X) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(X.\mathtt{getTrs})\,. \\ \quad\quad\quad X\;\mathsf{CALLS}\;\{\}\end{array}$$

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{AF}}(Y, X, Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(Y.\mathtt{logFull})\,. \\ \quad\quad\quad Y\;\mathsf{CALLS}\;\{X.\mathtt{getTrs}, Z.\mathtt{getBalance}\}\end{array}$$

$$\begin{array}{l}\mathsf{SPEC}_{\mathtt{RaC}}(Z) \stackrel{\text{def}}{=} \\ \quad \mathsf{ALWAYS}\,. \\ \quad\quad \mathsf{WITHIN}\;(Z.\mathtt{getBalance})\,. \\ \quad\quad\quad Z\;\mathsf{CALLS}\;\{\}\end{array}$$

The specification for the purse applet states that the method $X.\mathtt{getTrs}$ does not invoke any other method. The specification for AirFrance specifies which methods are invoked by $Y.\mathtt{logFull}$. The specification for RentACar specifies that $Z.\mathtt{getBalance}$ should not invoke any other method. Notice that these specifications do not fully specify the behaviour of the applets, they only describe the necessary behaviour in order to satisfy the global property.

Given the global specification $\mathsf{SPEC}_{\mathtt{EP}}$ for the electronic purse, and given the specifications for the individual applets `P`, `AF` and `RaC`, we establish the following theorem, presented as a Gentzen-style sequent, where free variables are (implicitly) universally quantified (where $X : \phi$ is an assertion meaning that applet X satisfies property ϕ).

$$X\colon \mathsf{SPEC}_{\mathtt{P}}(X), Y\colon \mathsf{SPEC}_{\mathtt{AF}}(Y, X, Z), Z\colon \mathsf{SPEC}_{\mathtt{RaC}}(Z) \;\vdash\; X \mid Y \mid Z\colon \mathsf{SPEC}_{\mathtt{EP}}(X, Y, Z)$$

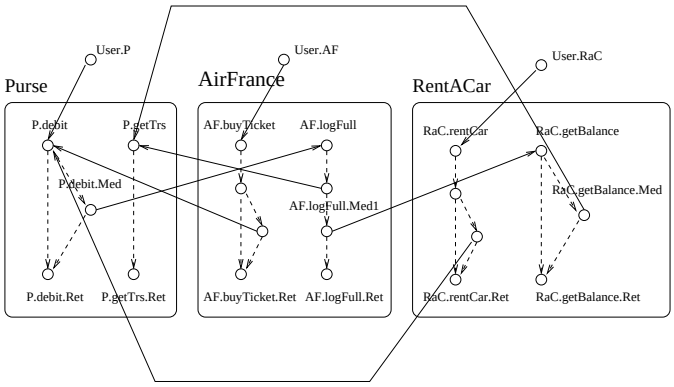

Fig. 2. Compositional model for the purse

Using this theorem one can reduce the proof of the global correctness assertion $\mathtt{P} \mid \mathtt{AF} \mid \mathtt{RaC}\colon \mathsf{SPEC}_{\mathtt{EP}}(\mathtt{P}, \mathtt{AF}, \mathtt{RaC})$ to proving the local correctness assertions $\mathtt{P}\colon \mathsf{SPEC}_{\mathtt{P}}(\mathtt{P}, \mathtt{AF})$, $\mathtt{AF}\colon \mathsf{SPEC}_{\mathtt{AF}}(\mathtt{AF}, \mathtt{P}, \mathtt{RaC})$ and $\mathtt{RaC}\colon \mathsf{SPEC}_{\mathtt{RaC}}(\mathtt{RaC})$ of the individual applets. Notice that we thus have two different kind of verification tasks in our framework, namely model-checking the local properties of the individual applets, and proving property decompositions correct. The use of general temporal logic patterns allows us to use different verification techniques. For example, we can model check the "local" applet properties, by translating the specifications into CTL (*e.g.* as input for NuSMV [2]) or LTL (*e.g.* as input for SPIN [6]), while we can use the modal μ-calculus [8] to prove the correctness of the property decomposition.

3 A program model for multi-application programs

To verify the properties as described above, we need a formal model, representing multi-application programs, with a formal (operational) semantics. This model is designed in such a way that it is suited for compositional verification. Based on the approach taken by Jensen *et al.* [7], we model a program as a transfer graph, modelling intra-procedural control flow, and a call graph, modelling method calls. A special set of vertices is identified, which are the return vertices, where a method hands back control to the caller. A function $\alpha\colon V \rightharpoonup A$ exists, which attributes vertices to applets. This is a partial function, as we allow vertices that do not belong to applets; these are the external vertices that model the environment. To illustrate the model, Fig. 2 shows the electronic purse formalised in this way. A suggestive naming and notation is used, to attribute vertices to applets (the function α), and to suggest the control flow in the methods. For clarity of presentation, in the picture we did not name all the intermediate

vertices. The dashed arrows are edges in the transfer graph, the solid arrows are edges in the call graph.

Every applet has a local state, which is a list of pairs of vertices, representing the control stack in the current program point. For example, given an applet a with local vertices v_2 and v_5, its local state $(v_1, v_2) \cdot (v_2, v_3) \cdot (v_4, v_5)$ can be interpreted as: vertex v_1 (which is external to a) invoked a vertex in a, during whose execution v_2 is reached. Next, v_2 invoked the vertex v_3 in some other applet. Execution continued in this other applet, but eventually somewhere in some applet a vertex v_4 is reached, which invoked a vertex in a again, and during the execution of this vertex, the vertex v_5 is reached.

The operational semantics of individual applets as well as of sets of applets is given compositionally, in terms of labelled transition systems induced by a set of transition rules. The latter are grouped in two parts: transition rules defining the behaviour of individual applets (that is, singleton applet sets), and transition rules for combining behaviours of applet sets.

The transition labels are denoting method invocations and returns. We distinguish between perfect and imperfect actions, the former being either intra-procedural control flow actions (left unlabelled) or method invocations/returns internal to a given applet set (labelled with call and ret, respectively), and the latter being method invocations/returns involving vertices external to the applet set (labelled with call?/ret? for input and call!/ret! for output action, respectively). Imperfect actions can form the corresponding perfect actions by synchronisation in the global trace of the system (thus leaving only the labels call and ret).

Applet transition rules Figure 3 gives the transition rules per applet. In this figure the applet name a is fixed, and π denotes the local state of applet a. We use $v_1 \rightarrow^T v_2$ to denote edges in the transfer graph, modelling intra-procedural control flow, and $v_1 \rightarrow^C v_2$ to denote edges in the call graph, respectively.

We use an applet-state predicate active_a and vertex predicates local_a and return_a, which are defined as follows.

$$\begin{aligned}
\mathsf{active}_a(\pi) &\stackrel{\text{def}}{=} \exists \pi', v, v'.(\pi = \pi' \cdot (v, v')) \ \wedge \ \mathsf{local}_a(v') \\
\mathsf{local}_a(v) &\stackrel{\text{def}}{=} \alpha(v) \in \mathsf{dom}(\alpha) \ \wedge \ \alpha(v) = a \\
\mathsf{return}_a(v) &\stackrel{\text{def}}{=} v \in V^R \ \wedge \ \mathsf{local}_a(v)
\end{aligned}$$

Thus, an applet is active if the second vertex in the last pair of π is local to this applet.

The first three rules describe transitions local to the applet. The rules send call and receive call describe the state transitions when a call to a different applet is made (either from an external vertex, or from applet to applet). Similarly, the rules send return and receive return describe the state transitions if a call over method borders is completed. The receive return transition is enabled if the return is sent by the same applet as the one the corresponding call was send to, there are no requirements on the local state of this applet. This is in accordance with the restrictions on compositional reasoning.

$$[\text{local call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \mathsf{local}_a(v_1) \qquad \mathsf{local}_a(v_2)}{\pi \cdot (v, v_1) \xrightarrow{v_1\ \mathsf{call}\ v_2} \pi \cdot (v, v_1) \cdot (v_1, v_2)}$$

$$[\text{local return}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad \mathsf{return}_a(v_3)}{\pi \cdot (v, v_1) \cdot (v_1, v_3) \xrightarrow{v_3\ \mathsf{ret}\ v_1} \pi \cdot (v, v_2)}$$

$$[\text{local transfer}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad v_1 \not\rightarrow_C}{\pi \cdot (v, v_1) \longrightarrow \pi \cdot (v, v_2)}$$

$$[\text{send call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \mathsf{local}_a(v_1) \qquad \neg\mathsf{local}_a(v_2)}{\pi \cdot (v, v_1) \xrightarrow{v_1\ \mathsf{call!}\ v_2} \pi \cdot (v, v_1) \cdot (v_1, v_2)}$$

$$[\text{receive call}]\ \frac{v_1 \rightarrow^C v_2 \qquad \neg\mathsf{local}_a(v_1) \qquad \mathsf{local}_a(v_2) \qquad \neg\mathsf{active}_a(\pi)}{\pi \xrightarrow{v_1\ \mathsf{call?}\ v_2} \pi \cdot (v_1, v_2)}$$

$$[\text{send return}]\ \frac{\mathsf{return}_a(v_2) \qquad \neg\mathsf{local}_a(v_1)}{\pi \cdot (v_1, v_2) \xrightarrow{v_2\ \mathsf{ret!}\ v_1} \pi}$$

$$[\text{receive return}]\ \frac{v_1 \rightarrow^T_a v_2 \qquad \neg\mathsf{local}_a(v_3) \qquad \alpha(v_3) = \alpha(v_4)}{\pi \cdot (v, v_1) \cdot (v_1, v_3) \xrightarrow{v_4\ \mathsf{ret?}\ v_1} \pi \cdot (v, v_2)}$$

Fig. 3. Applet transition rules

In all rules except receive call it is implicit whether applet a is active or not. The two receive rules are the only two rules that can apply when applet a is not active. Notice how the active applet changes when methods are called and returned: the applet that sends a call has to be active to be able to make the call, and as a result becomes inactive, while the applet that receives the call becomes active. A similar thing applies to the return transitions.

Using these transition rules, one can derive for example the trace fragment in Fig. 4 for the AirFrance applet.

Composing applets Applets can be composed into larger system components. Composite states are sets of local states, with the following restrictions:

- at most one applet is active,
- at most one external vertex is mentioned in the trace, and in this case this vertex occurs as the first component of the first pair of the trace.

The last condition ensures that we can only get single execution threads (which is for the time being appropriate for JavaCard). Computations are always started by the environment, they do not begin spontaneously. External vertices can only invoke methods, and wait for their return. By requiring that external vertices only occur at the beginning of the trace, we enforce that the environment only invokes a method in an applet, if there is no active applet. If necessary this

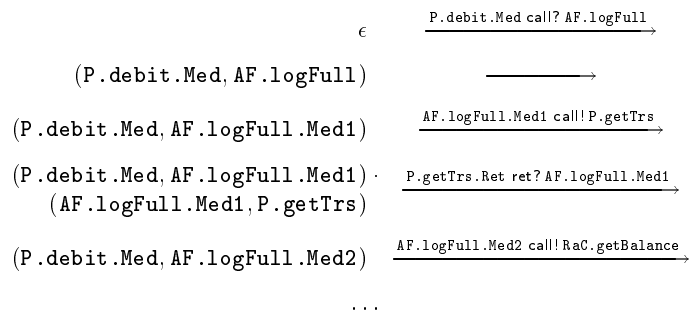

Fig. 4. Local trace AirFrance applet

$$[\mathsf{synchro}]\ \frac{\mathcal{A}_1 \xrightarrow{v_1\ \ell?\ v_2} \mathcal{A}_1' \qquad \mathcal{A}_2 \xrightarrow{v_1\ \ell!\ v_2} \mathcal{A}_2'}{\mathcal{A}_1 \mid \mathcal{A}_2 \xrightarrow{v_1\ \ell\ v_2} \mathcal{A}_1' \mid \mathcal{A}_2'}\ \ell \in \mathsf{call}, \mathsf{ret}$$

$$[\mathsf{propagation}]\ \frac{\mathcal{A}_1 \xrightarrow{\ell} \mathcal{A}_1'}{\mathcal{A}_1 \mid \mathcal{A}_2 \xrightarrow{\ell} \mathcal{A}_1' \mid \mathcal{A}_2}\ \mathsf{perfect}(\ell) \text{ or } \neg\mathsf{involved}_{\mathcal{A}_2}(\ell)$$

Fig. 5. Transition rules for composite states

restriction can be relaxed to allow multi-threading. For the global state, *i.e.* the set of all applets, we strengthen the last restriction and require that the first component in the first pair of the trace is an external vertex. In this way, we ensure that it is always an external vertex that triggers the global execution.

The way the labelled transitions of composite states are induced by the labelled transitions of its subsets is defined through the rules given in Fig. 5. In these rules $\mathcal{A}_1$ and $\mathcal{A}_2$ denote disjoint sets of applet states. Symmetric counterparts exist for both rules. The transition rule synchro applies when both sets of applets can do a transition, labelled with an imperfect action, and when these imperfect actions can synchronise into one perfect action (a perfect action is labelled with call or ret only, it does not contain tags ? or !). This results in a single transition in the composite system, labelled with the corresponding perfect action. The propagation transition rule applies when one set of applets can do a transition, labelled with ℓ, such that ℓ is a perfect action, or ℓ does not involve vertices from applets in the other set. The notion of being involved is defined as follows (where $\mathcal{A}$ is a set of applets).

$$\mathsf{involved}_{\mathcal{A}}(\rho) \stackrel{\text{def}}{=} \exists v1,\ v2 \in V. \exists \ell \in \{\mathsf{call}, \mathsf{ret}\}.\ (\rho = v_1\ \ell?\ v_2 \vee \rho = v_1\ \ell!\ v_2) \wedge (\alpha(v_1) \in \mathcal{A}\ \vee \alpha(v_2) \in \mathcal{A})$$

Using these transition rules, one can find *e.g.* the global trace fragment for the electronic purse, depicted in Fig. 6.

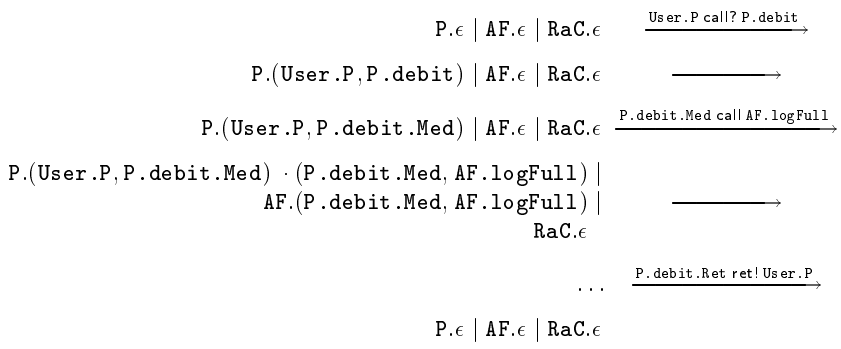

Fig. 6. Fragment of the global trace

4 Conclusions & future work

We have outlined a compositional program model, which will help us to verify security properties over multi-application smart cards. Further we have shown how typical properties of multi-application programs can be specified, and decomposed into specifications over the applets. The program model and logic are language-independent, but can easily be instantiated for JavaCard applications, as is illustrated by the purse example.

Future work The work presented here is only a first step towards a specification and verification framework for (security) properties of multi-application smart cards. Future work will concentrate on the following topics.

- Based on [4, 5] a proof system will be developed (and proven sound and complete) which will allow one to prove the correctness of the decomposition.
- At the moment the program model only deals with the control flow structure of the program. To be able to express integrity properties as *the balance of the purse is not changed by any action in the loyalty applet* one also needs to be able to talk about data. To this end, the program model has to be extended with data. Every applet will contain several variables (or fields), and for each program step it has to be described how these variables might be affected.
- After decomposing the global property, it remains to be shown that the individual applets satisfy the required properties. When dealing with control flow based security properties only, we can fall back on the model checking techniques developed by Jensen *et al.* [7], but after extending the model with data, more sophisticated techniques will be required. Abstraction techniques will be used to simplify the applets and the properties in such a way that they can be checked by model checking.

References

1. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification: extended abstract. In S. Schneider and P. Ryan, editors, *Proceedings of the workshop on secure architectures and information flow*, volume 32 of *Elect. Notes in Theor. Comp. Sci.* Elsevier Publishing, 2000.
2. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *Software Tools for Technology Transfer (STTT)*, 2/4:410–425, 2000.
3. J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, number 1885 in LNCS. Springer, 2000.
4. M. Dam and D. Gurov. Compositional verification of CCS processes. In D. Bjorner, M. Broy, and A.V. Zamulin, editors, *Perspectives of System Informatics '99*, number 1755 in LNCS, pages 247–256. Springer, 1999.
5. M. Dam and D. Gurov. μ-calculus with explicit points and approximations. In *FICS 2000*, 2000.
6. G. Holzmann. The model checker SPIN. *Transactions on Software Engineering*, 23(5):279–295, 1997.
7. 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.
8. D. Kozen. Results on the propositional μ-calculus. *Theor. Comp. Sci.*, 27:333–354, 1983.