Checking Absence of Illicit Applet Interactions: A Case Study*

Marieke Huisman[1], Dilian Gurov[2],
Christoph Sprenger[1], and Gennady Chugunov[3]

[1] INRIA Sophia Antipolis, France
[2] Royal Institute of Technology, Kista, Sweden
[3] Swedish Institute of Computer Science, Kista, Sweden

Abstract. This paper presents the use of a method – and its corresponding tool set – for compositional verification of applet interactions on a realistic industrial smart card case study. The case study, an electronic purse, is provided by smart card producer Gemplus as a test case for formal methods for smart cards. The verification method focuses on the possible interactions between different applets, co-existing on the same card, and provides a technique to specify and detect illicit interactions between these applets. The method is compositional, thus supporting post-issuance loading of applets. The correctness of a global system property can algorithmically be inferred from local applet properties. Later, when loading applets on a card, the implementations are matched against these local properties, in order to guarantee the global property. The theoretical framework underlying our method has been presented elsewhere; the present paper evaluates its practical usability by means of an industrial case study. In particular, we outline the tool set that we have assembled to support the verification process, combining existing model checkers with newly developed tools, tailored to our method.

1 Introduction

The growing market for smart cards and other small personal devices has increased the need to use formal validation and verification techniques in industry. These devices often contain privacy–sensitive information; this is the case in typical usages for smart cards such as health care information systems and electronic purses. Therefore strong security guarantees are needed for their wide–spread acceptance. With the acceptance of evaluation schemes such as Common Criteria[1] industry has come to realise that the only way to achieve such high guarantees is to adopt the use of formal methods in industrial practice.

Various work has been done, aiming at the verification of different kinds of properties of smart card applications. Properties under study are for example functional correctness, confidentiality, availability and restrictions on information flow. Often this work focuses on the correctness of a single applet, or of a set

* Partially supported by the EU as part of the VerifiCard project IST-2000-26328.
[1] See `http://www.commoncriteria.org`.

of applets that is known in advance. However, future generations of smart cards are expected to allow post–issuance loading of applets, where newly installed applets interact with the applets already present on the card. As a consequence, at the time the card is issued, it is not known which applets it might contain. Therefore, it is necessary to state minimal requirements for the applets that can be loaded later on the card, and to be able to verify at loading time that the applets actually respect these requirements. Only then, existing applets can safely communicate with new applets, without corrupting the security of the card.

In the present case study we focus on a particular kind of properties to ensure the security of the card, namely the absence of illicit control flow between the different applets. For multi–application smart cards, certain control flow paths can be undesirable because of general platform–dependent restrictions, like the recommendation to avoid recursion due to limited resources, or due to application–specific restrictions, like undesirable information flow caused by illicit applet interactions as studied in this paper.

In a companion paper we presented an algorithmic compositional verification technique for such control flow based safety properties [14], using a temporal logic specification language for specifying applet properties. These can be either structural, interpreting formulae over the control flow graph of an applet, or behavioural, interpreting formulae over applet behaviour. The approach is compositional in that it allows global control flow properties of the whole system to be inferred from local control flow properties of the individual applets. In this way, global security properties can be guaranteed to hold even in the presence of post–issuance loading of applets, as long as these applets satisfy their local properties. The latter check can be delegated to a separate authority not necessarily possessing the code of the applets already residing on the card. However, while the global properties can be behavioural or structural, we require the local properties to be structural; our technique does not allow global behavioural properties to be algorithmically inferred from local behavioural ones. For a more detailed motivation for using structural assumptions the reader is referred to [14].

An important asset of our method is that the verification tasks involved are all based on algorithmic techniques, as opposed to earlier work in which we developed a proof system for compositional verification [1]. Therefore, once the specifications for the different applets and the illicit applet interaction are given, all verifications can be done automatically, using push–button technology. This paper presents the tool set that we have assembled to support the whole verification process, and illustrates its usefulness by applying it to a realistic, industrial electronic purse case study, provided by the smart card producer Gemplus. The application is not actually used by Gemplus, but has been provided as a test case to apply formal methods to smart card applications. The properties that we verify illustrate typical application–dependent illicit applet interactions.

As far as we are aware, this work is the first to develop algorithmic techniques for the compositional verification of control flow properties for applets. Earlier, we used part of our tool set for non-compositional verification of control flow properties [8]. The underlying program model has been inspired by the work of

Besson *et al.* [2], who verify stack properties for Java programs. Our work differs considerably from more known model checkers for multi-threaded Java such as Bandera [11] and Java PathFinder [5]. In contrast to these tools, we focus on the control flow of applications and the compositionality of the verification. Finally, we mention the model checking algorithms for Push–Down Automata, developed by Bouajjani *et al.* [4]. We use the implementation of these algorithms in the model checker Alfred [13] to verify the correctness of the decomposition.

The paper is structured as follows. First, Section 2 outlines the general structure of the tool set. Next, Section 3 summarises the theoretical framework underlying our approach. Then, Section 4 introduces the electronic purse example, and motivates the property that we are interested in. This property is formalised in Section 5, together with appropriate local properties for the individual applets. Finally, Section 6 discusses the use of our tool set to establish the correctness of the property decomposition and of the local properties *w.r.t.* an implementation. For a more detailed account of the theoretical framework we refer to our companion paper [14].

2 General Overview of the Approach

As explained above, we aim at checking the absence of illicit applet interactions, given the possibility of post–issuance loading, by using a compositional verification method. In our method, we identify the following tasks:

1. specification of global security properties as behavioural safety properties;
2. specification of local properties as structural safety properties;
3. algorithmic verification of property decompositions, ensuring that the local properties imply the global ones; and
4. algorithmic verification of local properties for individual applets.

Our method is based on the construction of maximal applets *w.r.t.* structural safety properties. An applet is considered to be maximal *w.r.t.* a property if it simulates all applets respecting this property.

Concretely, suppose we want to prove that the composition of applets A and B respects a security property, formulated as behavioural safety property ϕ (Task 1). We specify structural properties σ_A and σ_B (Task 2) for which we construct maximal applets $\theta_{I_A}(\sigma_A)$ and $\theta_{I_B}(\sigma_B)$, respectively (where I_A and I_B are the interfaces of the applets A and B, respectively). We show, using existing model checking techniques, that their composition respects the behavioural safety property ϕ, *i.e.* $\theta_{I_A}(\sigma_A) \uplus \theta_{I_B}(\sigma_B) \models \phi$. The validity of this assertion corresponds to the correctness of the property decomposition (Task 3), since the simulation pre–order is preserved under applet composition and behavioural properties expressible in our logic are preserved by simulation. When we get concrete implementations for A and B, we use existing model checking techniques to check whether these implementations respect σ_A and σ_B, respectively (Task 4).

To support our compositional verification method, we have developed a tool set, combining existing model checking tools and newly developed tools, specific

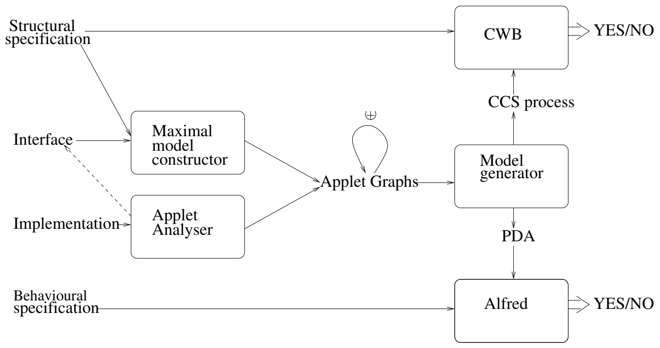

Fig. 1. Overview of tool set

to our method. Figure 1 gives a general overview of our tool set. Section 3 below introduces the underlying theoretical framework.

As input we have for each applet either an implementation, or a structural property, restricting its possible implementations, plus an interface, specifying the methods provided and required by the applet. For these inputs, we construct an applet representation, which is basically a collection of control flow graphs representing methods, plus the applet interface. In case we have the applet implementation, we use the *Applet Analyser* to extract the applet graph. In case we have a structural property, we use the *Maximal Model Constructor* to construct an applet graph that simulates all possible implementations of applets respecting the formula. For a given applet implementation, the Applet Analyser can also be used to obtain the applet interface. If required, applets can be composed, using the *applet composition* operator $\uplus$. This operation essentially corresponds to forming the disjoint union of applets. Using the *Model Generator* the resulting applet graphs are translated into models which serve as input for different model checkers. If we want to check structural properties, we translate the resulting graphs into CCS processes, which can be used as input for the Edinburgh Concurrency Workbench (CWB) [9]. If for a composed system we want to verify whether it respects a behavioural safety property, we translate the composed graphs into Push–Down Automata (PDA), which form the input for the model checker Alfred [13].

3 A Framework for Compositional Verification

This section outlines the theoretical framework underlying our tool set. For a more comprehensive account of the technical details the reader is referred to [14].

3.1 Program Model

As we are only studying control flow properties, we abstract away from all data in our program model. Further, since we are only concerned with smart card applications, we only consider sequential programs[2]. Basically, an applet is a collection of method graphs, plus an interface specifying which methods it provides and requires. For each method, there is a method graph describing its possible control flow. Edges in the graphs denote method calls or internal computations.

As explained above, we distinguish between structural level properties, restricting possible implementations of methods, and behavioural level properties, restricting the possible behaviour of methods. Therefore, we also have two different views on applets (and methods): structural and behavioural. However, these two views are instantiations of a single framework (see [14]).

General Framework. First we present the general framework, defining the notions of *model* and *specification* over a set of labels L and a set of atomic propositions A. These are later instantiated to the structural and behavioural level.

Definition 1. (Model) *A* model *over labels L and atomic propositions A is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, L is a finite set of labels, $\rightarrow \subseteq S \times L \times S$ is a transition relation, A is a finite set of atomic propositions, and $\lambda : S \rightarrow \mathcal{P}(A)$ is a valuation assigning to each state s the atomic propositions that hold at s. A* specification *$\mathcal{S}$ over L and A is a pair $(\mathcal{M}, E)$, where $\mathcal{M}$ is a model over L and A and $E \subseteq S$ is a set of states.*

Intuitively, one can think of E as the set of entry states of the model. We define the usual notion of simulation $\leq$ (where related states satisfy the same atomic propositions).

Applet Structure. Before instantiating the notion of model on the structural level, we first define the notion of applet interface. Let $\mathcal{M}eth$ be a countably infinite set of method names.

Definition 2. (Applet interface) *An* applet interface *is a pair $I = (I^+, I^-)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of* provided *and* required *methods, respectively. The* composition *of two interfaces $I_1 = (I_1^+, I_1^-)$ and $I_2 = (I_2^+, I_2^-)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$.*

As mentioned above, a method specification is an instance of the general notion of specification.

Definition 3. (Method specification) *A* method graph *for $m \in \mathcal{M}eth$ over a set M of method names is a finite model*

$$\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$$

where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, $m \in \lambda_m(v)$ for all $v \in V_m$, i.e. *each node is tagged with the method name, and the nodes $v \in V_m$ with $r \in \lambda_m(v)$ are* return points*. A* method specification *for $m \in \mathcal{M}eth$ over M is a pair $(\mathcal{M}_m, E_m)$, where $\mathcal{M}_m$ is a method graph for m over M and $E_m \subseteq V_m$ is a non–empty set of* entry points *of m.*

[2] For example, Java Card, a dialect of Java for programming smart cards, does currently not allow multi-threading.

Table 1. Applet Transition Rules

$$\text{(transfer)}\quad \frac{m \in I^+ \qquad v \to_m v' \qquad v \models \neg r}{(v, \sigma) \xrightarrow{\varepsilon} (v', \sigma)}$$

$$\text{(call)}\quad \frac{m_1, m_2 \in I^+ \qquad v_1 \xrightarrow{m_2}_{m_1} v_1' \qquad v_1 \models \neg r \qquad v_2 \models m_2 \qquad v_2 \in E}{(v_1, \sigma) \xrightarrow{m_1 \mathsf{call}\ m_2} (v_2, v_1' \cdot \sigma)}$$

$$\text{(return)}\quad \frac{m_1, m_2 \in I^+ \qquad v_2 \models m_2 \wedge r \qquad v_1 \models m_1}{(v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1} (v_1, \sigma)}$$

An applet is basically a collection of method specifications and an interface. For the formal definition we use the notion of disjoint union of specifications $\mathcal{S}_1 \uplus \mathcal{S}_2$, where each state is tagged with 1 or 2, respectively, and $(s, i) \xrightarrow{a}_{\mathcal{S}_1 \uplus \mathcal{S}_2} (t, i)$, for $i \in \{1, 2\}$, if and only if $s \xrightarrow{a}_{\mathcal{S}_i} t$.

Definition 4. (Applet) *An* applet $\mathcal{A}$ *with interface* I*, written* $\mathcal{A} : I$*, is defined inductively by*

– $(\mathcal{M}_m, E_m) : (\{m\}, M)$ *if* $(\mathcal{M}_m, E_m)$ *is a method specification for* $m \in \mathcal{M}eth$ *over* M*, and*
– $\mathcal{A}_1 \uplus \mathcal{A}_2 : I_1 \cup I_2$ *if* $\mathcal{A}_1 : I_1$ *and* $\mathcal{A}_2 : I_2$*.*

An applet is *closed* if $I^- \subseteq I^+$, *i.e.* it does not require any external methods. Simulation instantiated to this particular type of models is called structural simulation, denoted as $\leq_s$.

Applet Behaviour. Next we instantiate specifications on the behavioural level.

Definition 5. (Behaviour) *Let* $\mathcal{A} = (\mathcal{M}, E) : (I^+, I^-)$ *be a closed applet where* $\mathcal{M} = (V, L, \to, A, \lambda)$*. The* behaviour *of* $\mathcal{A}$ *is described by the specification* $b(\mathcal{A}) = (\mathcal{M}_b, E_b)$*, where* $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$ *such that* $S_b = V \times V^*$, i.e. *states are pairs of control points and stacks,* $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\varepsilon\}$*,* $\to_b$ *is defined by the rules of Table 1,* $A_b = A$*, and* $\lambda_b((v, \sigma)) = \lambda(v)$*.*

The set of initial states E_b *is defined by* $E_b = E \times \{\varepsilon\}$*, where* ε *denotes the empty sequence over* V*.*

Note that applet behaviour defines a Push–Down Automaton (see, *e.g.*, [7] for a survey of verification techniques for infinite process structures). We exploit this by using a model checker for PDAs to verify behavioural properties.

Also on the behavioural level, we instantiate the definition of simulation $\leq_b$. Any two applets that are related by structural simulation, are also related by behavioural simulation, but the converse is not true (since behavioural simulation only requires reachable states to be related).

3.2 Property Specification Language

We use a fragment of the modal μ–calculus [12], namely the one excluding diamonds and least fixed points, to express properties restricting applet structure and behaviour[3]. We call this fragment *simulation logic*, because it is able to characterise simulation logically and, vice versa, satisfaction of a formula corresponds to being simulated by a maximal model derived from the formula. Similar logics have been studied earlier for capturing branching–time safety properties (see e.g. [3]). Let $\mathcal{X}$ be a countably infinite set of variables over sets of states. Let $X \in \mathcal{X}$, $a \in L$ and $p \in A$ denote state variables, labels and atomic propositions, respectively. The formulae in simulation logic are inductively defined as follows.

$$\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\phi \mid \nu X.\phi$$

We only consider closed formulae of simulation logic, *i.e.* all variables $X \in \mathcal{X}$ have to be bound by some binder νX. Their semantics is standard, see *e.g.* Kozen [12]. The satisfaction relation is extended from states to specifications as usual: a specification satisfies a formula if all its entry points do. This relation is instantiated at both the structural and the behavioural level, denoted as $\models_s$ and $\models_b$, respectively. For each applet $\mathcal{A} : I$, we have an atomic proposition for each $m \in I^+$ and an atomic proposition r. At the structural level, labels are in $I^- \cup \{\epsilon\}$, and boxes are interpreted over edges in the method graphs. At the behavioural level, labels are in L_b (see Definition 5), and boxes are interpreted over transitions (see Table 1).

Writing specifications in the modal μ–calculus is known to be hard (even in our fragment), therefore we define a collection of commonly used specification patterns (inspired by the Bandera Specification Pattern project [10]). In our experience, all relevant behavioural control flow safety properties can be expressed using a small set of such patterns – however, it is important to remember that one can always fall back on the full expressiveness of simulation logic. Below we present several specification patterns, both at structural and behavioural level. These are all used in the case study at hand.

Structural Specification Patterns. We shall use *Everywhere* with the obvious formalisation:

$$\mathit{Everywhere}\,\sigma \;=\; \nu Z.\, \sigma \wedge [\varepsilon, I^-]Z$$

as well as the following patterns, for method sets M and M' of an applet with interface I:

$$\begin{aligned} M\ \mathit{HasNoCallsTo}\ M' &= \left(\textstyle\bigwedge_{m \in M} \neg m\right) \vee (\mathit{Everywhere}\ [M']\,\mathsf{false}) \\ \mathit{HasNoOutsideCalls}\ M &= M\ \mathit{HasNoCallsTo}\ (I^- \setminus M) \end{aligned}$$

The first pattern specifies that method graphs in the set M do not contain edges labelled with elements of the set M'. The second specifies a closed set of methods M, *i.e.* methods in M only contain calls to methods in M.

[3] In fact, in our theoretical framework, we use an alternative, but equivalent formulation, expressing formulae as modal equation systems.

Behavioural Specification Patterns. Pattern *Always* is standard:

$$\mathit{Always}\,\phi = \nu Z.\,\phi \wedge [L_b]Z$$

For specifying that a property ϕ is to hold within a call to method m, we use the *Within* pattern formalised as follows:

$$\mathit{Within}\,m\,\phi \;=\; \neg m \vee (\mathit{Always}\,\phi)$$

Notice that this is a typical behavioural pattern: the notion of *Within* a method invocation encompasses all methods that might be invoked during the call to m. This reachability notion cannot directly be expressed at the structural level.

Finally, for applet $\mathcal{A} : (I^+, I^-)$ and method set M, we define:

$$\mathit{CanNotCall}\,\mathcal{A}\,M \;=\; \bigwedge_{m \in I^+} \bigwedge_{m' \in M} [m\,\mathit{call}\,m']\,\mathsf{false}$$

This pattern holds for state (v, σ) if no call to a method in M is possible.

3.3 Maximal Models and Compositional Verification

Our compositional verification rests on the idea of constructing a so–called maximal model for a given property (*w.r.t.* a simulation pre–order). For every structural property σ and applet interface I, we can construct a so–called *maximal applet* $\theta_I(\sigma)$, *i.e.* an applet with interface I that simulates all applets with this interface, respecting property σ. As the simulation pre–order is preserved under applet composition and behavioural properties expressible in the logic are preserved by the simulation pre–order, we have the following compositional verification principle:

$$\frac{\mathcal{A} \models_s \sigma \qquad \theta_I(\sigma) \uplus \mathcal{B} \models_b \phi}{\mathcal{A} \uplus \mathcal{B} \models_b \phi}\;(\mathsf{beh\text{-}comp})$$

This rule states that the composition of applets $\mathcal{A}\colon I$ and $\mathcal{B}\colon J$ satisfies (global) behavioural property ϕ, if one can find a (local) structural property σ, satisfied by $\mathcal{A}$, such that the composition of the maximal applet *w.r.t.* σ and interface I, composed with applet $\mathcal{B}$ satisfies property ϕ. Thus, if we are given a structural property for an applet $\mathcal{A}$ and an implementation for an applet $\mathcal{B}$ we can verify whether their composition satisfies the required properties. We use the Maximal Model Constructor to compute $\theta_I(\sigma)$, the Applet Analyser to extract the applet graph for $\mathcal{B}$, and the Model Generator to produce input for Alfred, so it can check $\theta_I(\sigma) \uplus \mathcal{B} \models_b \phi$. Later, when an implementation for applet $\mathcal{A}$ becomes available, it can be verified independently whether it respects σ, by using the Applet Analyser to extract the applet graph for $\mathcal{A}$, and the Model Generator to generate input for CWB, which is used to check structural properties.

Note that, since applet composition is commutative, we can apply the composition principle above to its second premise and also replace applet $\mathcal{B}$ by a local structural property (in the same way as displayed above for applet $\mathcal{A}$).

4 Illicit Applet Interactions in the Electronic Purse

The Gemplus electronic purse case study PACAP [6] is developed to provide a realistic case study for applying formal methods to Java Card applications. The case study defines three applications: *CardIssuer*, *Purse* and *Loyalty*. Typically, a card will contain one card issuer and one purse applet, but several loyalty applets. The property that we verify for this case study only is concerned with *Purse* and *Loyalty,* therefore we will not discuss *CardIssuer* any further. If the card holder wishes to join a loyalty program, the appropriate applet can be loaded on the card. Subsequently, the purse and the different loyalties will exchange information about the purchases made, so the loyalty points can be credited. Current versions of Java Card use shareable interfaces to exchange this kind of information, but in the future this is likely to change. However, for our techniques it is not relevant how this communication exactly takes place, we only require that it is done in terms of method calls. The goal of our work is to ensure that no illicit interactions can happen between the applets on the card.

To understand the property that we are interested in, we look closer at how the purse and the loyalties communicate about the purchases made with the card. For efficiency reasons, the electronic purse keeps a log table of all credit and debit transactions, and the loyalty applets can request the (relevant) information stored in this table. Further, loyalties might have so–called partner loyalties, which means that a user can add up the points obtained with the different loyalty programs. Therefore, each loyalty should keep track of its balance and a so–called extended balance. If the user wishes to know how many loyalty points are available exactly, the loyalty applet will ask for the relevant entries of the purse's log table in order to update its balance, and it will also ask the balances of partner loyalties in order to compute the extended balance.

If the log table is full, existing entries will be replaced by new transactions. In order to ensure that loyalties do not miss any of the logged transactions, they can subscribe to the so–called *logFull* service. This service signals all subscribed loyalties that the log table will be overwritten soon, and that therefore they should update their balances. Typically, loyalties will have to pay for this service.

Suppose we have an electronic purse, which contains besides the electronic purse itself two partner loyalties, say L_1 and L_2. Further, suppose that L_1 has subscribed to the *logFull* service, while L_2 has not. If in reaction to the *logFull* message L_1 always calls an interface method of L_2 (say to ask for its balance), L_2 can implicitly deduce that the log table might be full. A malicious implementation of L_2 might therefore request the information stored in the log table before returning the value of its local balance to L_1. If loyalties have to pay for the *logFull* service, such control flow is unwanted, since the owner of the *Purse* applet will not want other loyalties to get this information for free.

This is a typical example of an illicit applet interaction, that our compositional verification technique can detect. Below, we show how the absence of this particular undesired scenario can be specified and verified algorithmically. We allow an arbitrary number of loyalty applets on the card. Since all loyalty applets have the same interface, we apply class–based analysis. We assume that at verification time only the *Purse* applet has been loaded on the card; the code of the loyalty applet class is not yet available. We use compositional reasoning to reduce the global behavioural property expressing the absence of the scenario described above to local structural properties of the purse and loyalty applet classes. The purse applet code is then checked against its structural property. When the loyalty applet code becomes available, possibly after the card has been issued, it is checked against its structural property before loading it on the card.

5 Specification

This section presents the formalisation of the global and local security properties that we need for our example. The next section discusses the verification of the decomposition and of the implementations *w.r.t.* the local properties.

As mentioned above, communication between applets takes place via so–called shareable interfaces. The *Purse* applet defines a shareable interface for communication with loyalty applets, containing among others the methods *getTransaction,* and *isThereTransaction.* The *Loyalty* applet defines shareable interfaces for communication with *Purse* and with other loyalty applets, containing among others the method *logFull.* The set I_P^+ denotes the methods provided by *Purse,* and M_L^{SI} denotes the set of shareable interface methods of *Loyalty.*

The Global Security Property. To guarantee that no loyalty will get the opportunity to circumvent subscribing to the *logFull* service, we require that if the *Purse* calls the *logFull* method of a loyalty, within this call the loyalty does not communicate with other loyalties. However, as the *logFull* method is supposed to call the *Purse* for its transactions, we also have to exclude indirect communications, via the *Purse.* We require the following global behavioural property:

A call to *Loyalty.logFull* does not trigger any calls to any other loyalty.

This property can be formalised with the help of behavioural patterns:

$$(\phi)\ \textit{Within Loyalty.logFull}$$
$$(\textit{CanNotCall Loyalty } M_L^{SI}) \wedge (\textit{CanNotCall Purse } M_L^{SI})$$

Thus, if loyalty receives a *logFull* message, it cannot call any other loyalty (because it cannot call any of its shareable interface methods), and in addition, if the *Purse* is activated within the call to *logFull,* it cannot call any loyalty applet.

Property Decomposition. Next, we phrase local structural properties for *Purse* and *Loyalty*. Here we explain their formalisation; Section 6 presents how we actually verify that they are sufficient to guarantee the global behavioural property. Within *Loyalty.logFull,* the *Loyalty* applet has to call the methods *Purse.isThereTransaction* and *Purse.getTransaction*, but it should not make any other external calls (where calls to shareable interface methods of *Loyalty* are considered external[4]). Thus, a natural structural property for *Loyalty* would be, informally:

[4] Notice that since we are performing class–based analysis, we cannot distinguish between calls to interface methods of other instances, and those of the same instance.

> From any entry point of *Loyalty.logFull*, the only reachable external calls are calls to *Purse.isThereTransaction* and *Purse.getTransaction.*

Reachability is understood in terms of an extended graph of *Loyalty* containing explicit inter–method call edges.

For the *Purse* applet we know that within a call to *Loyalty.logFull* it can only be activated via *Purse.isThereTransaction* or *Purse.getTransaction.*

> From any entry point of *Purse.isThereTransaction* or *Purse.getTransaction*, no external call is reachable.

Again, reachability should be understood in terms of a graph containing explicit inter–method call edges. As our program model does not contain these, the above properties cannot be formalised directly in our logic. However, they can be formalised on a meta–level; for example for the *Purse,* the property holds, if and only if there exist sets of methods $M_{gT} \subseteq I_P^+$, containing *Purse.getTransaction*, and $M_{iTT} \subseteq I_P^+$, containing *Purse.isThereTransaction,* such that:

$$(\sigma_P)\ \mathit{HasNoOutsideCalls}\ M_{iTT}\ \wedge\ \mathit{HasNoOutsideCalls}\ M_{gT}$$

These sets represent the methods in *Purse* which can be called transitively from *Purse.isThereTransaction* and *Purse.getTransaction*, respectively. We can use the Applet Analyser to find them. Similarly, to express the property for *Loyalty* we need a set of methods $M_{lF} \subseteq I_L^+$ containing *Loyalty.logFull*, such that:

$$(\sigma_L)\ M_{lF}\ \mathit{HasNoCallsTo}\ I_L^- \setminus \left(M \setminus M_L^{SI}\right)$$

where $M = M_{lF} \cup \{Purse.isThereTransaction, Purse.getTransaction\}$. Calls to M_L^{SI} are excluded, since, as explained above, the methods in M_L^{SI} are treated as external. Since we assume that the code of the loyalty applet class is not yet available at verification time, M_{lF} has to be guessed. Here we take the (possibly too) simple choice $M_{lF} = \{\mathit{Loyalty.logFull}\}$. Under this choice, σ_L simplifies to $M_{lF}\ \mathit{HasNoCallsTo}\ I_L^- \setminus \{Purse.isThereTransaction, Purse.getTransaction\}$. However, if later one wishes to load an implementation of *Loyalty* with a different set M_{lF}, correctness of the decomposition can be re–established automatically.

6 Verification

Now that we have specified global and local security properties, we have to show: (1) the local properties are sufficient to establish the global security property, and (2) the implementations of the different applets respect the local properties. In order to do this, we identify the following (independent) tasks, discussed below.

1. Verifying the correctness of the property decomposition by:
   (a) building $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$, the maximal applets for σ_P and σ_L; and
   (b) model checking $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L) \models_b \phi$.
2. Verifying the local structural properties by:
   (a) extracting the applet graphs P of the *Purse* and L of the *Loyalty*; and
   (b) model checking $P \models_s \sigma_P$ and $L \models_s \sigma_L$.

Table 2. Statistics for maximal applet construction.

	# nodes	# edges	constr. time
σ_L	474	277 700	25 min.
σ_P	2 786	603 128	13 hrs.

As explained above, we have developed a tool set to support these verification tasks, combining existing model checking tools (CWB and Alfred) with our own tools (Maximal Model Constructor, Applet Analyser and the Model Generator).

6.1 Correctness of the Property Decomposition

To check correctness of the property decomposition, we construct maximal applets *w.r.t.* the specifications of the *Purse* and the *Loyalty*, and verify whether their composition respects the global behavioural property.

Constructing Maximal Applets. Given applet interface I and structural safety property σ, we produce $\theta_I(\sigma)$, the maximal applet for I and σ, using the procedure described in [14], implemented in Ocaml as the Maximal Model Constructor. The construction proceeds in three steps. First, the interface I is translated into a structural safety property characterising all behaviour possible under this interface. Then, the conjunction of this formula and the property σ is transformed into a semantically equivalent normal form, which can directly be translated into a model. This model is the maximal applet $\theta_I(\sigma)$. In general, the size of a maximal applet is exponential in the size of the input. We implemented some optimisations, which save both time and, more importantly, memory.

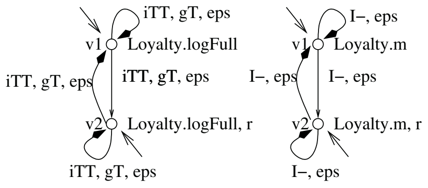

Fig. 2. Methods in $\theta_{I_L}(\sigma_L)$

In the maximal applet for σ_L we can distinguish between two kinds of methods, which are illustrated in Figure 2: the methods in M_{lF} (that is *logFull)* have the left method graph, and only contain calls to *Purse.iTT* and *Purse.gT*. All other methods provided by *Loyalty* have the form of the right method graph, and do not contain any restrictions on the method calls. Each method of the applet $\theta_{I_L}(\sigma_L)$ has two nodes. The maximal applet for σ_P is similar, but each method consists of two to eight nodes depending on the set it belongs to (M_{iTT}, M_{gT} or I_P^+). Table 2 provides statistics on the size of the constructed graphs, and the corresponding construction time on a Pentium 1.9 GHz machine.

Model Checking Behavioural Properties. Once the maximal applets $\theta_{I_P}(\sigma_P)$ and $\theta_{I_L}(\sigma_L)$ are constructed, we produce their composition $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L)$. The behaviour of this applet is a (possibly infinite state) model generated by a pushdown automaton (PDA) given as a set of production rules. The model checking

Table 3. Statistics on applet graph extraction and verification.

	# classes	# methods	# nodes	# edges	extr. time	verif. time
Loyalty	11	237	3 782	4 372	5.6 sec.	12 sec.
Purse	15	367	5 882	7 205	7.5 sec.	19 sec.

problem for this class of models is exponential both in the size of the formula and in the number of control states of the PDA [7]. We base our experiments on Alfred [13], a demonstrator tool for model checking alternation–free modal μ–calculus properties of PDAs. We developed the Model Generator – implemented in Java – to translate applet graphs (in this case $\theta_{I_P}(\sigma_P) \uplus \theta_{I_L}(\sigma_L)$) to a PDA representation, which serves as input to Alfred. We were successful in checking correctness of (similar) property decompositions for applets with a small number of interface methods; when dealing with applets with large interfaces as in our case study, however, Alfred failed to scale up. Currently, we are investigating how to encode applets more efficiently, into context-free processes, which are equivalent to PDAs with a single control state. For this class of processes the model checking complexity becomes polynomial in the number of productions.

6.2 Correctness of the Local Structural Properties

Extracting Applet Graphs. The Applet Analyser is used to extract applet graphs and the appropriate set of entry points from the byte code of an applet. This is a static analysis tool, built on top of the SOOT Java Optimization Framework [15]. The byte code of a Java Card applet is transformed into Jimple basic blocks, while abstracting away variables, method parameters, and calls to methods of the Java Card API. We use SOOT's standard class hierarchy analysis to produce a safe over-approximation of the call graph. If, for example, the static analysis cannot determine the receiver of a virtual method call, a call edge is generated for every possible method implementation. Table 3 provides statistics on the extracted applet graphs.

Model Checking Structural Properties. Applet graphs can be viewed as finite Kripke structures. This allows structural properties expressed in temporal logics to be checked using standard model checking tools such as CWB [9]. The Kripke structures of the CWB are labelled transition systems generated from CCS process definitions. For this purpose, we use the Model Generator to convert applet graphs into a representation as CCS processes. Since CCS does not have the notion of valuation, atomic propositions p assigned to a node in an applet are represented by *probes*, that is, self–loops labelled by p. The translation also produces a set of process constants corresponding to the entry nodes of the respective applet. To model check an applet graph against a structural safety property, all initial states have to be checked individually. We encode the properties to be checked as μ–calculus formulae, replacing atomic propositions p by $\langle p \rangle\,\mathsf{true}$. Since CWB supports parametrised formulae, our specification patterns can directly be encoded.

When verifying $L \models_s \sigma_L$, we realised that in fact the choice of M_{lF} was too optimistic, as the implementation of *Loyalty.logFull* uses several other (internal) methods. Using the Applet Analyser we computed M_{lF} as the set of methods reachable from *Loyalty.logFull*, adapted the specification σ_L and reverified $L \models_s \sigma_L$. Reverifying the decomposition can be done automatically. The last column in Table 3 gives the verification times for model checking $P \models_s \sigma_P$ and $L \models_s \sigma_L$ on a Pentium 1.9 GHz machine.

7 Conclusions

This paper demonstrates a method to detect illicit interactions between applets, installed on a single smart card. The method is compositional, and therefore supports secure post–issuance loading of applets. In particular, the method allows to establish global control flow safety properties for a composed system, provided sufficient local properties are given for the applets. When the applets are loaded (post–issuance) it only remains to be shown that they respect their local property. while the global properties can be structural or behavioural, the local properties need to be structural. To support the specification process, a collection of specification patterns is proposed, with appropriate translations into the underlying logic.

We assembled a tool set – combining existing and newly developed tools – to support the verification tasks that arise in our method. Once the specifications are available, all verifications can be done using push–button technology. Thus, it can be automatically checked whether an applet can be accepted on the card.

The case study shows that the presented verification method and tool set can be used in practice for guaranteeing absence of illicit applet interactions. However, there are some possibilities for improvement. Finding suitable local properties, which requires ingenuity, is complicated by the requirement of formulating local properties structurally. Another difficulty stems from the inherent algorithmic complexity of two of the tasks: both maximal model construction and model checking behavioural properties are problems exponential in the size of the formula, thus making optimisations of these algorithms crucial for their successful application. For some common property patterns such as *Everywhere* σ, the size of the formula depends on the size of the interface. Therefore, it is crucial to develop abstraction techniques to abstract away from method names which are irrelevant to the given property.

Future work will thus go into fine–tuning the notion of interface, by defining public and private interfaces. Now interfaces contain all methods provided and required by a method. We wish to restrict the verification of the global safety properties to public interfaces, containing only the externally visible methods, provided and required by an applet. In order to check whether an implementation respects its local property, we will need to define an appropriate notion of hiding. We also intend to extend the set of specification patterns that we use, by investigating which classes of security properties generally are used. Finally, on a more theoretical side, we will study if we can extend the expressiveness of the logic used (*e.g.* by adding diamond modalities) and under what conditions we can allow behavioural local properties.

References

1. G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering 2002*, number 2306 in LNCS, pages 15–32. Springer, 2002.
2. F. Besson, T. Jensen, D. Le Métayer, and T. Thorn. Model checking security properties of control flow graphs. *J. of Computer Security*, 9(3):217–250, 2001.
3. A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *Automata, Languages and Programming*, pages 76–92, 1991.
4. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.
5. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - second generation of a Java model checker. In *Workshop on Advances in Verification*, 2000.
6. E. Bretagne, A. El Marouani, P. Girard, and J.-L. Lanet. Pacap purse and loyalty specification. Technical Report V 0.4, Gemplus, 2000.
7. O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. North Holland, 2000.
8. G. Chugunov, L.-å. Fredlund, and D. Gurov. Model checking of multi-applet JavaCard applications. In *CARDIS'02*, pages 87–95. USENIX Publications, 2002.
9. R. Cleaveland, J. Parrow, and B. Steffen. A semantics based verification tool for finite state systems. In *Proc. 9th IFIP Symp. Protocol Specification, Verification and Testing*, 1989.
10. J. Corbett, M. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN Model Checking and Software Verification*, number 1885 in LNCS. Springer, 2000.
11. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. Technical report, SAnToS Laboratory, Department of Computing and Information Sciences, Kansas State University, 2000.
12. D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.
13. D. Polanský. Verifying properties of infinite-state systems. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2000.
14. C. Sprenger, D. Gurov, and M. Huisman. Simulation logic, applets and compositional verification. Technical Report RR-4890, INRIA, 2003.
15. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON 1999*, pages 125–135, 1999.