Interface Abstraction for Compositional Verification

Dilian Gurov
Royal Institute of Technology
Kista, Sweden
dilian@imit.kth.se

Marieke Huisman
INRIA Sophia Antipolis
Sophia Antipolis, France
marieke.huisman@inria.fr

Abstract

To support dynamic loading of applications on portable devices, one needs compositional reasoning techniques to ensure that newly loaded applications cannot break the overall security of a device. In earlier work, we developed an algorithmic verification technique for control flow based safety properties of smart card applications, which allows global system properties to be inferred from the properties of the components. Application of the technique requires knowledge of the names of all methods implemented by these components. In a truly compositional setting, however, one only knows the public interface *of the new applet and does not have access to any implementation details. To compositionally verify interface properties of applets, one therefore has to combine our verification technique with an* abstraction *which preserves the* interface behaviour *and reduces the set of implemented methods to the set of public methods. In this paper, we develop such an abstraction technique: we formally define the notion of interface behaviour, and propose an* inlining transformation *which we prove to preserve the interface properties expressible in our specification language. In addition, we show on a concrete case study how the reduction in the number of methods resulting from the interface abstraction drastically improves the performance of the computationally most expensive step of the compositional verification technique.*

1 Introduction

With the emergence of small and mobile personal devices, such as smart cards and mobile phones, security has become a major concern. Typically, such personal devices contain privacy–sensitive information, *e.g.* financial data, health care information or electronic identities. Thus, for the widespread acceptance of the use of such devices, security of private information needs to be guaranteed.

Ideally, a smart device user should have the flexibility to install new applications (usually called *applets*) by need. To enable this, efficient verification techniques are needed for checking, prior to loading a new applet, whether it could break the security of the device. In earlier work, we developed a control–flow based compositional verification technique supporting post–issuance loading of applets, and showed its feasibility by means of an industrial case study [8, 13]. The technique is based on a program model suggested by Jensen and others (see [9]), and addresses safety properties of inter–procedural control flow. These are either structural, *i.e.* properties of the control–flow graph, or behavioural, *i.e.* applet interaction properties describing safe sequences of method invocations. In our set–up, global properties are structural or behavioural, while local properties are structural[1]. Following our technique, compositional verification includes the following steps:

1. Specify a global property ϕ that should hold of the composed system.
2. For each applet $\mathcal{A}_i$, specify a structural local property $\sigma_{\mathcal{A}_i}$.
3. Verify the correctness of the property decomposition (*i.e.*, that the local properties guarantee the global one) by computing, for each applet $\mathcal{A}_i$, its maximal model $\mathcal{M}ax(\sigma_{\mathcal{A}_i})$, and by checking that their composition satisfies ϕ, *i.e.* $\mathcal{M}ax(\sigma_{\mathcal{A}_1}) \uplus \ldots \uplus \mathcal{M}ax(\sigma_{\mathcal{A}_n}) \models \phi$.
4. Whenever the implementation of applet $\mathcal{A}_i$ becomes available, verify that $\mathcal{A}_i \models \sigma_{\mathcal{A}_i}$ prior to loading it on the device.

Notice that the approach also allows a different scenario, where a new applet comes with its own local property, and step 3 is repeated (possibly on–device) to ensure that this local property is sufficient to ensure the security of the whole system.

The maximal model construction of step 3 is the centrepiece of the technique; the remaining verification tasks are

[1]Dealing with local behavioural properties would require restricting the logic and a non-standard maximal model construction.

standard model checking problems for finite state machines (step 4) resp. pushdown automata (step 3). It is inspired by a similar construction due to Grumberg and Long (see [6]), forming the basis of an automatic modular verification technique. Our construction takes a structural property and a set of method names, and returns an applet which satisfies the property and implements the set of methods, and which in addition is maximal, in the sense that it simulates, both structurally and behaviourally, all other such applets. Furthermore, properties are preserved by simulation, and simulation is preserved by composition. So, by composing the respective maximal models and by checking that their composition satisfies the global property, one can show that any applet implementations satisfying the respective structural properties and implementing the respective sets of methods, when put together will satisfy the given global property. Notice, however, that the maximal model construction needs the names of all methods implemented by the given applets. The correctness of a property decomposition can thus only be established for applets with a known interface. This is a limitation of the proposed technique, since in a truly compositional setting one only knows the public interface of yet unavailable applets and one does not know their implementation details. Component properties can hence only be specified at the public level. Moreover, the lack of a mechanism for abstraction from private methods causes a blowup in the size of the formulae and hence of the maximal models used for the verifications, since these are parameterized on the interface considered. For example, in the industrial case study we considered an electronic purse applet which implemented 367 methods, of which only 4 were public, all others were private.

To be able to abstract from internal, private behaviour and apply our compositional verification method to interface properties of applets, one therefore needs an *abstraction technique* which (i) preserves the interface behaviour of applets, while (ii) reducing their set of methods to the set of public methods. The latter requirement comes from the maximal model construction. In this paper we propose an abstraction based on *inlining* of private methods. We define the notion of *interface behaviour*, and show the abstraction to be sound with respect to public interface properties: every property that holds for the interface behaviour of the inlined applet (which coincides with its behaviour since it has no private behaviour) also holds for the interface behaviour of the original applet. Moreover, in case the concrete implementation is last-call recursive (that is, recursive calls are not followed in the control flow graph by any other method calls[2]), the abstraction technique is also complete with respect to public interface properties: if such a property does not hold of the inlined applet it does not hold of the original applet either. Completeness, however, does not hold in general, since the abstraction transformation can introduce new observable behaviours.

[2]This notion is a generalization of the notion of tail recursivity, where recursive calls are the last statements of their methods.

Using the abstraction techniques described in this paper, our improved scenario for secure post-issuance loading becomes:

1. Identify the set of public methods M used for interaction between applets $\mathcal{A}_1, \ldots, \mathcal{A}_n$.

2. Specify a public global property ϕ over M that should hold of the composed system.

3. For each applet $\mathcal{A}_i$, specify a structural local property $\sigma_{\mathcal{A}_i}$ which only mentions (public) methods in M.

4. Compute, for each applet $\mathcal{A}_i$, its maximal model $\mathcal{M}ax(\sigma_{\mathcal{A}_i})$, and check that their composition satisfies ϕ, *i.e.* $\mathcal{M}ax(\sigma_{\mathcal{A}_1}) \uplus \ldots \uplus \mathcal{M}ax(\sigma_{\mathcal{A}_n}) \models \phi$.

5. Whenever the implementation of applet $\mathcal{A}_i$ becomes available, compute the abstraction $\alpha_M(\mathcal{A}_i)$, and verify that $\alpha_M(\mathcal{A}_i) \models \sigma_{\mathcal{A}_i}$.

In addition, we show that specifying structural properties for an inlined applet allows some natural properties to be expressed which are not expressible as properties of the original applet. For instance, reachability properties of the call graph can only be expressed as structural properties of the inlined applet, since there are no explicit inter-method call arcs in our program model.

The present paper shows that all verifications can be done efficiently. In particular, Section 7 reconsiders the case study [8].

Related work The inlining procedure as described in this paper closely resembles standard inlining procedures used in compiler optimisations, see *e.g.* [10]. However, compiler optimisations *must* be behaviourally equivalent, while our verification technique only requires that all existing behaviours are preserved by inlining. We believe that our approach for proving property preservation is applicable to such compiler optimisations as well.

The approach of combining property preserving abstraction with verification is standard, see *e.g.* [3]. Usually, the goal of applying abstraction is to obtain a smaller or simply finite model for verification. In our case, the primary purpose of applying the inlining transformation is different: to reduce the set of methods to the set of public methods while preserving the interface behaviour.

Typically, the behaviour of programs with recursion is modeled as pushdown systems, as *e.g.* in [5]. The notion of interface behaviour presented in the present paper also defines a pushdown system, and hence inlining is generally not needed for the verification of behavioural properties. In our approach, the need for inlining comes from the requirements of the maximal model construction.

Finally, we should mention the temporal logic of calls and returns CaRet [1]. This logic allows to specify properties in terms of method calls and returns. A special verification strategy is defined, that is able to jump over internal computations. Our approach is in a way the opposite: we compute an abstract model, and use standard verification techniques to verify properties – expressed in a standard temporal logic – on this abstract model.

Overview of the paper The remainder of this paper is organised as follows. Sections 2 and 3 introduce the necessary background, and in particular the logic and program model that we use. Section 4 defines the behaviour of an applet *w.r.t.* a set of public methods. Next, Section 5 presents the inlining algorithm that forms the basis of our abstraction technique, and proves that it is property preserving. Section 6 describes formally how the abstraction techniques are used for compositional reasoning. Finally, Section 7 revisits the industrial case study, and shows the practical impact of the abstraction techniques, while Section 8 draws more general conclusions on the applicability of our method.

2 Simulation and Logic

First, we briefly recall some definitions and results that form the basis for our compositional verification method. For a full overview, the reader is referred to [12, 13]. We use a subset of the modal μ-calculus [11] as our specification language. We exploit that formulae in this subset can be characterised by simulation, and vice versa, therefore we call this logic *simulation logic.* Throughout, we fix a set of labels L, a set of atomic propositions A, and a set of propositional variables V.

Definition 2.1. ***(Simulation Logic)*** The formulae of *simulation logic* are inductively defined by:

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

where $p \in A$, $a \in L$ and $X \in V$.

Next, we define a general notion of model and specifications.

Definition 2.2. ***(Model)*** A *model* is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a transition relation, and $\lambda \colon S \rightarrow \mathcal{P}(A)$ a valuation, assigning to each state s the atomic propositions that hold in s. A *specification* $\mathcal{S}$ is a pair $(\mathcal{M}, E)$, where $\mathcal{M}$ is a model and $E \subseteq S$ is a set of states.

Intuitively, one can think of E as the set of entry states of the model. For specifications, we define the usual notions of satisfaction $\models$ and simulation $\leq$ (where related states satisfy the same atomic propositions). This simulation relation preserves (backwards) logical properties.

Theorem 2.3. *$\mathcal{S}_1 \leq \mathcal{S}_2$ and $\mathcal{S}_2 \models \phi$ implies $\mathcal{S}_1 \models \phi$*

Proof. Corollary 2.16 in [13] □

Weak simulation and logic. In Section 4 we show how private method calls can be abstracted away into internal transitions, labelled with the distinguished *silent* action ε. When abstracting in such a way from part of the concrete behaviour of a system, one also has to abstract from the internal behaviour, and instead consider the visible behaviour in terms of *weak* transitions. We use the standard definition of weak transitions $s \overset{a}{\Rightarrow} t$ in terms of strong transitions: $s \overset{\varepsilon}{\Rightarrow} t$ whenever $s(\overset{\varepsilon}{\rightarrow})^* t$, and $s \overset{a}{\Rightarrow} t$ whenever $s \overset{\varepsilon}{\Rightarrow}\overset{a}{\rightarrow}\overset{\varepsilon}{\Rightarrow} t$, for all $a \neq \varepsilon$. Weak simulation $\leq_w$ is then defined as simulation *w.r.t.* weak transitions. Similarly, for the weak satisfaction relation $\models_w$, we interpret the box modality over the weak transitions. As above, the weak simulation relation preserves weak satisfaction of logical properties.

Theorem 2.4. *$\mathcal{S}_1 \leq_w \mathcal{S}_2$ and $\mathcal{S}_2 \models_w \phi$ implies $\mathcal{S}_1 \models_w \phi$*

Proof. Immediate consequence of Theorem 5 in [12]. □

Finally, a standard transformation from weak to strong formulae exists [14]. This transformation, which we denote δ, can be characterised as follows.

Proposition 2.5. *$\mathcal{S} \models_w \phi$ iff $\mathcal{S} \models \delta(\phi)$.*

3 Applet Structure and Behaviour

Our program model, inspired by [9], is control–flow based and thus over–approximates actual program behaviour. It defines two different views on applets: a structural and a behavioural view. Both views are instantiations of the general notions of model and specification, allowing the results presented above to be instantiated at both levels. Notice in particular that these instantiations yield a structural and a behavioural version of simulation and simulation logic. Again, we refer to [12, 13] for more detail.

Applet Structure Since we abstract away from all data, applet structure is defined as a collection of call graphs for the methods the applet implements. Further, since smart cards are our primary application domain, we only consider sequential methods[3]. Let $\mathcal{M}eth$ be a countably infinite set of method names. A method specification is an instance of the general notion of specification.

[3]With the possible emergence of multi-threaded smart card platforms the techniques presented here will have to be generalized accordingly.

Definition 3.1. ***(Method specification)*** A *method graph* for $m \in \mathcal{M}eth$ over a set $M \subseteq \mathcal{M}eth$ of method names is a finite model

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

where V_m is the set of control nodes of m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\}$, and $\lambda_m \colon V_m \to \mathcal{P}(A_m)$ is so that $m \in \lambda_m(v)$ for all $v \in V_m$ (*i.e.* each node is tagged with its method name). The nodes $v \in V_m$ with $r \in \lambda_m(v)$ are called *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.

We write $\lambda_{\mathsf{Meth}}(v)$ to denote the function returning the name of the method to which v belongs.

Next we define the notion of applet interface. For each applet, we distinguish an *implementation interface,* defining all methods provided and required by the applet, and a *public interface,* defining all methods that are visible to and used from other applets.

Definition 3.2. ***(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 $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-)$.

To formally define the notion *applet with implementation interface*, 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 3.3. ***(Applet)*** An *applet* $\mathcal{A}$ with implementation 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.

The *public interface* of an applet $\mathcal{A} \colon I$ is characterised by a set of methods M such that $M \subseteq I^+$: the set of methods publicly provided by the applet is M, while the set of publicly required methods is $I^- - (I^+ - M)$; thus applet $\mathcal{A} \colon I$ has public interface $(M, I^- - (I^+ - M))$. The left-hand column of Figure 1 on page 6 is an example of a closed applet with one public method m and two private methods a and b.

Simulation and satisfaction, instantiated to this particular type of models are called structural simulation $\leq_s$, and structural satisfaction $\models_s$, respectively.

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

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

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

Table 1. Applet Transition Rules

Applet Behaviour Next we instantiate specifications on the behavioural level.

Definition 3.4. ***(Behaviour)*** Let $\mathcal{A} = (\mathcal{M}, E) : I$ 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 $\mathcal{M}_b = (S_b, L_b, \to_b, A_b, \lambda_b)$ is such that $S_b = V \times V^*$, *i.e.* states are pairs of control points and stacks, $L_b = \{m_1\, k\, m_2 \mid k \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\varepsilon\}$, $\to_b$ 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 $E_b = E \times \{\epsilon\}$, where ϵ denotes the empty sequence over V.

Note that applet behaviour defines a pushdown automaton. We exploit this by using a model checker for PDAs to verify behavioural properties (see, *e.g.*, [2] for a survey of verification techniques for infinite process structures).

Also on the behavioural level, we instantiate the definitions of simulation $\leq_b$ and satisfaction $\models_b$. Any two applets that are related by structural simulation, are also related by behavioural simulation (Theorem 3.9 in [13]), but the converse is not true (since behavioural simulation only requires reachable states to be related).

For convenience, below we will often write the states of the behavioural model as a simple sequence of states, *i.e.* $v \cdot \sigma$, instead of (v, σ). We use reverse indexing to denote the i^{th} element from the back of a sequence, so that $(v \cdot \sigma)_{|\sigma|} = v$ (where $|\sigma|$ denotes the length of a sequence σ), and use $last(\sigma)$ to denote σ_0.

4 Interface Behaviour

The next section defines an inlining algorithm that transforms a concrete applet implementation into an applet that contains only method calls to public methods. We want to prove that for any closed applet, every behaviour of the concrete applet is also a behaviour of the inlined applet. However, for this to hold, we have to abstract the concrete behaviour to the level of public methods. Therefore, we introduce the notion of *interface behaviour* of an applet *w.r.t.* a set of public methods M.

First we define the *top* public method *w.r.t.* M, which for a given callstack σ is the first public method to which a node in the call stack belongs.

$$\begin{aligned}\mathsf{top\_index}^M(\sigma) &= Max(\{i \mid 0 \leq i < |\sigma| \wedge \\ &\qquad \lambda_{\mathsf{Meth}}(\sigma_i) \in M\}) \\ \mathsf{top}^M(\sigma) &= \lambda_{\mathsf{Meth}}(\sigma_{\mathsf{top\_index}^M(\sigma)})\end{aligned}$$

Using these definitions, we can define a relabelling ρ^M of transition labels to the public level. Labels for calls and returns between public methods are left unchanged. A call from a private to a public method is relabelled as a call from the *top* public method in the pending call stack. A return from a public to a private method is relabelled as a return to the *top* public method. All other transitions get labelled as silent actions.

$$\rho^M((v,\sigma),\ell) = \begin{cases} \ell & \text{if } \ell = m_1\{\mathsf{call}/\mathsf{ret}\}m_2 \wedge \\ & \quad m_1, m_2 \in M \\ \mathsf{top}^M(v\cdot\sigma)\ \mathsf{call}\ m_2 & \text{if } \ell = m_1\ \mathsf{call}\ m_2 \wedge \\ & \quad m_1 \notin M, m_2 \in M \\ m_1\ \mathsf{ret}\ \mathsf{top}^M(\sigma) & \text{if } \ell = m_1\ \mathsf{ret}\ m_2 \wedge \\ & \quad m_1 \in M, m_2 \notin M \\ \varepsilon & \text{otherwise} \end{cases}$$

Now we are ready to define the interface behaviour of applet $\mathcal{A}$ *w.r.t.* a set of public methods M.

Definition 4.1. ***(Interface behaviour)*** Let $\mathcal{A} : I$ be a closed applet with behaviour $b(\mathcal{A}) = ((S, L, \rightarrow, A, \lambda), E)$. Let $M \subseteq I^+$ be a set of public methods. The *interface behaviour of* $\mathcal{A}$ *w.r.t.* M is defined as $b^M(\mathcal{A}) = ((S, L^M, \rightarrow^M, A^M, \lambda^M), E^M)$, where

- $L^M = \{\varepsilon\} \cup \{m_1\,k\,m_2 \mid \quad m_1, m_2 \in M \wedge k \in \{\mathsf{call}, \mathsf{ret}\}\}$
- $\rightarrow^M = \{ \quad ((v,\sigma), \ell, (v',\sigma')) \mid \exists a \in L.\ (v,\sigma) \xrightarrow{a} (v',\sigma') \wedge \rho^M((v,\sigma),a) = \ell\,\}$
- $A^M = M \cup \{r\}$
- $\lambda^M = (v,\sigma) \mapsto \quad \{\mathsf{top}^M(v\cdot\sigma)\} \cup \mathsf{if}(v \in M \ \wedge v \models r)\ \mathsf{then}\ \{r\}\ \mathsf{else}\ \varnothing$
- $E^M = \{v \mid v \in E \ \wedge \ \lambda_{\mathsf{Meth}}(v) \in M\}$.

The interface behaviour of an applet also defines a push-down automaton.

Proposition 4.2. *The interface behaviour of* $\mathcal{A}$ w.r.t. I^+ *is identical to its behaviour,* i.e. $b^{I^+}(\mathcal{A}) = b(\mathcal{A})$.

We define behavioural interface simulation $\mathcal{A} \leq_b^M \mathcal{B}$ as $b^M(\mathcal{A}) \leq b^M(\mathcal{B})$, and weak behavioural interface simulation $\mathcal{A} \leq_{b,w}^M \mathcal{B}$ as $b^M(\mathcal{A}) \leq_w b^M(\mathcal{B})$. Notice that $\mathcal{A}$ and $\mathcal{B}$ need not have the same interfaces – we only require $M \subseteq I_{\mathcal{A}}^+$ and $M \subseteq I_{\mathcal{B}}^+$. Similarly, for any formula ϕ in simulation logic over L^M and A^M, we define behavioural interface satisfaction $\mathcal{A} \models_b^M \phi$ as $b^M(\mathcal{A}) \models \phi$, and weak behavioural interface satisfaction $\mathcal{A} \models_{b,w}^M \phi$ as $b^M(\mathcal{A}) \models_w \phi$.

5 The Inlining Transformation

Next we define an inlining algorithm α_M that, given a set of public methods M, transforms an applet graph by inlining all private calls. Recursive calls to private methods are not inlined, but create loops in the resulting graph. We prove that the interface behaviour of the original applet $\mathcal{A}$ is simulated by the behaviour of the inlined applet $\alpha_M(\mathcal{A})$, thus (by Theorem 2.3) all properties ϕ of the latter, *i.e.* $\alpha_M(\mathcal{A}) \models_b \phi$, are also properties of the former, *i.e.* $\mathcal{A} \models_b^M \phi$. Moreover, we prove that if the applet is last-call recursive, the two behaviours are weak simulation equivalent – thus both applets satisfy exactly the same observable properties at the public interface level.

Notice that the inlining algorithm does not require the applet to be closed and treats all external methods as public.

The Inlining Algorithm. The algorithm is applied to each public method and (recursively) inlines all calls to private methods. Intuitively, constructing the transformed (or inlined) graph for a public method m corresponds to executing the interface behaviour of m, where method calls to public methods are skipped and recursion is replaced by iteration. The nodes of the inlined applet can thus be seen as states of the (interface) behaviour of the original applet, modulo an abstraction function which replaces recursion by iteration.

During the inlining, each edge that represents internal transfer or a call to a public method is left unchanged. Each edge that represents a call to a private method is replaced by two internal edges: one from the calling point to the entry point of the method; and another from the return point of the method to the destination of the calling edge[4]. The private method is inlined recursively. Each node is replaced by a sequence denoting the fragment of the call stack from the activation of the public method up to the current node (except for the case of a recursive call). Since we keep track of the pending call stack, we can recognise recursive calls to private methods. In that case, the appropriate initial fragment of the call stack is used to decide the exact new edges.

For the formal definition of the inlining algorithm, we need some new notions. Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$

[4]If a method has several entry or return points, several internal edges are created.

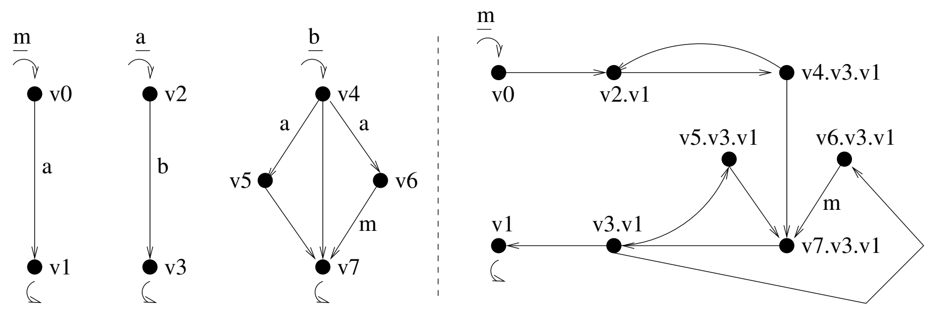

Figure 1. Example applet before and after inlining

be a set of public methods. An M-*frame* is a sequence of nodes σ of which only $\lambda_{\mathsf{Meth}}(\sigma_0)$ is in M. An M-frame is called *normal*, if all nodes in the frame belong to different methods. We choose to represent the nodes of the inlined applet by normal M-frames derived from the behaviour of the original applet. The abstraction function mentioned above (replacing recursion by iteration) is formalised by means of the (normalising) conditional rewrite rule $\sigma \cdot v \cdot \sigma' \cdot v' \cdot \sigma'' \hookrightarrow \sigma \cdot v \cdot \sigma''$ if $\lambda_{\mathsf{Meth}}(v) = \lambda_{\mathsf{Meth}}(v')$ and $\sigma' \cdot v' \cdot \sigma''$ is a normal M-frame. Let $\nu(\sigma)$ denote the normal form of σ *w.r.t.* the rule. Note that if σ is an M-frame, then $\nu(\sigma)$ is a normal M-frame. Moreover, for any (normal) M-frame σ we have $\mathsf{top}^M(\sigma) = \lambda_{\mathsf{Meth}}(\sigma_0)$.

Further, we define $\mathcal{I}nt$, $\mathcal{P}ub$ and $\mathcal{P}riv$, denoting the sets of internal, public and private edges of a method *w.r.t.* a set of public methods M, respectively.

$$\begin{array}{rcl} \mathcal{I}nt(m) & = & \{(v, \varepsilon, v') \mid v \rightarrow_m v' \wedge v \models \neg r\} \\ \mathcal{P}ub_M(m) & = & \{(v, m', v') \mid v \xrightarrow{m'}_m v' \wedge \\ & & \qquad v \models \neg r \wedge m' \in M\} \\ \mathcal{P}riv_M(m) & = & \{(v, m', v') \mid v \xrightarrow{m'}_m v' \wedge \\ & & \qquad v \models \neg r \wedge m' \notin M\} \end{array}$$

The definition of the inlining algorithm uses auxiliary functions χ and ζ. The function χ considers all edges related to a method: it returns internal and public edges with renamed nodes – using the pending call stack, and calls function ζ on private edges. Function ζ adds edges to the entry point, and from the return point of the private method, using the pending call stack argument, and if necessary normalising the result (this uses the fact that the pending call stack is always a normalised M-frame). Then it checks if the private call is non-recursive, in which case the private method is inlined recursively.

Definition 5.1. ***(Inlined applet)*** Let $\mathcal{A} : I$ be an applet, and let M be a set of public methods, such that $M \subseteq I^+$. Let M' be the set $M \cup (I^- - I^+)$. We define the *inlined applet* $\alpha_M(\mathcal{A}) = ((V', L', \rightarrow', A', \lambda'), E')$, where

- $V' = \{w \in V^+ \mid w \text{ is a normal } M\text{-frame}\}$,
- $L' = M' \cup \{\varepsilon\}$,
- $\rightarrow' = \bigcup_{m \in M} \chi(m, \epsilon)$ *where*

$$\begin{array}{l} \chi(m, \sigma) = \\ \quad \{(v \cdot \sigma, \ell, v' \cdot \sigma) \mid (v, \ell, v') \in \mathcal{I}nt(m) \cup \mathcal{P}ub_{M'}(m)\} \cup \\ \quad \bigcup \zeta(\sigma, (v, m', v')) \end{array}$$

$$\begin{array}{l} \zeta(\sigma, (v, m', v')) = \\ \quad \{(v \cdot \sigma, \varepsilon, \nu(e \cdot v' \cdot \sigma)) \mid e \models m' \wedge e \in E\} \cup \\ \quad \{(\nu(rt \cdot v' \cdot \sigma), \varepsilon, v' \cdot \sigma) \mid rt \models (m' \wedge r)\} \cup \\ \quad \begin{array}{ll} \mathsf{if} & \neg\exists i.\, (0 \leq i \leq |\sigma| \wedge (v' \cdot \sigma)_i \models m') \\ \mathsf{then} & \chi(m', v' \cdot \sigma) \\ \mathsf{else} & \varnothing \end{array} \end{array}$$

- $A' = M \cup \{r\}$
- $\lambda' = \sigma \mapsto \begin{array}{l} \{\lambda_{\mathsf{Meth}}(\sigma_0)\} \cup \\ \mathsf{if}\ (|\sigma| = 1 \wedge \sigma_0 \models r)\ \mathsf{then}\ \{r\}\ \mathsf{else}\ \varnothing \end{array}$
- $E' = \{v \in E \mid \lambda_{\mathsf{Meth}}(v) \in M\}$.

Example Before discussing properties of the inlining algorithm, we first show a simple example. Suppose we have an applet as depicted in the left-hand column of Figure 1. Inlining this applet with the public method set $\{m\}$ results in the applet depicted in the right-hand column of Figure 1. Notice that all internal and public edges are preserved, while private method calls are replaced by two edges: to the entry and from the return point of the called method, respectively.

Properties We state several useful properties of the inlining algorithm. First of all, the inlining algorithm computes an applet having as interface the public interface of the original applet.

Proposition 5.2. *Let $\mathcal{A} : I$ be an applet and $M \subseteq I^+$ a set of public methods. The inlined applet $\alpha_M(\mathcal{A})$ has interface $I_{\alpha_M(\mathcal{A})} = (M, I^- - (I^+ - M))$,* i.e. *$\alpha_M(\mathcal{A}) : (M, I^- - (I^+ - M))$.*

By Proposition 4.2 we thus get:

$$b^M(\alpha_M(\mathcal{A})) = b(\alpha_M(\mathcal{A}))$$

Since the inlining transformation α_M only inlines methods not in M, α_{I^+} is the identity operation.

Proposition 5.3. *Let $\mathcal{A} : I$ be an applet. Then $\alpha_{I^+}(\mathcal{A}) = \mathcal{A}$.*

Finally, the inlining algorithm enjoys the following distributivity property.

Proposition 5.4. *Let $\mathcal{A} : I_\mathcal{A}$ and $\mathcal{B} : I_\mathcal{B}$ be applets, $M_\mathcal{A} \subseteq I^+_\mathcal{A}$ and $M_\mathcal{B} \subseteq I^+_\mathcal{B}$ be disjoint, and let $I^-_\mathcal{A} - I^+_\mathcal{A} \subseteq M_\mathcal{B}$ and $I^-_\mathcal{B} - I^+_\mathcal{B} \subseteq M_\mathcal{A}$. Then*

$$\alpha_{M_\mathcal{A} \cup M_\mathcal{B}}(\mathcal{A} \uplus \mathcal{B}) = \alpha_{M_\mathcal{A}}(\mathcal{A}) \uplus \alpha_{M_\mathcal{B}}(\mathcal{B})$$

Simulation Results. As already mentioned, the interface behaviour of the original applet is preserved by the inlining algorithm, *i.e.* every execution of the interface behaviour of $\mathcal{A}$ is an execution of the behaviour of $\alpha_M(\mathcal{A})$. This is due to the close correspondence between the interface behaviour of $\mathcal{A}$ and the structure of $\alpha_M(\mathcal{A})$. We provide an "inlining" transformation α'_M on the states of $b^M(\mathcal{A})$ by defining $\alpha'_M(v, \sigma) = (hd(\gamma), tl(\gamma))$, where $\gamma = \beta_M(v \cdot \sigma)$ and where $\beta_M(\sigma)$ denotes the sequence of normalised M-frames. Notice that we always have $hd(hd(\gamma)) = hd(v \cdot \sigma)$. We show that α'_M is a simulation relating the original interface behaviour with the inlined behaviour.

Theorem 5.5. *Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. Then $b^M(\mathcal{A}) \leq b(\alpha_M(\mathcal{A}))$.*

Proof. (Sketch) We show by co-induction that α'_M is a simulation between $b^M(\mathcal{A})$ and $b(\alpha_M(\mathcal{A}))$, *i.e.*, we show that (1) the valuations of (v, σ) in $b^M(\mathcal{A})$ and $\alpha'_M(v, \sigma)$ in $b(\alpha_M(\mathcal{A}))$ agree, and (2) if $(v, \sigma) \xrightarrow{l} (v', \sigma')$ in $b^M(\mathcal{A})$, then $\alpha'_M(v, \sigma) \xrightarrow{l} \alpha'_M(v', \sigma')$ in $b(\alpha_M(\mathcal{A}))$. The result then follows since α'_M maps the entry states of $b^M(\mathcal{A})$ to entry states of $b(\alpha_M(\mathcal{A}))$ (in fact, the entry states coincide, and α'_M maps every entry state to itself). It is easy to check that the valuations agree; for the proof that the transitions are simulated, we refer to [7]. □

Notice that in general we do not have behavioural simulation equivalence. The inlining construction introduces transfer edges for calls to and returns from private methods. Because of the latter, the behaviour of the inlined applet can contain a silent transition corresponding to a return from a private method (in the original applet), even when the inlined applet has not yet followed a silent transition corresponding to a call to this private method (in the original applet). The inlining thus introduces new behaviours. Notice however, that these new behaviours are only observable in applets which are not last-call recursive.

A set of methods is *recursive* if every method in the set contains a (reachable) call edge to some method in the set. A call edge is recursive if the calling and the called methods belong to some minimal (and thus, mutually) recursive method set. A program is called *last-call recursive* if from any destination node of any recursive call edge, only transfer edges are reachable. In addition, we shall assume that a return node is reachable from every such destination node.

For last-call recursive applets, we prove the reverse correspondence for observable behaviours.

Theorem 5.6. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b(\alpha_M(\mathcal{A})) \leq_w b^M(\mathcal{A})$.*

Proof. (Sketch) Consider a state (w, γ) in $b(\alpha_M(\mathcal{A}))$, where $\lambda_{\mathsf{Meth}}(hd(w)) \notin M$ and $hd(w) \models r$. For last-call recursive applets, the inlining transformation α_M has the property that for any such w, the nodes w' such that $\nu(hd(w) \cdot w') = w$ but $hd(w) \cdot w' \neq w$ and which are structurally reachable from w in $\alpha_M(\mathcal{A})$ form (together with w) a strongly connected component and are equivalent *w.r.t.* structural simulation. As a consequence, in $b(\alpha_M(\mathcal{A}))$, all states (w', γ) for a given γ also form a strongly connected component and are weak simulation equivalent. Modulo such "return" equivalence classes, we show by co-induction that $(\alpha'_M)^{-1}$ is a weak simulation between $b(\alpha_M(\mathcal{A}))$ and $b^M(\mathcal{A})$. More exactly, we show that (1) the valuations of $\alpha'_M(v, \sigma)$ and (v, σ) agree, and (2) if $\alpha'_M(v, \sigma) \xrightarrow{l} (w', \gamma')$ is a transition in $b(\alpha_M(\mathcal{A}))$ other than a (silent) transition within a return equivalence class, then $(v, \sigma) \overset{l}{\Rightarrow} (v', \sigma')$ in $b^M(\mathcal{A})$ for some $v'$and σ' such that $\alpha'_M(v', \sigma') = (w', \gamma')$ (in most cases we even show the corresponding strong transition). The result then follows since α'_M maps entry states of $b(\alpha_M(\mathcal{A}))$ to entry states of $b^M(\mathcal{A})$. Again, it is easy to check that the valuations agree; for the proof that the transitions are simulated, we refer to [7]. □

Since weak simulation contains simulation we have the following.

Corollary 5.7. *Let $\mathcal{A} : I$ be a closed last-call recursive applet, and let $M \subseteq I^+$. Then $b^M(\mathcal{A}) \equiv_w b(\alpha_M(\mathcal{A}))$.*

6 Interface Abstraction and Compositional Reasoning

Using the results obtained above, we can state several verification principles that can be used to prove properties of applet interface behaviour. We first present two abstraction principles, and then show how these can be combined with our compositional verification principle from [13] to support the improved scenario for secure post–issuance loading of applets on smart devices presented in the Introduction.

Interface Abstraction. Let $\mathcal{A} : I$ be a closed applet, and let $M \subseteq I^+$. With the results established above, we can justify the following abstraction principle (abstract), where ψ is a behavioural interface formula.

$$\frac{\alpha_M(\mathcal{A}) \models_b \psi}{\mathcal{A} \models_b^M \psi}$$

Theorem 6.1. *Rule* (abstract) *is sound.*

Proof. Follows from the definition of behavioural satisfaction, Theorem 5.5, Theorem 2.3, and the definition of behavioural interface satisfaction. □

When $\mathcal{A}$ has last-call recursion, we can even provide a faithful abstraction principle (weak-abstract) for properties of the observable behaviour by using transformation δ from Section 2.

$$\frac{\alpha_M(\mathcal{A}) \models_b \delta(\psi)}{\mathcal{A} \models_{b,w}^M \psi}$$

Theorem 6.2. *Rule* (weak-abstract) *is sound and complete.*

Proof. Follows from the definition of behavioural satisfaction, Proposition 2.5, Corollary 5.7, Theorem 2.4, and the definition of weak behavioural interface satisfaction, all of which are equivalences. □

Compositional Reasoning. In earlier work [13] we presented the following sound and complete compositional verification principle (compos):

$$\frac{\mathcal{A} \models_s \sigma \qquad \mathcal{M}ax_{I_\mathcal{A}}(\sigma) \uplus \mathcal{B} \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b \psi} \ \mathcal{A} : I_\mathcal{A}$$

Here $\mathcal{A}$ and $\mathcal{B}$ are applets, such that $\mathcal{A} \uplus \mathcal{B}$ is a closed applet. Further, σ is a formula in simulation logic on the structural level (*i.e.* boxes are interpreted over the edges in the call graph), while ψ is a property at the behavioural level[5]. Finally, $Max_{I_\mathcal{A}}(\sigma)$ is a construction described in [13] yielding a so-called *maximal applet w.r.t.* σ and $I_\mathcal{A}$, *i.e.* an applet with interface $I_\mathcal{A}$ that simulates all other applets with this interface satisfying property σ.

[5]A similar principle exists if ψ is a structural property, since behavioural simulation contains structural simulation.

Observe that the maximal model construction can only be applied if the complete interface $I_\mathcal{A}$ of applet $\mathcal{A}$ is known. The correctness of a property decomposition can thus only be established for applets with a known interface, and knowledge of the public interface only is hence not sufficient. To allow compositional verification of public interface properties, we combine the above rule with the abstraction principle (abstract) to obtain the following abstract compositional verification principle (abstract-compos):

$$\frac{\alpha_M(\mathcal{A}) \models_s \sigma \qquad \mathcal{M}ax_{I_{\alpha_M(\mathcal{A})}}(\sigma) \uplus \mathcal{B} \models_b \psi}{\mathcal{A} \uplus \mathcal{B} \models_b^{M \cup I_\mathcal{B}^+} \psi} \ \begin{array}{l}\mathcal{A} : I_\mathcal{A},\\ I_\mathcal{B}^- - I_\mathcal{B}^+ \subseteq M\end{array}$$

Theorem 6.3. *Rule* (abstract-compos) *is sound.*

Proof. Follows from the abstraction and the compositional verification principle, plus Propositions 5.3 and 5.4. □

The improved scenario for secure post–issuance loading of applets presented in the Introduction is based on the verification principle embodied by this rule. Notice that the interface of required methods that is used for the maximal model construction uses $I_\mathcal{A}^- - I_\mathcal{A}^+$. Typically, this will correspond to the public interface of $\mathcal{B}$, and for each implementation of $\mathcal{A}$ it should be checked whether it respects this public interface of $\mathcal{B}$.

Finally, similarly as for the abstraction principle, we can state a faithful compositional verification principle (weak-abstract-compos) for properties of the observable interface behaviour of applets which are last-call recursive.

$$\frac{\alpha_M(\mathcal{A}) \models_s \sigma \quad Max_{I_{\alpha_M(\mathcal{A})}}(\sigma) \uplus \mathcal{B} \models_b \delta(\psi)}{\mathcal{A} \uplus \mathcal{B} \models_{b,w}^{M \cup I_\mathcal{B}^+} \psi} \ \begin{array}{l}\mathcal{A} : I_\mathcal{A},\\ I_\mathcal{B}^- - I_\mathcal{B}^+ \subseteq M\end{array}$$

Theorem 6.4. *Rule* (weak-abstract-compos) *is sound and complete.*

7 Practical Impact of Inlining

As explained above, we are interested in studying the abstract behaviour of applets, because in a truly compositional setting nothing is known about the different components, except (some properties of) their interface behaviour. For a newly downloaded applet we only require that it implements the shareable interface; we do not put any restrictions on *how* it implements this shareable interface, except that the implementation should respect the global security requirements. Studying compositional verification at the abstract level allows to specify the local and global properties at the abstract level, without taking any implementation

	$\mathcal{M}ax(\sigma_L)$	$\mathcal{M}ax(\sigma_L)$ in [8]	$\mathcal{M}ax(\sigma_P)$	$\mathcal{M}ax(\sigma_P)$ in [8]
#nodes	8	474	8	2786
#edges	120	277 700	88	603 128
constr. time	0.05 s.	25 min.	0.05 s.	13 hrs.

Table 2. Size and timing for maximal model construction

details into account. Moreover, when considering shareable interfaces only, the maximal models that we compute to verify the decomposition of the global property into the local ones are significantly reduced in size, making the verification much more efficient.

In order to show the impact of abstraction and inlining on a realistic case study, this section revisits the electronic purse case study [8], specifying an illicit interaction between applets *Purse* and *Loyalty*. In the original case study we computed maximal applets using the implementation interfaces (containing about 300 methods per applet). This was time-consuming (25 mins. to 13 hrs.) and moreover, the size of the outcome was so large that verification was unfeasible. However, the public interfaces (*i.e.* the shareable interfaces) of these applets both provide only 4 methods. If we refer to the shareable interfaces as SI_P (methods provided by *Purse* for *Purse* and *Loyalty*) and SI_L (*Loyalty* for *Purse* and *Loyalty*), respectively, we can identify the following public interfaces: $(SI_P, SI_P \cup SI_L)$ for *Purse*, and $(SI_L, SI_P \cup SI_L)$ for *Loyalty*.

We use the tool set described in [8], plus an implementation of the inlining algorithm in Ocaml to redo the case study at the abstract level. For convenience we repeat the global and local specifications, but this time specified at the interface level; for further motivations we refer to [8].

The global specification ψ says that a call to *Loyalty.logFull* does not trigger any calls to any other loyalty, including indirect communications, via the *Purse*. The specification uses several abbreviations for readability (where $\mathcal{A}$ is an applet such that $\mathcal{A} : (I^+, I^-)$ and M a set of methods).

$$\begin{array}{rcl} \mathit{Always}\ \phi & = & \nu Z.\ \phi \wedge [L_b]Z \\ \mathit{Within}\ m\ \phi & = & \neg m \vee (\mathit{Always}\ \phi) \\ \mathit{CanNotCall}\ \mathcal{A}\ M & = & \bigwedge_{m \in I^+} \bigwedge_{m' \in M} [m\ \mathit{call}\ m']\,\mathsf{false} \end{array}$$

$$\begin{array}{ll} (\psi) & \mathit{Within}\ loyalty.logFull \\ & \quad \mathit{CanNotCall}\ \mathit{Loyalty}\ SI_L \wedge \\ & \quad \mathit{CanNotCall}\ Purse\ SI_L \end{array}$$

For the *Loyalty* applet we exclude any external calls, except those to the methods *Purse.isThereTransaction* and *Purse.getTransaction* (σ_L). For the *Purse* applet we specify that both these methods do not make any external calls (σ_P). Again we use several abbreviations.

$$\begin{array}{rcl} \mathit{Everywhere}\ \sigma & = & \nu Z.\ \sigma \wedge [\varepsilon, I^-]Z \\ M\ \mathit{HasNoCallsTo}\ M' & = & \left(\bigwedge_{m \in M} \neg m\right) \vee \\ & & \mathit{Everywhere}\ [M']\,\mathsf{false} \\ \mathit{HasNoOutsideCalls}\ M & = & M\ \mathit{HasNoCallsTo}\ (I^- \setminus M) \end{array}$$

$$\begin{array}{ll} (\sigma_L) & \mathit{loyalty.logFull}\ \mathit{HasNoCallsTo} \\ & \quad (SI_P \cup SI_L)/\{Purse.isThereTransaction, \\ & \qquad Purse.getTransaction\} \\ (\sigma_P) & \mathit{HasNoOutsideCalls}\ Purse.\mathit{isThereTransaction} \wedge \\ & \mathit{HasNoOutsideCalls}\ Purse.getTransaction \end{array}$$

These specifications refer to the inlined versions of the applets. To exclude external calls from a method of an inlined applet is equivalent to excluding *transitive* external calls made from the public method with the same name in the original applet. Notice that such a property is not directly expressible in our logic (cf. [8]).

To redo the case study at the abstract level, we take the following steps (where P and L denote implementations of *Purse* and *Loyalty*, respectively):

- compute $\mathcal{M}ax_{(SI_P, SI_P \cup SI_L)}(\sigma_P)$ and $\mathcal{M}ax_{(SI_L, SI_L \cup SI_P)}(\sigma_L)$ using the Maximal Model constructor [13, 8];
- model check $\mathcal{M}ax_{(SI_P, SI_P \cup SI_L)}(\sigma_P) \uplus \mathcal{M}ax_{(SI_L, SI_L \cup SI_P)}(\sigma_L) \models_b \delta(\psi)$ using a prototype implementation of a model checker for PDAs;
- compute $\alpha_{SI_P}(P)$ and $\alpha_{SI_L}(L)$ using the inlining algorithm; and
- model check $\alpha_{SI_P}(P) \models_s \sigma_P$ and $\alpha_{SI_L}(L) \models_s \sigma_L$ using CWB [4].

Table 2 compares the outcome and timing for the maximal model construction with the corresponding step in the original case study. Checking the correctness of the decomposition took approximately 5 seconds. The inlining algorithm took 0.6 seconds on both *Loyalty* and *Purse*. Even though theoretically the worst-case blowup in the number of nodes of the inlined applets, determined by the number of normal M-frames, is exponential in the number of private methods, in practice this is not likely to happen. In our case, we even observed a reduction in size of the graphs, due to the fact that the inlining focuses on interaction with other applets, and thus any code that is executed only when the applet is selected and receives commands from the runtime environment, is left out by the inlining. Verifying the local properties of the inlined applets of *Loyalty* and *Purse* took approximately 15 and 10 seconds, respectively.

8 Conclusions

In this paper, we propose a notion of interface behaviour of program components which abstracts from the internal, private behaviour. Based on this notion, behavioural properties can be specified at the public interface level without requiring knowledge about the implementation. Focusing on interface behaviour is significant from a methodological, software engineering point of view. In particular, it supports compositional verification by allowing global, program–wide properties to be inferred from the interface properties of the not yet available components.

We propose a program transformation based on inlining of private methods, and show that it preserves the interface behaviour. The inlining transformation reduces the number of methods of a program to the number of its public methods. This is a necessary condition for applying the maximal model construction from [8, 13] in a truly compositional manner, and gives rise to an improved scenario for secure post–issuance loading of applets on smart devices. The reduction in the number of methods resulting from the interface abstraction drastically improves the performance of the maximal model construction which is of exponential worst–case complexity. Finally, we observe that some natural structural properties are only directly expressible as properties of the inlined applet.

Acknowledgments

We would like to thank Gennady Chugunov for helping us redoing the verifications, and Christoph Sprenger for numerous suggestions for improvement on an earlier draft.

References

[1] R. Alur, K. Etessami, and P. Madhusudan. A temporal logic for nested calls and returns. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Analysis and Construction of Software, TACAS 04*, number 2998 in LNCS, pages 467–481. Springer, 2004.

[2] 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.

[3] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[4] 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.

[5] 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.

[6] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. on Prog. Lang. & Syst.*, 16(3):843–871, 1994.

[7] D. Gurov and M. Huisman. Abstraction over public interfaces. Technical Report RR-5330, INRIA, 2004.

[8] M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, FASE 2004*, number 2984 in LNCS, pages 84–98. Springer, 2004.

[9] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security policies. In *IEEE Symposium on Research in Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.

[10] O. Kaser and C.R. Ramakrishnan. Evaluating inlining techniques. *Journal of Computer Languages (JCL)*, 24(2):55–72, 1998.

[11] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27:333–354, 1983.

[12] C. Sprenger, D. Gurov, and M. Huisman. Simulation logic, applets and compositional verification. Technical Report RR-4890, INRIA, 2003.

[13] C. Sprenger, D. Gurov, and M. Huisman. Compositional verification for secure loading of smart card applets. In *Formal Methods and Models for Co-Design (Memocode 2004)*, pages 211–222. IEEE, 2004.

[14] C. Stirling. *Modal and Temporal Logics of Processes*. Springer, 2001.