USENIX Association

Proceedings of the 5th Smart Card Research and Advanced Application Conference

San Jose, California, USA
November 21–22, 2002

© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Model Checking of Multi-Applet JavaCard Applications*

Gennady Chugunov[1] Lars-Åke Fredlund[1] Dilian Gurov[2]

[1]Swedish Institute of Computer Science

[2]Department of Microelectronics and Information Technology, Royal Institute of Technology (KTH)

Abstract

The paper describes a framework for model checking JavaCard applets on the bytecode level. From a set of JavaCard applets we extract their method call graphs using a static analysis tool. The resulting structure is translated into a pushdown system for which the model checking problem for Linear Temporal Logic (LTL) is decidable, and for which there are efficient model checking tools available. The model checking approach of the paper is tailored to the analysis of inter applet (intra card) communications and we demonstrate it using a prototypical example of a purse applet and a set of loyalty applets.

1 Introduction

Smart cards have come to play an ever increasing role in our lives. We use them in electronic banking, to keep health care data, for mobile telephony, and in many other applications. The most important aspect of smartcards is their security; users and card issuers have to agree that the level of security provided by a smartcard platform is enough to prevent malicious agents from abusing their trust in a card application.

Since the number of smartcard applications is growing rapidly, it is natural to provide smartcards with the possibility of accommodating multiple applications, and the possibility to delete or add new applications after the card has been issued. Furthermore, such multi-application smartcards allow partner applications to cooperate and exchange data. Popular applications of multi-application cards are partner loyalty programs, mobile telephone to banking partnership programs, etc. The JavaCard platform [12] is one platform for building such multi-application smartcards. It is based on a subset of Java tailored to the task of embedding on a smartcard. The current standard omits many of the features of Java such as concurrency through threads, garbage collection, and many API functions but has a notion of applets to support multiple applications.

One important aspect which distinguishes multi-applet JavaCards from single-applet ones is the support for inter-applet communication via method calls. Communication naturally comes at a price: applets must guard against illicit invocations of their public methods from unwarranted applets, and from leakage of data to third parties. Even if a multi-applet application were to be proved safe, there still exists the possibility of new unsafe applets being loaded onto the card post–verification. The JavaCard platform provides features to partially address these security concerns. Apart from a Java-style byte code verifier, which in the current generation of JavaCard smartcards is typically located off–card, there is a concept of a communication firewall that by default prohibits applets from communicating with each other. To enable communication to flow between applets, a recipient applet has to explicitly permit calls from the caller applet.

Such checks as above are static in nature, e.g., method calls are always allowed, or they are never allowed. The work reported here in contrast permits to begin to characterise the temporal restrictions of inter-applet communications. In the formulation of such restrictions we consider a situation when a set of applets have been loaded onto a smartcard,

*The research has been conducted within the VerifiCard project with financial support from the IST programme of the European Union.

and formulate properties in Linear Temporal Logic (LTL) regarding inter–applet communications (in addition to properties about intra–applet method calls and API usage).

To provide a semantic bridge between multi-applet programs and the temporal logic specification language, we use the abstract notion of a program graph, capturing the control flow of programs with procedures/methods, and which can be efficiently computed. The behaviour of such program graphs is defined through the notion of pushdown systems, which provide a natural execution model for programs with methods (and possibly recursion), and for which completely automatic model checkers for LTL exist.

In more detail the model checking proceeds as follows. First the method call graphs of a set of JavaCard applets are obtained using a Java byte code analysis tool [13] developed at INRIA Rennes, which we have adapted for JavaCard. The analysis is performed on a class basis. As a consequence individual applet instances cannot be reasoned about; correctness properties concern activation of methods of classes extending the JavaCard `Applet` class, rather than activation of methods of an applet instance. Further details and limitations of this static analysis procedure are discussed in Section 2.

The resulting method call graphs are translated into pushdown systems, a natural execution model for programs with recursion. Essentially a pushdown system is a pair of a control location with a stack of stack symbols. In our encoding we use a single control location and let the stack symbols represent the program points of the underlying JavaCard applets. The details of the translation are elaborated in Section 3.1.

For pushdown systems the model checking procedure for Linear Temporal Logic (LTL) is decidable and of polynomial complexity in the size of the system [3, 9, 7]. The atomic predicates of the logic, tailored to JavaCard, are the program points themselves and predicates expressing class and package membership of program points. The Moped model checker [8] is used to check LTL properties of pushdown systems. Sections 3.2,3.3 and 3.4 describes the logic and our use of the Moped tool in further detail.

To motivate and demonstrate our approach we have selected a prototypical JavaCard example: a purse applet stores money, and interacts with loyalty applets on receiving a purchase order. A loyalty applet can have agreements with other applets, and can thus in turn communicate with another applet on receiving information about a purse transaction. In Section 4 we demonstrate the effectiveness of our approach in analysing such inter-applet communication patterns.

There exists by now a growing number of related work concerning model checking Java (or JavaCard), or more general formal analysis of JavaCard applications; below we will mention a few of them.

The Compaq Extended Static Checker for Java (ESC/Java) [14], developed at the Compaq Systems Research Center (SRC), is a programming tool for finding errors in Java programs. ESC/Java includes an annotation language with which programmers can express design decisions using light-weight specifications. Checking is neither sound nor complete, but can yield informative warning messages[1]. A case study in the context of JavaCard, based on the Gemplus purse applet, is presented in [5].

The first version of the Java PathFinder [10], JPF, was a translator from a subset of Java 1.0 to PROMELA, the programming language of the Spin model checker. A similar translator tool from Java to PROMELA (actually the variant of PROMELA for the dSpin tool) is reported in [11]. The Java Pathfinder tool is especially suited for analyzing multi-threaded Java applications, where normal testing usually falls short. The tool can find deadlocks and violations of boolean assertions stated by the programmer in a special assertion language. A second version of the tool reportedly works directly on bytecode and has support for garbage collection[2].

The Bandera Project [6] aims to develop techniques and tools for automated reasoning about Java based software system behavior, and to apply these tools to construct high-confidence mission-critical software. Automated reasoning is achieved by (1) mechanically creating high-level models of software systems using abstract interpretation and partial evaluation technologies, and then (2) employing model-checking techniques to automatically verify that software specifications are satisfied by the model[3].

[1] `http://research.compaq.com/SRC/esc/`
[2] `http://ase.arc.nasa.gov/jpf/`
[3] `http://www.cis.ksu.edu/santos/bandera/`

In [2] an approach is presented for checking properties of multi-applet interactions of JavaCards based on associating security levels to applets and applet data, and to thus detect illegal flow of information between applets. Technically the approach requires building abstract models by hand from byte code, and then to check them automatically using the SMV model checker.

Our work is related to the program verification approach of [13] which is based on method call graphs. The operational semantics of the graphs, however, is given there directly through a set of transition rules (rather than through pushdown systems), and security properties are expressed as call-stack invariants. Following a similar program representation, a compositional account is given in [1], where a compositional proof system for inferring temporal properties of a multi-applet program from the properties of the individual applets is presented.

2 Constructing Method Call Graphs

We use an external static analysis tool, developed for a Java verification framework [13], to generate call graphs which abstract from everything (such as data variables, and parameters to method calls) but the presence and order of method calls inside method bodies. The analysis tool performs a safe over-approximation (with regards to preservation of LTL safety properties) in the sense that call edges may be present in the result call graph even if they cannot be invoked at runtime, but the opposite does not hold. For instance, when the static analysis cannot determine which class method is invoked in a method call, typically due to subtyping, then a call edge is generated to a target method in every possible class, thus increasing the nondeterminism in the generated call graph. The static analysis tool generates graphs with information about exceptional behaviours. In this work exceptional edges, and nodes, are translated into nondeterministic constructs thus effectively increasing the non-determinism in program behaviour in a conservative fashion.

The call graph generation is also conservative with respect to the JavaCard firewall mechanism, which is not considered during static analysis. That is, a method call that at runtime will fail the security checks of the JavaCard runtime environment will nevertheless invariably be included in the method call graphs.

Analysis starts from a set of JavaCard classes, which should include the implementation of all on-card applets. To refine the analysis, and to permit analysis of JavaCard API usage, the API classes of SUN's Java Card Development Kit (version 2.1.2) are included in the method call generation. The result of analysis is a set of method call graphs.

2.0.1 Method Call Graphs

The methods M are partitioned into classes C, which are themselves partitioned into packages P. We assume the usual Java naming conventions with fully qualified names, i.e., a class has a name *Package.identifier* and a method has a name *Class.identifier*.

Definition 1 (Method Graph, adapted from [13]). A *method graph* is a tuple

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

such that:

(i) V_m are the *program points* of m,

(ii) $\rightarrow_m \subseteq V_m \times V_m$ are the *transfer edges* of m, and

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

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

We assume the program point sets V_m to be pairwise disjoint. The program points of the program is the set $V \triangleq \bigcup_{m \in M} V_m$.

The program point type indicates whether (`entry`) a node is the entry point of a method, (`seq`) a node in which no method call or return takes place, (`call`) a node from which a method call takes place, or (`return`) a node in which the execution of the method finishes and control flow returns to the calling method.

For convenience, we introduce the predicates

$$\begin{array}{rcl} v : t & \triangleq & \lambda_m(v) = t \text{ for } t \in T \\ v : \mathsf{loc}\, m & \triangleq & v \in V_m \\ v : \mathsf{entry}\ m & \triangleq & v : \mathsf{entry} \ \wedge\ v : \mathsf{loc}\, m \\ v : \mathsf{return}\ m & \triangleq & v : \mathsf{return} \ \wedge\ v : \mathsf{loc}\, m \\ v : \mathsf{class}\ c & \triangleq & \exists m.\ v : \mathsf{loc}\, m \ \wedge\ m \in c \\ v : \mathsf{package}\ p & \triangleq & \exists c.\ v : \mathsf{class}\ c \ \wedge\ c \in p \end{array}$$

We further define a predicate $v : \mathsf{api}$, which holds if the program point v occurs in a method in a JavaCard API package (for standard JavaCard this corresponds to one of `java.lang`, `javacard.framework`, `javacard.security` or `javacardx.crypto`).

3 Model Checking Method Call Graphs

3.1 Pushdown Systems

Pushdown systems provide a natural execution model for programs with recursion. They form a well-studied class of infinite-state systems for which many important problems like equivalence checking and model checking are decidable [4].

Definition 2 (PDS, from [7]). A *pushdown system (PDS)* is a tuple

$$\mathcal{P} \triangleq (P, \Gamma, \Delta)$$

where:

(i) P is a finite set of *control locations*;

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

(iii) $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^\star)$ is a finite set of *rewrite rules* of the shape $\langle p, \gamma \rangle \to \langle q, \sigma \rangle$.

The set $P \times \Gamma^\star$ are the *configurations* of $\mathcal{P}$. If $\langle p, \gamma \rangle \to \langle q, \sigma \rangle$ is a rewrite rule of $\mathcal{P}$, then for each $\omega \in \Gamma^\star$ the configuration $\langle q, \sigma \cdot \omega \rangle$ is an *immediate successor* of the configuration $\langle p, \gamma \cdot \omega \rangle$. A *run* of $\mathcal{P}$ is a sequence $\rho = \langle p_0, \sigma_0 \rangle \langle p_1, \sigma_1 \rangle \langle p_2, \sigma_2 \rangle \cdots$, such that for all i, $\langle p_{i+1}, \sigma_{i+1} \rangle$ is an immediate successor of $\langle p_i, \sigma_i \rangle$.

We now define how a set of methods M induces a PDS.

Definition 3 (Induced PDS, formalising [8]). A set of methods M *induces* a PDS

$$\mathcal{P} \triangleq (P, \Gamma, \Delta)$$

as follows:

(i) P consists of the single control location p;

(ii) Γ is the set V of program points;

(iii) Δ is the set $\bigcup_{m \in M} \bigcup_{v \in V_m} \mathrm{Prod}(v)$, where $\mathrm{Prod}(v)$ is a set of rewrite rules defined as:

$$\begin{cases} \{\langle p, v \rangle \to \langle p, v' \rangle \mid v \to_m v'\} \\ \quad \text{if } v : \mathsf{entry} \text{ or } v : \mathsf{seq} \\ \\ \bigcup_{m' \in \mu_m(v)} \left\{ \begin{array}{c} \langle p, v \rangle \to \langle p, v' \cdot v'' \rangle \mid \\ v' : \mathsf{entry}\ m', v \to_m v'' \end{array} \right\} \\ \quad \text{if } v : \mathsf{call} \\ \\ \{\langle p, v \rangle \to \langle p, \epsilon \rangle\} \\ \quad \text{if } v : \mathsf{return} \end{cases}$$

The rewrite rules of the pushdown system can be interpreted as simply manipulating the calling stack of the program from which the PDS was obtained. Given a configuration $c \equiv \langle p, v \cdot \sigma \rangle$ let $\mathsf{point}\ (c) \triangleq v$.

3.2 Specification Language

Our specification language is linear temporal logic (LTL), with program point predicates p as atomic propositions but omitting the type predicate $v : t$. The choice of linear temporal logic as the specification language, instead of for instance the modal μ-calculus for which the model checking problem for our encoding into pushdown systems is also efficiently decidable, was solely motivated by the existence of the efficient model checker Moped [8] for LTL.

The operators of the logic are the standard ones. If ϕ and ψ are formulas then so are $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, $\mathcal{X}\phi$ and $\phi\ \mathcal{U}\ \psi$. The meaning of formulas is defined with respect to runs of infinite length $r \equiv c_0 c_1 c_2 \ldots$. We let r_i denote the suffix of r starting in configuration c_i. Then satisfaction $r \models \phi$ of a formula ϕ by a run r is defined as:

$$
\begin{array}{rll}
r \models p & \text{iff} & \mathsf{point}\ (c_0) : p \\
r \models \neg\phi & \text{iff} & \text{not } r \models \phi \\
r \models \phi \wedge \psi & \text{iff} & r \models \phi \text{ and } r \models \psi \\
r \models \phi \vee \psi & \text{iff} & r \models \phi \text{ or } r \models \psi \\
r \models \mathcal{X}\ \phi & \text{iff} & r_1 \models \phi \\
r \models \phi\ \mathcal{U}\ \psi & \text{iff} & \text{there is an } i \geq 0 \text{ such that} \\
 & & r_i \models \psi \text{ and } r_j \models \phi \\
 & & \text{for all } 0 \leq j < i
\end{array}
$$

Henceforth let false abbreviate $p \wedge \neg p$ for some atomic predicate p, true abbreviate $\neg\mathsf{false}$, $\phi \Rightarrow \psi$ abbreviate $\neg\phi \vee \psi$, and $\mathsf{next}\ \ \phi$ abbreviate $\mathcal{X}\ \phi$ and $\phi\ \ \mathsf{until}\ \ \psi$ abbreviate $\phi\ \ \mathcal{U}\ \ \psi$. Further define $\mathsf{eventually}\ \ \phi \ \ \triangleq\ \ \mathsf{true}\ \ \mathcal{U}\ \ \phi$ and $\mathsf{always}\ \phi \triangleq \neg\,(\mathsf{eventually}\ \neg\phi)$. The weak until operator ϕ $\mathsf{weakuntil}$ ψ abbreviates ϕ until $\psi \vee \mathsf{always}\ \phi$. Finally let $\mathsf{never}\ \phi \triangleq \mathsf{always}\ \neg\phi$.

Given a PDS *pds* let the notation $m \vdash \phi$ express the judgement that all runs starting in the entry program point of the method m satisfy ϕ. More formally:

Definition 4 (Model Checking a Method Call). Given a PDS *pds* with the single control location p and a method m, the judgement $m \vdash \phi$ is valid iff for every run r of the PDS pds' from the initial configuration $\langle p, v \cdot m\_\mathtt{loop}\rangle$, $r \models \phi$ holds, where v is the entry program point of method m (i.e. $v : \mathsf{entry}\ m$), and pds' is the PDS *pds* extended with the fresh stack symbol $m\_\mathtt{loop}$ and the single rewrite rule $\langle p, m\_\mathtt{loop}\rangle \rightarrow \langle p, m\_\mathtt{loop}\rangle$ to achieve infinite runs.

The definition of a judgement $m \vdash \phi$ is motivated by the Moped tool which implements an algorithm for checking an initial configuration against an LTL formula.

3.3 Specification Patterns

As in the Bandera project [6] specification patterns are used to facilitate formulating correctness properties. These specification patterns concern temporal properties of method invocations, and are either *temporal patterns* or *judgement patterns* concerning the invocation of a particular method. Below a set of patterns that we have defined, and which are commonly used, are given.

To express that *within the call of a method m the property ϕ holds* the judgment pattern

$$\mathsf{Within}\ m\ \phi\ \triangleq\ m \vdash \phi$$

is used. The property that *a call to m_1 never triggers method m_2* can be specified as:

$$
\begin{array}{l}
m_1\ \mathsf{never\ triggers}\ m_2 \\
\quad \triangleq\ \mathsf{Within}\ m_1\ (\neg(\mathsf{eventually\ loc}\ m_2)) \\
\quad \equiv\ \mathsf{Within}\ m_1\ (\mathsf{never\ loc}\ m_2)
\end{array}
$$

Next define the temporal patterns (formulas) (i) m_2 after m_1, i.e., m_2 can only be called after a call to m_1; (ii) m_2 $\mathsf{through}$ m_1, i.e., m_2 can only be called from m_1; (iii) m_2 from m_1, i.e., m_2 can only be called directly from m_1; and (iv) m_1 $\mathsf{excludes}$ m_1, i.e., when m_1 is called this excludes the possibility that m_2 will later be called; (v) p $\mathsf{cannotCall}$ m, i.e., the method m cannot be directly called from any method in package p.

$$
\begin{array}{l}
m_2\ \mathsf{after}\ m_1 \\
\quad \triangleq\ (\mathsf{never\ loc}\ m_2)\ \ \mathsf{weakuntil\ loc}\ m_1
\end{array}
$$

$$
\begin{array}{l}
m_1\ \mathsf{excludes}\ m_2 \\
\quad \triangleq\ (\mathsf{eventually\ loc}\ m_1)\ \Rightarrow\ \mathsf{never\ loc}\ m_2
\end{array}
$$

$$
\begin{array}{l}
m_2\ \mathsf{from}\ m_1 \\
\quad \triangleq\ \begin{array}{l} \mathsf{always}\ (\neg\,(\mathsf{loc}\ m_1 \vee \mathsf{loc}\ m_2) \Rightarrow \mathsf{next}\ \neg\mathsf{loc}\ m_2) \\ \wedge\ \neg\mathsf{loc}\ m_2 \end{array}
\end{array}
$$

$$
\begin{array}{l}
m_2\ \mathsf{through}\ m_1 \\
\quad \triangleq\ \begin{array}{l} \neg\mathsf{loc}\ m_2\ \mathsf{weakuntil\ loc}\ m_1 \\ \wedge \left(\begin{array}{l} \mathsf{always\ return}\ m_1 \Rightarrow \\ \quad \mathsf{next}\ \ (\neg\mathsf{loc}\ m_2\ \mathsf{weakuntil\ loc}\ m_1) \end{array} \right) \end{array}
\end{array}
$$

$$
\begin{array}{l}
p\ \mathsf{cannotCall}\ m \\
\quad \triangleq\ \mathsf{always}\,(\mathsf{package}\ p \Rightarrow \mathsf{next}\ \neg\mathsf{loc}\ m)
\end{array}
$$

The intuitive idea of the formulation of m_2 from m_1 is to express that the current program point can be in method m_2 only because of a direct call from m_1, or because it was already in m_2, and initially the program point is not in m_2.

The above patterns can be combined with the Within pattern. For example,

$$\mathsf{Within}\ m_1\ \ (m_3\ \mathsf{after}\ m_2)$$

expresses that during a call to m_1 the method m_3 will be called only after calling m_2.

An alternative technique for expressing correctness properties of behaviours of programs of stack-based languages is to use stack inspection techniques [13]. Essentially these techniques express constraints on the set of all possible runtime stacks. Note however that for instance the after property above cannot directly be coded as a stack inspection property since the calls to m_1 and m_2 need not be concurrent.

3.4 A Tool for Model Checking Pushdown Systems

The Moped tool [8] can check a pushdown system, from an initial configuration, against an LTL formula where the atomic predicates consists of a set of atomic symbols that checks the identity of the top stack symbol or the control location (i.e., simply checks name equality). In case the LTL formula is falsified a reduced pushdown system constructed from the original one, that also falsifies the LTL formula, is presented as diagnostic information.

To represent the non-identity atomic predicates (e.g., package, entry, ...) as "Moped LTL formulas" a number of options are possible. Consider for instance the package atomic predicate. A direct representation of the predicate in Moped LTL would consist of a disjunction over all the program points in any class in the package.

An alternative representation strategy is to enrich the translation from a call graph to a pushdown system. Since Moped provides boolean variables we could represent the current package identity encoded in a set of boolean variables in the pushdown system. These variables would then be updated for every rewrite rule that crosses package boundaries. Finally the representation of the package predicate itself would consist of a simple boolean condition.

We have instead opted to extend the Moped tool with atomic predicates that can match a control location, or the top stack symbol, against a regular expression. These predicates check the syntactic shape of the symbol being tested.

Consider the naming of program points of a method m by the call graph construction. Its entry program point will be named m_entry, its (unique) return program point will be named m_exit, and all other program points in m are of the form m_n where n is a natural number.

With these conventions in place the atomic predicates can be represented in "regular expression Moped" as indicated below:

$$\begin{array}{rcl}
\mathsf{loc}\ m & \triangleq & m\texttt{_.*}\\
\mathsf{entry}\ m & \triangleq & m\texttt{_entry}\\
\mathsf{return}\ m & \triangleq & m\texttt{_exit}\\
\mathsf{class}\ c & \triangleq & c\texttt{\textbackslash..*_.*}\\
\mathsf{package}\ p & \triangleq & p\texttt{\textbackslash..*\textbackslash..*_.*}
\end{array}$$

In the encoding it is assumed that the dot symbol '.' has to be quoted using a backslash character inside a regular expression to represent itself, rather than representing any character.

So called wildcards can be used in a regular expression to achieve a limited form of quantification over program points. The static analysis tool, for instance, gives the name $p.c.$<init> to an object constructor method $p.c$. Thus, whether the current program point is in any object constructor can be tested by the regular expression predicate .*\..*\.<init>_.*. As a further example, the api predicate, which recognises control points inside an API function, can be defined '(java\.lang|javacard\..*|javacardx\..*).*'.

4 Example

The model checking of JavaCard applets will be illustrated with an example; a modification of the purse example from SUN's JavaCard Development Kit (version 2.1.2). This example is a prototypical purse and loyalty smartcard application, which comprises around 1430 lines of JavaCard code.

To understand the example it is helpful to recall the execution characteristics of JavaCard applets (language version 2.1.1). An interaction with the card (after installation of an applet, and its selection) is initiated through calling its process method. Inter-applet communications, crossing package borders, are controlled by the JavaCard firewall mechanism and take place through special interface objects. The methods of such interface objects are indicated in Figure 1.

The purse applet keeps a balance that is updated upon requests from the environment. Purse transactions, whether successful or not, are logged to a transaction log. The operations of updating the balance, logging the new transaction and updating the

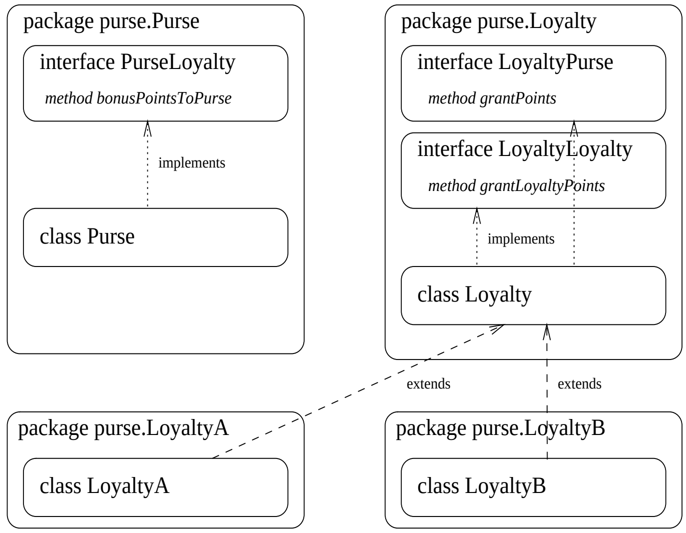

Figure 1: Purse Class Diagram

transaction number are made atomic through use of the transaction facility of JavaCard.

Upon completion of a new purse transaction the purse applet notifies subsidiary loyalty applets via the interface method `grantPoints`. These are applets that should be notified of the balance update so that they, for example, can award loyalty points. A concrete example is a bank smartcard with an embedded loyalty applet for a car rental company that awards bonus points for every car rented using the bank card.

In addition to these functionalities there are methods, accessible through the `process` method of the applet, for modifying most of the parameters of the purse applet, including adding knowledge about new loyalty applets that should be notified when card transactions occur.

The loyalty applets of the Development Kit purse application do not attempt to communicate with other applets. We have extended the example loyalty applet with two new functionalities: (i) A loyalty applet can have agreements with other loyalty applets to share bonus points; to achieve this we introduce direct loyalty applet to loyalty applet communication using the interface method `grantLoyaltyPoints`. (ii) loyalty applets can have an agreement with the purse to transfer, according to same fixed rate, part of the bonus points back to the purse. This is achieved through calling the interface method `bonusPointsToPurse` of the purse.

The modified purse and loyalties example is a rewarding example to study using our model checking approach as many key applet correctness properties can be phrased as properties of inter-applet communications.

4.1 Example Properties

Below we list a number of properties of the purse and loyalty applets, formulated using our judgement patterns. We introduce the following abbreviations of the applet class names:

$$\begin{array}{rcl} purse & \triangleq & purse.Purse.Purse \\ loyaltyA & \triangleq & purse.LoyaltyA.LoyaltyA \\ loyaltyB & \triangleq & purse.LoyaltyB.LoyaltyB \end{array}$$

Property 1: there are no calls to both grantPoints and grantLoyaltyPoints for the same applet. For all loyalty applets L it is the case that a call to $L.grantPoints$ never triggers a call to $L.grantLoyaltyPoints$.

$$\begin{array}{rcl} \phi_{1.1} & \triangleq & loyaltyA.grantPoints \text{ never triggers} \\ & & loyaltyA.grantLoyaltyPoints \\ \phi_{1.2} & \triangleq & loyaltyB.grantPoints \text{ never triggers} \\ & & loyaltyB.grantLoyaltyPoints \end{array}$$

Property 2: grantPoints is not transitive. For all loyalty applets L and L' it is the case that a call to $L.grantPoints$ never triggers a call to $L'.grantPoints$. That is, the *grantPoints* method is neither transitive nor recursive.

$$\begin{aligned}
\phi_{2.1} &\triangleq \mathit{loyaltyA.grantPoints}\ \textsf{never triggers}\ \mathit{loyaltyA.grantPoints}\\
\phi_{2.2} &\triangleq \mathit{loyaltyA.grantPoints}\ \textsf{never triggers}\ \mathit{loyaltyB.grantPoints}\\
\phi_{2.3} &\triangleq \mathit{loyaltyB.grantPoints}\ \textsf{never triggers}\ \mathit{loyaltyA.grantPoints}\\
\phi_{2.4} &\triangleq \mathit{loyaltyB.grantPoints}\ \textsf{never triggers}\ \mathit{loyaltyB.grantPoints}
\end{aligned}$$

Property 3: grantLoyaltyPoints is not transitive. The same as Property 2, but for *grantLoyaltyPoints*.

Property 4: grantLoyaltyPoints is called only through grantPoints. That is, within all purse methods m accessible from outside the card, the method *L.grantLoyaltyPoints* of a loyalty applet L is called only through a call to $L'.grantPoints$ of another loyalty applet L' and never directly by the purse applet.

$$\begin{aligned}
\phi_{4.1} \triangleq\ & \textsf{Within}\ m\\
& \quad \mathit{loyaltyA.grantLoyaltyPoints}\ \textsf{through}\ \mathit{loyaltyB.grantPoints}\\
\phi_{4.2} \triangleq\ & \textsf{Within}\ m\\
& \quad \mathit{loyaltyB.grantLoyaltyPoints}\ \textsf{through}\ \mathit{loyaltyA.grantPoints}
\end{aligned}$$

Property 5: Bonus point are awarded at most once within a transaction. Transfer of bonus points from a loyalty to the purse does not cause further bonus points to be awarded.

$$\begin{aligned}
\phi_5 \triangleq\ & \textsf{Within}\ \mathit{purse.bonusPointsToPurse}\\
& \quad \textsf{package}\ \mathit{purse.Purse} \vee \textsf{api}
\end{aligned}$$

That is, calls to the *bonusPointsToPurse* method does not cause a context switch to any other applet package (but possibly to the JavaCard Runtime Environment – JCRE).

The previous correctness properties were specific to certain applications whereas the following express properties that can be beneficial for any JavaCard applet.

Property 6: no constructors called. For all applets A it is the case that no constructor method is called within a call to *A.process*. This can be a crucial property for applets due to the absence of garbage collection in standard JavaCards. Let constructor express the regular expression predicate `.*\..*\.<init>_.*` which tests whether the current location is in a constructor method.

$$\begin{aligned}
\phi_{6.1} &\triangleq \textsf{Within}\ \mathit{purse.process}\ \neg\textsf{constructor}\\
\phi_{6.2} &\triangleq \textsf{Within}\ \mathit{loyaltyA.process}\ \neg\textsf{constructor}\\
\phi_{6.3} &\triangleq \textsf{Within}\ \mathit{loyaltyB.process}\ \neg\textsf{constructor}
\end{aligned}$$

This property holds of the loyalty applets, but not of the purse applet which can create a new object during a call to the process method (without bad consequences, due to conditions involving data).

Property 7: recursion freeness. For all non-API methods m it is the case that a call to m never triggers another call to m.

$$\phi_7 \triangleq m\ \textsf{never triggers}\ m$$

The elapsed time to construct the set of call graphs from the example classes was approximately 16 seconds on a Linux workstation with a Pentium III 1.9 GHz CPU and 256 MB of memory. The resulting call graphs, which includes API program points, consists of 2034 nodes and 3747 edges. The pushdown system generated from these call graphs has approximately 1200 production rules. To check the pushdown system against each of the formulas above, given an initial configuration, took less than one second on the same computer hardware as used for call graph generation.

5 Conclusions and Future Work

The paper proposes a framework for automatic model checking of temporal constraints on inter–applet communications in multi–applet JavaCards.

The framework has been realised by combining a class–based static analysis tool with an automatic model checker for pushdown system and linear temporal logic.

In the future we will refine the static analysis to permit the analysis of communication capabilities of single applets thus connecting to the work on compositional proof systems for JavaCard applets suggested in [1]. This will permit us to analyse whether an applet can operate safely on a smart card even when the knowledge about other applets on the card is imperfect.

Further information regarding the model checking framework and the availability of the tool components and examples can be obtained at the web location `http://www.sics.se/fdt/projects/VeriCode/`.

6 Acknowledgment

We would like to thank Mads Dam for his insightful comments on the verification of JavaCard applets, and Florimond Ployette of INRIA Rennes for his help with modifying the Java based static analysis tool for JavaCard. Thanks are due also to Stefan Schwoon from Technische Universität München for his assistance with the use of the Moped tool, and to the anonymous referees for their valuable remarks on the submitted version of this paper.

References

[1] G. Barthe, D. Gurov, and M. Huisman. Compositional verification of secure applet interactions. In *Proc. FASE'02*, volume 2306 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2002.

[2] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS*, pages 1–16, 2000.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. CONCUR'97*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[4] 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.

[5] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study. In *Proc. FME'02*, Lecture Notes in Computer Science. Springer, 2002. To appear.

[6] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[7] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. CAV'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[8] J. Esparza and S. Schwoon. A BDD-based model checker for recursive programs. In *Proc. CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.

[9] A. Finkel, B. Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. In *Electronic Notes in Theoretical Computer Science*, volume 9, 1997.

[10] K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[11] R. Iosif, C. Demartini, and R. Sisto. Modeling and validation of Java multithreading applications using spin, 1998.

[12] JavaCard 2.1.1 Documentation. Technical report, Sun Microsystems, May 2000. `http://java.sun.com/products/javacard/-specs.html#211`.

[13] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[14] R. Leino, G. Nelson, and J. Saxe. ESC/Java User's Manual. Technical Report 2000–004, Compaq Systems Research Center, October 2002.