Formalisation and Verification of Java Card Security Properties in Dynamic Logic

Wojciech Mostowski

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
e-mail: `woj@cs.chalmers.se`

Abstract. We present how common Java Card security properties can be formalised in Dynamic Logic and verified, mostly automatically, with the KeY system. The properties we consider, are a large subset of properties that are of importance to the smart card industry. We discuss the properties one by one, illustrate them with examples of real-life, industrial size, Java Card applications, and show how the properties are verified with the KeY Prover – an interactive theorem prover for Java Card source code based on a version of Dynamic Logic that models the full Java Card standard. We report on the experience related to formal verification of Java Card programs we gained during the course of this work. Thereafter, we present the current state of the art of formal verification techniques offered by the KeY system and give an assessment of interactive theorem proving as an alternative to static analysis.

1 Introduction

Java Card [8] is a technology designed to incorporate Java in smart card programming. The main ingredient of this technology is the Java Card language specification, which is a stripped down version of Java. In recent years Java Card technology gained interest in the formal verification community. The two main reasons for this are: (1) Java Card applications are safety and security critical, and thus a perfect target for formal verification, (2) due to the relative language simplicity Java Card is also a feasible target for formal verification.

In this paper we show how common Java Card security properties can be formalised in the Dynamic Logic used in the KeY system and proved with the KeY interactive theorem prover (the background of the KeY project is given in Sect. 2). The properties in question are a rather large subset of properties that are of interest to the smart card industry [18]. We demonstrate the formalisation and verification of the properties on two real-life Java Card applets (the case studies are described in Sect. 3). After giving the detailed description of the properties we formalised and proved (Sect. 4), we report on the experience we gained during the course of this work and analyse the main difficulties we encountered. In an earlier paper [12] we reported on the verification of transactions related safety properties based on a somewhat simplified example of a

Java Card purse applet. We proposed the approach of *design for verification*, where we argue that certain precautions have to be taken into account during the design and coding phase to make verification feasible. In this work however, we concentrate on source code verification of already existing Java Card applications without any simplifications whatsoever, and we discuss wider range of security properties than before. In particular, one of the assumptions we made, is that we should be able to specify properties and perform verification without modifying the source code of the verified program. Thus, this work presents the current state of the art of automated formal verification techniques offered by the KeY system for industrial size Java Card applications with respect to meaningful, industry related security properties. This is discussed in Sect. 5. The main conclusion is that full source code verification of Java Card applications is absolutely possible and in most part can indeed be achieved automatically, however, such verification requires deep understanding of the specification issues, including full understanding of the application being verified and the specificities of the Java Card environment. Therefore, we consider the KeY system, assuming the approach we present in this work, mostly suitable for experienced users. The properties that we consider here, originate from the area of static analysis [18], however, to the best of our knowledge, no static analysis technique for thorough treatment of those properties has been developed. We managed to formalise and verify almost all of the properties using the KeY interactive theorem prover. For the remaining properties we give concrete suggestions on how to treat them with the KeY system. We give arguments why we think that interactive theorem proving is a reasonable, and in fact in some ways better, alternative to static analysis. This discussion is included in the second part of Sect. 5.

2 Background

The KeY Project. The work presented here is part of the KeY project[1] [1]. The main goals of KeY are to (1) provide deductive verification for a real world programming language and to (2) integrate formal methods into industrial software development processes.

For the first goal a deductive verification tool for Java source programs, the KeY Prover, has been developed. The main target of the KeY system is the Java Card language – a stripped down version of Java used to program smart cards (e.g., Java Card does not support concurrency or large primitive data types, and has a very small API). The verification is based on a specifically tailored version of Dynamic Logic – Java Card Dynamic Logic (Java Card DL), which supports most of sequential Java, in particular the full Java Card language specification including the Java Card transaction mechanism. Java Card DL and the KeY Prover are designed to make the verification process as automated as possible.

For the second goal, the KeY Prover was integrated into a commercial CASE tool, which uses UML (Unified Modelling Language) as the design language

[1] `http://www.key-project.org`

and OCL (Object Constraint Language) as the specification language. For the present work however, due to specificities of the security properties in question, and the necessity to operate on relatively low level of the specification, we took the approach of using JAVA CARD DL directly as a specification language.

JAVA CARD Dynamic Logic with Strong Invariants. We give a very brief introduction to JAVA CARD DL. We are not going to present or explain any of its sequent calculus rules. Dynamic Logic [13] can be seen as an extension of Hoare logic. It is a first-order modal logic with parametric modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of legal JAVA CARD statements). In the Kripke semantics of Dynamic Logic the worlds are identified with execution states of programs. A state s' is *accessible* from state s *via* p, if p terminates with final state s' when started in state s.

The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p, and $\langle p \rangle \phi$ expresses that ϕ holds in *some* final state of p. Since JAVA CARD programs are deterministic, there is exactly one final state (p terminates) or no final state (p does not terminate). In JAVA CARD DL termination forbids exceptions to be thrown, i.e., a program that throws an uncaught exception is considered to be non terminating (or, terminating abruptly) [5]. The formula $\phi \rightarrow \langle p \rangle \psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds. The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, i.e., ψ needs only to hold *if* p terminates. On top of that, a "throughout" modality ($[\![\cdot]\!]$) has been introduced to JAVA CARD DL. As opposed to the box and diamond modalities, the throughout modality requires that a certain property is maintained at *all intermediate* program states, so for the throughout modality the semantics of a program is a sequence of all states the execution passes through when started in the current state (its *trace*). This allows us to ensure that a certain property should hold even in case of an unexpected/abrupt termination (e.g., when the smart card is ripped out from the terminal). We call such properties *strong invariants*. Strong invariants are the central part of one of the discussed security properties.

JAVA CARD DL is axiomatised in a sequent calculus to be used in deductive verification of JAVA CARD programs. The detailed description of the calculus can be found in [2]. The calculus covers all features of JAVA CARD, such as exceptions, complex method calls, or JAVA arithmetic. The full JAVA CARD DL sequent calculus is implemented in the KeY Prover. The prover itself is implemented in JAVA. The calculus is implemented by means of so-called taclets [3], that avoid rules being hard coded into the prover. Instead, rules can be dynamically added to the prover. As a consequence, one can, for example, use different versions of arithmetic during a proof: idealised arithmetic, where all integer types are infinite and do not overflow, or JAVA arithmetic, where integer types are bounded and exhibit overflow behaviour [6]. Full treatment of strong invariants also required formalisation of JAVA CARD transactions in the logic. The transaction mechanism [8] ensures that a piece of JAVA CARD program is executed to completion or not at all. The theoretical aspects of integration of the throughout modality and transactions into JAVA CARD DL are discussed in [4].

Related Work. For us, the most interesting formal approaches to Java Card application development are those considered with source code level verification, based on static checking and various program calculi. The work of Jacobs et al. [14] is most closely related to our work and can partly serve as a brief overview of verification techniques targeted at source code. It reports on successful verification attempts of a commercial Java Card applet with different verification tools: ESC/Java2 [10], Krakatoa [16], Jive [19], and LOOP [15]. The security property under consideration, one of the properties we discuss in this paper, is that only `ISOExceptions` are thrown at the top level of the applet. The analysed applet is a commercial one, sold to customers. There are no technical details revealed about the applet, so it is difficult to compare its complexity to our case studies. Jacobs et al. detected subtle bugs in the applet with respect to a possible uncaught `ArrayIndexOutOfBoundsException` (with LOOP and Jive tools), as well as full verification (no exceptions other than `ISOException`, satisfied postcondition, and preserved class invariant) of single methods with the Krakatoa tool. The paper admits that expertise and considerable user interaction with the back-end theorem provers (PVS and Coq) were required. It is also noted that the provers are the performance and scalability bottlenecks in the verification process. We will relate to those issues while we present our results.

3 Case Studies

In the remainder of this paper we will use two Java Card case studies. The first one is a Java Card electronic purse application *Demoney*[2] [17]. While *Demoney* does not have all of the features of a purse application actually used in production, it is provided by *Trusted Logic S.A.* as a realistic demonstration application that includes all major complexities of a commercial program, in particular it is optimised for memory consumption, which, as noted in [12], is one of the major obstacles in verification. The *Demoney* source code is at present not publicly available, so we are not able to disclose some of the technical details necessary to fully discuss the verification problems associated with *Demoney*, but we hope that what we present is convincing enough.

The second case study is an RSA based authentication applet for logging into a Linux system (`SafeApplet`). It was initially developed by Dierk Bolten for Java Powered iButtons[3] and was one of the motivating case studies to introduce strong invariants into Java Card DL. Here, we use a fully refactored version of `SafeApplet`, which is described in [20].

4 Security Properties

The security properties that we discuss here are directly based on the ones described in [18], which we will refer to as the *SecSafe* document in the rest

[2] We thank Renaud Marlet of Trusted Logic S.A. for providing the *Demoney* code.

[3] `http://www.ibutton.com`

of the paper. We considered all of the properties listed there, but few of them we did not yet analyse in full detail. However, we still discuss those remaining properties and the possibilities of handling them in the KeY system at the end of this section. Let us start with a brief overview of the five properties that we do discuss in detail.

Only `ISOExceptions` at Top Level (Sect. 3.4 of the SecSafe document). The exceptions of type `ISOException` are used in JAVA CARD to signal error conditions to the outside environment (the smart card terminal). Such an exception results with a specific APDU (Application Protocol Data Unit) carrying an error code being sent back to the card terminal. To avoid leaking out the information about error conditions inside the applet, a well written JAVA CARD applet should only throw exceptions of type `ISOException` at top level.

No X Exceptions at Top Level. Due to its complexity, the first property is proposed to be decomposed into simpler subproperties. Such properties say that certain exceptions are not thrown, including most common ones (e.g., `NullPointerException`). A special case of this property is the next one.

Well Formed Transactions. This property consists of three parts, which say, respectively: do not start a transaction before committing or aborting the previous one, do not commit or abort a transaction without having started any, and do not let the JAVA CARD Runtime Environment close an open transaction. The JAVA CARD specification allows only one level of transactions, i.e., there is no nesting of transactions in JAVA CARD. As we show later, this property can be expressed in terms of disallowing JAVA CARD's `TransactionException`.

Atomic Updates (Sect. 3.5 of the SecSafe document). In general, this property requires related persistent data in the applet to be updated atomically. In the context of our work this property is directly connected to the "rip-out" properties and strong invariants, which we will use to deal with this property.

No Unwanted Overflow (Sect. 3.6 of the SecSafe document). This property simply says that common integer operations should not overflow.

In the following we will go through these security properties one by one. For each of the properties we will give a general guideline on how to formalise it in JAVA CARD DL, give an example based on one or both of the case studies, give comments about the verification of a given property and possibly discuss some more issues related to the property. Due to space restrictions and the lengthy code snippets in our examples, we are going to show abbreviated versions of the examples. A technical report discussing all the examples in full detail is available [21].

4.1 Only ISOExceptions at Top Level

The KeY system provides a uniform framework for allowing and disallowing exceptions of any kind in JAVA CARD programs. We explain this with a general example. Given some applet `MyApplet` one can forbid `aMethod` to throw any exception other than `ISOException` in the following way (this is the actual syntax used by the KeY Prover, we will explain it shortly):

```
java {"source/"}

program variables { MyApplet self; }

problem {
 preconditions ->
  <{ method-frame(MyApplet()): {
      try { self.aMethod(); } catch(ISOException ie) {}
     } }> true }
```

This is a proof obligation that is an input to the KeY Prover. The first section in the file tagged with `java` tells the prover where the source code of the program to be verified is. The `program variables` section defines all the program/Java variables that are going to be used in the proof obligation. The `problem` section defines the actual proof obligation. The string `preconditions` is a place holder for the preconditions necessary to establish the correct execution of `aMethod`. One of the obvious conditions to put there, is that the `self` reference is not `null`: `!self = null`. With this proof obligation we want to prove that a call to `aMethod` either terminates normally or with an exception of type `ISOException`. The actual call to the method, `self.aMethod()`, appears inside the diamond modality (`<{}>`) and is wrapped with some additional statements. The diamond requires the program to terminate normally – the trivial postcondition `true` is only satisfied if no exceptions are thrown. So, to specify that a program throws a certain kind of exception only, one wraps the actual program with a `try-catch` statement catching the particular kind of exception. This way, if our method terminates normally or throws an `ISOException` (only), the program inside the diamond still terminates normally, making the proof obligation valid. In case any other kind of exception is thrown the proof obligation becomes invalid. The `method-frame` statement tells the prover that our program is executed in the context of the `MyApplet` class (e.g., such information is necessary if `aMethod` is private). The `method-frame` statements is one of the extensions to Java syntax used in Java Card DL to deal with scopes of methods, method return values, etc. We want to stress here, that this extension is a *superset* of Java, not a subset – any valid Java/Java Card program can be used inside the modality.

Let us now demonstrate this property with real examples. First we give a specification of *Demoney*'s method `verifyPIN`. This method is common to almost every Java Card applet, it is responsible for verifying the correctness of the PIN passed in the APDU. If the PIN is correct the method sets a global flag indicating successful PIN verification and returns, otherwise an `ISOException` with a proper status code (including the number of tries left to enter the correct PIN) is thrown. The proof obligation below specifies that `verifyPIN` is only allowed to throw `ISOException`. The example is abbreviated; however, no important issues are omitted:

```
program variables {
  fr.trustedlogic.demo.demoney.Demoney self;
  javacard.framework.APDU apdu; ... }

problem {
  // General preconditions for verifyPIN, e.g., !self = null & ...
  // PIN well formed preconditions: !self.pin = null & ...
  // ISOException well formed preconditions: ...
-> <{ method-frame(fr.trustedlogic.demo.demoney.Demoney()): {
       try{ self.verifyPIN(apdu,offset,length); }catch(ISOException ie){}
     } }> true }
```

There are numerous preconditions to guard the execution of `verifyPIN`. It took some trial and error steps to get all the preconditions right (we discuss this issue in detail in Sect. 5). Missing even the smallest one renders the program not terminating normally. This proof obligation is proven automatically by the KeY Prover in slightly more than 3 minutes[4] with less that 10 000 proof steps.

The *SecSafe* document requires that exceptions other than `ISOException` are not thrown as a result of invoking the entry point of the applet. For us, it means that we would have to prove our property for the applet entry method `process`. At the current stage of our experiments we found it technically difficult to perform a proof of this kind for the applet of the size of *Demoney*. We know however, that such a proof can be modularised (see next example).

The second example is based on `SafeApplet`. Among other things, `SafeApplet` keeps a table of registered users that can be authenticated with the applet. For each user a unique user ID and a set of RSA encryption keys are stored. Method `dispatchDeleteKeyPair` is responsible for unregistering a given user ID, it takes an APDU, which stores the user ID to be unregistered. In case no user with such an ID is registered an `ISOException` with a proper code (`SW_USER_UNREGISTERED`) is thrown, otherwise the proper entry in the user table is removed:

```
// APDUException, ISOException well formed, ...
& !self = null & !self.temp = null & ...
-> <{ method-frame(SafeApplet()):{
       finishedWithISOEx = false; finishedOK=false;
       try { self.dispatchDeleteKeyPair(apdu); finishedOK = true;
       }catch(ISOException e1){ finishedWithISOEx = true; }
     } }> (finishedOK = TRUE | (finishedWithISOEx = TRUE &
       ISOException.instance.theSw[0] = SafeApplet.SW_USER_UNREGISTERED))
```

Among other things, the precondition says that the entries in the user table are not `null`. In the postcondition we also want to specify that the `ISOException` that might be thrown contains the right status code. Because of this, we need to distinguish between two cases in the postcondition: *either* the method terminates normally *or* an `ISOException` is thrown with a proper status code – two boolean variables (`finishedOK` and `finishedWithISOEx`) keep track of this. The way the program in the modality is constructed ensures that those two variables cannot be `true` at the same time (this can also be verified).

[4] All the benchmarks presented here were run on a Pentium IV 2.6 GHz Linux system with 1.5 GB of memory. The version of the KeY system used is available on request.

Proof Modularisation. This proof obligation is proved automatically with the KeY Prover in about 15 minutes with less than 40 000 proof steps. This may seem to be a lot. The reason for such performance is threefold. First of all, there is a loop involved, which goes through the table of users. This loop is symbolically unwound step by step and the proof size depends on the actual constant value of `MAX_USERS` (equal to 5). Secondly, the method performs a lot of preliminary work before the users table is modified. Finally, for this particular benchmark result, there was no proof modularisation used whatsoever – when a method call is made in a program the prover replaces the call with the actual method body and executes it symbolically. Instead, one can use the specification of the called method – it is enough to establish that the precondition of the called method is satisfied, and then the call can be replaced with the postcondition of the called method. Obviously, one also has to prove that the called method satisfies its specification. One limitation of this technique is that the method specification have to include so called modification conditions [22, 7] – a complete set of attributes that the method possibly modifies. Factoring out method calls this way shortens the total proof effort even in the simplest cases – although a method might be called only once in a program, due to proof branching, it may need to be analysed in the proof multiple times. For comparison, we applied such modularisation to our last example – we used specification for just one method that contains a loop. The resulting proof took less that one minute (5 000 proof steps), the side proof establishing that the factored out method satisfies its specification took less than 2 minutes (12 000 proof steps) – the time performance increased 5 times.

4.2 No X Exceptions at Top Level

As already mentioned, the KeY system provides a uniform framework for dealing with exceptions. The Java Card DL calculus rules and the semantics of the diamond modality require that no exceptions are thrown whatsoever. In particular, the calculus is carefully designed to establish that each object that is dereferenced is not `null`, that the indices used to access array elements are within array bounds, etc. So, as long as the total correctness semantics is used, the KeY Prover establishes absence of all possible exceptions. Still, for the sake of consistency, one can disallow only one kind of exception this way:

```
preconditions & unwantedException = FALSE ->
  <{ method-frame(MyApplet()): {
      try { self.aMethod(); } catch(Exception e) {
        unwantedException = (e instanceof UnwantedException); } }
  }> (unwantedException = FALSE)
```

Here, the boolean variable `unwantedException` will become true only when the undesired exception is thrown in `aMethod`, thus the above proof obligation states that no `UnwantedException` is thrown by `aMethod`.

4.3 Well Formed Transactions

The first two parts of this property say that a transaction should not be started before committing or aborting the previous one, and that no transaction should be committed or aborted if none was started. This boils down to saying that no `TransactionException` related to well-formedness is thrown in the program. Since in our model of Java Card environment we do not consider transaction capacity, we can simplify this part of the property to “No `TransactionException` is thrown in the program.” – a special case of the previous property.

The last part of the property says that no transactions should be left open to be closed by JCRE. The information about open transactions is kept track of by JCRE and can be accessed through the Java Card API (static attribute `JCSystem.transactionDepth`). It is quite straightforward to specify that a given method does not leave an open transaction:

```
preconditions & JCSystem.transactionDepth = 0
-> <{ method-frame(MyApplet()): { self.aMethod(); }
    }> (JCSystem.transactionDepth = 0) }
```

The precondition states that there is no open transaction before `aMethod` is called. This is necessary in case `aMethod` is top-level and does not check for an open transaction before it starts its own. After `aMethod` is finished we require the `transactionDepth` to be equal to 0 again, this ensures that there is no open transaction. Also, what is implicit, is that no `TransactionException` is thrown. We will briefly illustrate this property with a real example in the next section.

4.4 Atomic Updates

This property requires *related* persistent data in the applet to be updated atomically. Strong invariants seem to be the right technique to deal with this property – as we stated already, they are used to specify consistency of data at all times, so that in case an abrupt termination occurs, the data (in particular, related data) stay consistent. We will illustrate this property briefly with the same example that is discussed in full in [12], for this work however we were able to use the real *Demoney* applet instead of the simplified one used in [12]. One of the routines of the electronic purse is responsible for recording information about the purchase in the log file. Among other things, the current balance after the purchase is recorded in a new log entry. As the *SecSafe* document points out accurately, when atomic consistency properties are considered, one has to be able to say what it means for the data to be related. In our example we want to state that the current balance of the purse is always the same as the one recorded in the most recent log entry. By using Java Card transaction mechanism, method `performTransaction` is responsible for debiting the purse balance and updating the log file in one atomic step. In Java Card DL, to express that a strong invariant is preserved, the throughout modality is used:

```
  JCSystem.transactionDepth = 0 & !self = null & ...
  // Strong Invariant: The current balance of the purse is equal to the
  // balance recorded in the most recent log entry: self.balance = ...
-> [[{method-frame(fr.trustedlogic.demo.demoney.Demoney()): {
       self.performTransaction(amount, apduBuffer, offsetTransCtx); }
   }]] // Strong Invariant: same as above
```

An important part of the precondition is the one saying that the strong invariant holds before the method is executed. This proof obligation is proved automatically in 12 minutes with less than 12 000 proof steps. This particular method uses two loops to copy array data, which were not factored out with modularisation, so we consider this a relatively good result. Modularisation has been used for some other, simple methods, however, we have to point out here, that in case of proof obligations involving the throughout modality and transactions using method specifications is not possible in general, and in cases where it is possible it has to be used with caution.

This proves that the related data stays consistent throughout the execution of `performTransaction`. Since a Java Card transaction is involved in this method it is also desirable to prove well-formedness of transactions for `performTransaction`, as stipulated in the previous section:

```
// Mostly the same preconditions as for the previous proof obligation
-> <{method-frame(fr.trustedlogic.demo.demoney.Demoney()): {
      self.performTransaction(amount, apduBuffer, offsetTransCtx); }
   }> (JCSystem.transactionDepth = 0)
```

This is proved automatically in 11 minutes with less than 12 000 proof steps.

We proved a similar property for `SafeApplet`, saying that all the registered users have a properly defined set of private and public encryption keys at all times. Here we only make two comments about the proof. First, there are no transactions used in `SafeApplet` to ensure data consistency, instead additional fields in objects associated with consistency property are used and accessing of those objects is carefully coded. This results in a more complex proof. Second, during the proof, some small amount of manual interaction with the prover was necessary, namely 8 manual quantifier instantiations were required, otherwise the proof proceeded automatically and took 3 minutes to finish.

4.5 No Unwanted Overflow

Finally, we deal with a property related to integer arithmetic: additions, subtractions, multiplications and negations must not overflow. To deal with all possible issues related to integer arithmetic, in particular overflow, the KeY Prover uses three different semantics of arithmetic operations [6]. The first semantics treats the integer numbers in the idealised way, i.e., the integer types are assumed to be infinite and, thus, not overflowing. The second semantics bounds all the integer types and prohibits any kind of overflow. The third semantics is that of Java, i.e., all the arithmetic operations are performed as in the JVM, in particular they are allowed to overflow and the effects of overflow are accurately modelled. Thus, to deal with overflow properties, it is enough for the user to choose appropriate integer semantics in the KeY Prover. Based on the *SecSafe* document, below is an example of a badly formed program with respect to overflow:

```
inShort(balance) & inShort(maxBalance) & inShort(credit) &
balance > 0 & maxBalance > 0 & credit > 0 ->
  <{ try { if (balance + credit > maxBalance) throw ie;
```

```
        else balance += credit;
  }catch(ISOException e){} }> balance > 0
```

The problem is that the `balance + credit` operation can overflow making the condition inside the `if` statement false resulting in a `balance` being less than 0 after this program is executed. When processed by the KeY Prover with the idealised integer semantics switched on, this proof obligation gets proved quickly. When the arithmetic semantics with overflow control is used, this proof obligation is not provable. The fix to the program to avoid overflow is to change the `if` condition to `balance > maxBalance - credit`. The modified proof obligation is provable with both kinds of integer semantics.

4.6 Other Properties

We have just shown how to formalise and prove five kinds of security properties from the *SecSafe* document. Here we briefly discuss the remaining ones.

Memory Allocation. Due to restricted resources of a smart card, one of the requirements on a properly designed Java Card applet is the constrained memory usage: bounded dynamic memory allocation and no memory allocation in certain life stages of the applet. This seems like a problem suitable for static analysis – in general there is no need for precise analysis of the control flow, although, for example, if memory allocation is performed inside a loop, a precise analysis is required to find out the loop bounds. Either way, we believe that this property in general can be formalised and proved with the KeY system as well. The main idea is the following. The KeY Prover maintains a set of implicit attributes for every object to model certain aspects of the Java virtual machine, in particular object creation. There is no obstacle to introduce a new static implicit attribute to our Java model that would keep track of the amount of allocated memory or the possibility to allocate memory. However, due to optimisation of inheritance and interface representation in JVM, the actual memory consumption may differ for each JVM implementation. Thus, keeping precise record of the allocated memory is not so simple and thorough treatment of this problem requires further research. At the moment, we could only give approximate figures for memory usage.

Conditional Execution Points. This property says that certain program points must only be executed if a given condition holds. Again, this is a subject to static analysis (e.g., ESC/Java2 provides means to annotate and check conditions at any program point), but it can also be done with theorem proving by introducing a generalised version of the throughout modality. The throughout modality requires that a property holds after every program statement. For the generalised case, such a property would have to hold only in certain parts of the program. So there are no theoretical obstacles here, but due to less priority this has not yet been implemented in KeY.

Information Privacy and Manipulation of Plain Text Secret. Those two properties fall into the category of data security properties. As it has been shown in [9], formalising and proving data security properties can in general be integrated into interactive theorem proving, however no experiments on real Java Card examples were performed so far.

5 Discussion

Lessons Learned. Here we sum up the practical experience we gained during the course of this work. The main lesson is that the current state of software verification technology that at least the KeY system offers makes the verification tasks feasible. Schematic formalisation of the security properties from the *SecSafe* document was easy, however, applying it to concrete examples was much more tricky. We found getting right all the preconditions to guard the execution of a given method very difficult. This particularly holds when normal termination is required. Constructing the preconditions requires deep understanding of the program in question and the workings of the JCRE. However, calculation of the preconditions can be tool supported as well:

In [14] ESC/Java2 is used to construct preconditions. In short, the tool is run interactively on an unspecified applet, which results in warnings about possible exceptions. Such warnings are removed step by step by adding appropriate expressions to the precondition. Alternatively, as [14] suggests, the weakest precondition calculus of the Jive system could be used by running the proof "backwards", i.e., by starting with a postcondition and calculating the necessary preconditions. This however, has not been presented in the paper and to our understanding the approach has certain limitations.

The KeY system itself provides a functionality to compute specifications for methods to ensure normal termination [23]. The basic idea behind computing the specification is to try to prove a total correctness proof obligation. In case it fails, all the open proof goals are collected and the necessary preconditions that would be needed to close those goals are calculated. There are two disadvantages to this technique: (1) for the proof to terminate the preconditions that guard the loop bounds cannot be omitted, so there is no way to calculate preconditions for loops, they have to be given beforehand, (2) proofs have to be performed the same way for computing the specification as it is done when one simply tries to prove the obligation, so computing the specification is in fact a front-end for analysing failed proof attempts in an organised fashion. Moreover, the specifications produced can be equally hard to read as is analysing the failed proof attempt manually. Despite all this, we still find the specification computation facility of the KeY system quite helpful for proof obligations that produce small failed proof attempts or at least ones containing only few open proof goals.

Proving partial correctness also requires caution. A wrong or unintended precondition can render the program to be always terminating abruptly. This makes any partial correctness proof obligation trivially true. Thus, in cases where a partial correctness proof is necessary (e.g., strong invariants), one should accompany such a proof with an additional termination property, as we did in Sect. 4.4.

To enable automation, the KeY Prover and the Java Card DL are designed in a way not to bother the user with the workings of the calculus and the proof system. However, we have realised that proper formulation of the DL expressions can further support automation. We have also introduced a small number of additional simplification rules for arithmetic expressions. Such rules considerably simplify the proof, but introducing them, although being relatively easy, requires a little bit more than the basic understanding of JAVA CARD DL. Moreover, each introduced rule has to be proven sound. The rules are very simple and we have means to do it automatically with the KeY system [3], but due to constantly changing set of those rules, we decided to leave the correctness proofs out for the time being.

Our experimental results show that proof modularisation greatly reduces the verification effort. The problem of modularising proofs using method specifications has been well researched [22, 7], but has been implemented in the KeY system only recently, thus, we gained relatively little experience here. So far we have learnt that using method specification in the context of the throughout modality is not always possible and has to be done with care.

Finally, one of the goals of formal verification is to find and eliminate bugs. So far, we have not found any in our case studies. We believe the reason for this is twofold. First, the properties we considered so far were relatively simple and the methods were expected not to contain bugs related to those properties. Second, neither of the applications we analysed as a whole, only parts of them. In particular, the bugs often occur at the points where the methods are invoked, due to an unsatisfied method precondition.

Static Analysis vs. Interactive Theorem Proving. The results of this paper show that we are able to formalise and prove all of the security properties defined in the *SecSafe* document. Many of the properties would require quite advanced static analysis and, as far as we know, no such static analysis technique has been developed so far. Moreover, we believe that some properties go beyond static analysis, e.g., certain aspects of memory allocation (Sect. 4.6) require accurate analysis of the control flow. Furthermore, each single property would probably require a different approach in static analysis, while the KeY Prover provides a uniform framework. For example, all properties related to exceptions are formalised in the same, general way, and in fact can be treated as one property. Also, dealing with integer overflow is done within the uniform framework of different integer semantics, that cover all possible overflow scenarios.

Therefore, we consider interactive theorem proving as a feasible alternative to static analysis. More generally, deep integration of static analysis with our prover is a subject of an ongoing research [11]. One argument that speaks for static analysis is full automation. However, our experiments show that the KeY system requires almost no manual interaction to prove the properties we discussed. Also, the time performance of the KeY prover seems to be reasonable, although the work on improving it continues. On the other hand, as we noticed earlier, constructing proof obligations require some user expertise. In our opinion however, this is something that is difficult to factor out when serious formal verification attempts are considered, no matter if theorem proving or static analysis is used as the basis.

6 Summary and Future Work

We have shown how most of the security properties of the industrial origin for Java Card applications can be formalised in Java Card DL and proved, for the most part automatically, with the KeY Prover. Most of the properties were illustrated by real-life Java Card applets. Considerable experience related to formal verification has been gained during the course of this work. This experience indicates that Java Card source code verification, at least using the KeY system, has recently become a manageable and relatively easy task, however, for scenarios like the one presented in this work, user expertise is required. Two main areas for improvement are clearly the modularisation of the proofs and tool support for calculating specifications (more precisely, preconditions). Our future work will concentrate on those two aspects, to reach full, truly meaningful verification of Java Card applications with as much automation as possible. We feel that the performance results should already be acceptable by software engineers, however, the work on improving the speed of the prover will continue. Finally, our experience clearly shows that interactive theorem proving is a reasonable alternative to static analysis – we plan to further explore this area by concentrating on the few properties we only discussed briefly here.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, April 2004. Online First issue, to appear in print.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.
3. B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
4. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2621 of *LNCS*, pages 246–260, Warsaw, Poland, April 2003. Springer.
5. B. Beckert and B. Sasse. Handling Java's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.
6. B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, April 2004.

7. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
8. Z. Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, June 2000.
9. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. Technical Report 2004–01, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004.
10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for JAVA. In *Proceedings, ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
11. T. Gedell. Integrating static analysis into theorem proving. Available from `http://www.cs.chalmers.se/~gedell/publications/satp.ps`.
12. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In G. Barthe and M. Huisman, editors, *CASSIS'04 Post Workshop Proceedings*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
13. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
14. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Proceedings, Algebraic Methodology And Software Technology, Stirling, UK*, volume 3116 of *LNCS*, pages 241–256. Springer, July 2004.
15. B. Jacobs and E. Poll. JAVA program verification at Nijmegen: Developments and perspective. In *Software Security – Theories and Systems: Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4–6, 2003. Revised Papers*, volume 3233 of *LNCS*, pages 134–153. Springer, 2003.
16. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVA CARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.
17. R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.
18. R. Marlet and D. L. Métayer. Security properties and JAVA CARD specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
19. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system – Implementation description. Available from `http://softech.informatik.uni-kl.de/old/en/publications/jive.html`, 2000.
20. W. Mostowski. Rigorous development of JAVA CARD applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proceedings, Fourth Workshop on Rigorous Object-Oriented Methods, London, U.K.*, March 2002. Available from `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.
21. W. Mostowski. Formalisation and verification of JAVA CARD security properties in Dynamic Logic. Technical Report 2004–08, Department of Computing Science, Chalmers University of Technology, Göteborg, Sweden, October 2004. Available from `http://www.cs.chalmers.se/~woj/papers/secprop.pdf`.
22. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.
23. A. Platzer. Using a program verification calculus for constructing specifications from implementations. Minor thesis, Karlsruhe University, Computer Science Department, February 2004.