Technical Report no. 2004–08

Formalisation and Verification of JAVA CARD Security Properties in Dynamic Logic

Wojciech Mostowski

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2004

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2004–08
ISSN: 1650–3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE 412–96 Göteborg, Sweden

Göteborg, Sweden, 2004

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 [9] is a technology designed to enable and 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. There are two main reasons for this: (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 properties in question are a rather large subset of properties that are of interest to the smart card industry [20]. We demonstrate the formalisation and verification of the properties on two real-life Java Card applets. After giving the detailed description of the properties we formalised and proved, we report on the experience we gained during the course of this work and analyse the main difficulties we encountered. In an earlier paper [14] 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. 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 [20], 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.

In the following Section we give the background information about the KeY project, its objectives, the Dynamic Logic used in the KeY interactive prover, and a brief overview of related work. Section 3 describes shortly the Java Card applications (case studies) used to demonstrate our results. In Section 4 we present the formalisation of security properties one by one illustrated with numerous examples and also discuss briefly properties not covered in this paper. In Section 5 we discuss the difficulties we encountered during the course of this work, the experience we gained, and we asses interactive theorem proving as an alternative to static analysis. Finally, Section 6 concludes the paper.

2 Background

2.1 The KeY Project

The work presented in this paper 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. 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 in a way to make the verification process as automated as possible.

For the second goal we enhance a commercial CASE tool with functionality for formal specification and deductive verification. The design and specification languages of our choice are respectively UML (Unified Modelling Language) and OCL (Object Constraint Language), which is part of the UML standard [24]. The KeY system translates OCL specifications into Java Card DL formulae, whose validity can then be proved with the KeY Prover. All this is tightly integrated into a CASE tool, which makes formal verification as transparent as possible to the untrained user.

Of course, the use of OCL is not mandatory: logically savvy users of the KeY system can write their proof obligations directly in Java Card DL and use its full expressive power. Due to specificities of the security properties in question and the necessity to operate on relatively low level of the specification this is actually the approach we have taken in the present work.

2.2 Java Card

Java Card technology [9] provides means of programming smart cards with (a subset of) the Java programming language. Smart cards are nothing more (and nothing less) than small computers, providing limited power CPU and three types of memory: ROM (read only), EEPROM (writable, persistent), and RAM memory (writable, non-persistent). The card's ROM contains a Java Card Virtual Machine and the implementation of the Java Card API, together they allow running Java Card applets on the card. The EEPROM memory is used to store applet's persistent data that is kept from session to session, while RAM is used for local run-time computations. Smart cards communicate with the rest of the world through application

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

protocol data units (APDUs, ISO 7816–4 standard). The communication is done in master-slave mode—it's always the master/terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU. Certain Java language features are not supported by the Java Card language: large primitive data types (`int`, `long`, `double`, `float`), characters and strings, multidimensional arrays, dynamic class loading, threads and garbage collection. Most of the remaining Java features, in particular object oriented ones like interfaces, inheritance, virtual methods, overloading, dynamic object creation, are supported by the Java Card language. Also, the Java Card API is a very small subset of the Java API designed to handle smart card specific routines and resources: Application IDentifiers (AIDs), APDUs, and Java Card applets among others. Schematically, Java Card applet implements the `install` method responsible for the initialisation of the applet and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host.

2.3 Java Card Dynamic Logic

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 [26, 15] 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. In versions of DL with a non-deterministic programming language there can be several final states, but Java Card programs are deterministic, so there is exactly one final state (when p terminates) or no final state (when 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, that is ψ needs only to hold *if* p terminates.

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, atomic transactions (see below), 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].

Strong Invariants. The most common semantics of an invariant is based on the initial and final states of a program, i.e., if an invariant holds before the program is executed then it should hold after the execution has completed. This however is not enough to treat certain atomicity properties, for example to specify that a certain property should hold in case of an unexpected/abrupt termination (e.g., when the smart card is ripped out from the terminal). Thus, we introduced the notion of a strong invariant to Java Card DL. Such an invariant on the objects' data is maintained at any time during applet execution and, in particular, in case of abrupt termination. This resulted in extending the Java Card DL with a new modal operator $[\![\cdot]\!]$ ("throughout"), which closely corresponds to Temporal Logic's $\Box$ operator. In the extended logic, the semantics of a program is a sequence of all states the execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of intermediate states in traces of terminating and non-terminating programs. To fully treat strong invariant related properties one also needs formalisation of Java Card transactions in the logic. The transaction mechanism [9] 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] and the practical experiences in [14]. We refer the reader to those two papers for more in-depth discussion about transaction related issues, here we should only say that transactions (specifically, the possibility of an programmatic transaction abort) make the technical details of Java Card DL quite involved. Strong invariants and transactions are central part of one of the discussed security properties.

2.4 Related Work

Formal approaches to Java Card application development cover a wide spectrum of techniques and we discuss only some of them here. One of the most common low-level ones are byte code level verification [8] and model checking [10]. For us, the most interesting approaches are those considered with source code level verification, based on static checking and various program calculi. The work of Jacobs et al. [16] is most closely related to our work and can partly serve as an 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 [12], the Krakatoa tool [18], the Jive system [21], and the LOOP tool [17]. 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] [19]. 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 [14], is one of the major obstacles in verification. The *Demoney* source code is at present not publicly available, so there are certain limits to the level of the technical detail in the presented examples.

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 [22].

4 Security Properties

The security properties that we discuss here are directly based on the ones described in [20], which we will refer to as the *SecSafe* document in the rest 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 (Section 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 `NullPointerException`, `ArrayIndexOutOfBoundsException`, or `NegativeArraySizeException`. 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

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

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

transactions in JAVA CARD. As we show later, this property can be expressed in terms of disallowing JAVA CARD's `TransactionException`.

Atomic Updates (Section 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 (Section 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.

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(javacard.framework.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, without any exceptions—any program p throwing an uncaught exception does not satisfy the formula $\langle p \rangle \mathit{true}$. 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. Such information is necessary, for example, when the method in question 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. What follows, and what cannot be seen in this schematic example, is that method calls can have arguments and return values of arbitrary Java type.

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. When the PIN is correct the method sets a global flag indicating successful PIN verification and returns. If the PIN is not correct or the maximum number of PIN entry trials has been reached an `ISOException` with a proper status code (including the number of tries left to enter the correct PIN) is thrown. The following is the proof obligation for the KeY Prover specifying that the `verifyPIN` method is only allowed to throw `ISOException`. For the simplicity of reading we diverge slightly from the actual KeY Prover syntax, however no important issues are omitted[4]:

```
java {"demoney/"}

program variables {
  fr.trustedlogic.demo.demoney.Demoney self;
  javacard.framework.APDU apdu;
  byte length;
  short offset;
}

problem {
// General preconditions for verifyPIN
     !self = null & !apdu = null
   & length = Demoney.VERIFY_PIN_LC & offset = ISO7816.OFFSET_CDATA
   & !apdu.buffer = null
   & apdu.buffer.length = length + ISO7816.OFFSET_CDATA
   & apdu.buffer[ISO7816.OFFSET_LC] = length
// PIN well-formed
```

[4]E.g., accessing private object attributes requires extra syntax, integer operations are not expressed with infix operators, etc.

```
   & !self.pin = null
   & !self.pin.isValidated = null & self.pin.isValidated.length = 1
   ...
// ISOException well-formed
   & !ISOException.systemInstance = null
   & !ISOException.systemInstance.theSw = null
   & ISOException.systemInstance.theSw.length = 1
-> <{ method-frame(fr.trustedlogic.demo.demoney.Demoney()): {
       try {
         self.verifyPIN (apdu, offset, length);
       }catch(javacard.framework.ISOException ie) {}
     } }> true
}
```

There are numerous preconditions to guard the execution of `verifyPIN`. The first set of preconditions defines, among other things, the proper values for the arguments passed to `verifyPIN`. The second set specifies that the `pin` attribute of the applet is a properly allocated `OwnerPIN` object. Finally, the third set of preconditions specifies well formedness of the singleton class `ISOException`. In general, Java Card does not support garbage collection, so, to avoid dynamic object allocation and discarding, the Java Card environment keeps single instances of each exception type and reuses them.

It took some trial and error steps to get all the preconditions right (we discuss this issue in detail in Section 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[5] with less that 10 000 proof steps. This proves that the `verifyPIN` method, given the preconditions, indeed can only throw `ISOException`.

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).

Let us show one more example of this property based on the `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. One of the methods in the applet is responsible for unregistering a given user ID. The method is called `dispatchDeleteKeyPair`. 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 marked as empty for future reuse. The actual proof obligation reads as follows (some things that have been shown already are marked with comments):

```
program variables {
```

[5]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.

```
  SafeApplet self;
  javacard.framework.APDU apdu;
  short expLen;
  boolean finishedWithISOEx;
  boolean finishedOK;
}

problem {
// APDUException well-formed
// ISOException well-formed
  & !self = null
  & !self.temp = null & self.temp.length = 200
  & expLen = 1 & !apdu = null
  & !apdu.buffer = null
  & apdu.buffer.length = expLen + ISO7816.OFFSET_CDATA
  & apdu.buffer[ISO7816.OFFSET_LC] = castToByte(expLen)
// User PIN well-formed and positively verified
  & !self.users = null & self.users.length = SafeApplet.MAX_USERS
  & all i:int. ((i >= 0 & i < SafeApplet.MAX_USERS) ->
              !self.users[i] = null)
-> <{ method-frame(SafeApplet()):{
       finishedWithISOEx = false; finishedOK=false;
       try {
         self.dispatchDeleteKeyPair(apdu);
         finishedOK = true;
       }catch(javacard.framework.ISOException e1){
         finishedWithISOEx = true;
       }
     } }> (finishedOK = TRUE |
    (finishedWithISOEx = TRUE &
     javacard.framework.ISOException.systemInstance.theSw[0]
       = SafeApplet.SW_USER_UNREGISTERED))
}
```

Among other things, the precondition says that the APDU that is a parameter to our method contains proper data (1 byte containing the user ID to be unregistered), and 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. That is why we had to use two local boolean variables: `finishedOK` and `finishedWithISOEx`. 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).

4.1.1 Proof Modularisation

This proof obligation is proved automatically with the KeY Prover in about 15 minutes and takes 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`, which in this case is 5. Secondly, the method performs a lot of preliminary work before the actual modification of the users table. 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 pre- and postcondition of the called method. In that case 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 [23, 7], i.e., 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, e.g., a call to a relatively small method may appear only once in a program, but, due to proof branching, it may appear multiple times in the proof. Thus, using method specification in the proof potentially avoids multiple symbolic execution of the same method. For comparison, we applied such modularisation to our last example—we used specification just for one method that contains a loop. The resulting proof took less that one minute with 5 000 proof steps, the side proof establishing that the method containing the loop satisfies its specification took less than 2 minutes with less than 12 000 rule applications—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, we may want to say that we disallow one type of exception in our program, while allowing all other kinds of exceptions. Following the same schema as before, the general property of this kind can be formalised as follows:

```
program variables {
  MyApplet self;
  boolean unwantedException;
}

problem {
   preconditions & unwantedException = FALSE ->
   <{ method-frame(MyApplet()): {
       try {
         self.aMethod();
       }catch(java.lang.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`. Since it seems obvious how to reuse previous examples to show that, e.g., no `NullPointerException` is thrown, we are not going to show any more examples of this property.

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 `TransactionExceptions` are only thrown when transactions are badly formed (i.e., so far we do not model transaction capacity), we can simplify this part of the property to "No `TransactionException` is thrown in the program." We have already shown how such a property is formalised and proved in previous sections.

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. In our model, the static attribute `transactionDepth` of the `JCSystem` class stores this information. It is quite straightforward to specify that a given method does not leave an open transaction:

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

The precondition says that there is no open transaction before `aMethod` is called. Such a precondition is necessary in case `aMethod` is considered to be 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. Alternatively, one can show that the transaction depth after executing the method is the same as before the execution. We will incorporate illustrating this property with a real example into the next Section, as it integrates nicely with the next property.

4.4 Atomic Updates

This property requires *related* persistent data in the applet to be updated atomically. As we stated already at the beginning of the paper, strong invariants 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. Hence, strong invariants seem to be the right technique to deal with consistency properties related to atomic updates.

We will illustrate this property briefly with the same example that is discussed in full in [14], for this work however we were able to use the real *Demoney* applet instead of the simplified one used in [14]. 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. The method that is responsible for debiting the purse balance and updating the log file is called `performTransaction` and uses Java Card transaction mechanism to ensure atomic update of the involved data. In Java Card DL, to specify that a property holds at all times, the throughout modality is used. Thus, the resulting proof obligation reads:

```
problem {
     JCSystem.transactionDepth = 0
   & !self = null & !apduBuffer = null
   & apduBuffer.length = 45
   & apduBuffer[ISO7816.OFFSET_LC] = DemoneyIO.COMPLETE_TRANSACTION_LC
   & offsetTransCtx = DemoneyIO.COMPLETE_TRANSACTION_OFF_TRANS_CTX
   & !self.logFile = null
   & !self.logFile.records = null
   ...
   & ex currentRecordPre:ArrayOfint.(
      currentRecordPre = self.logFile.records[
       (self.logFile.nextRecordIndex - 1) % self.logFile.records.length]
        & short_compose(
           currentRecordPre[DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE],
           currentRecordPre[DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE + 1]) =
          self.balance
     )
-> [[{method-frame(fr.trustedlogic.demo.demoney.Demoney()): {
         self.performTransaction (amount, apduBuffer, offsetTransCtx);
       }
   }]] all currentRecordPost:ArrayOfint.(
     currentRecordPost = self.logFile.records[
       (self.logFile.nextRecordIndex - 1) % self.logFile.records.length]
     ->
       short_compose(
         currentRecordPost[DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE],
         currentRecordPost[DemoneyIO.LOG_RECORD_OFF_NEW_BALANCE+1]) =
       self.balance
    )
}
```

The preconditions basically state that all the applet's data is properly formed and initialised. The main part of the specification is the strong invariant, which state that the current balance of the purse (`self.balance`) is equal to the one recorded in the most recent log entry (`short_compose...`). Our strong invariant occurs in two places, in the precondition and in the postcondition. The throughout modality requires the postcondition to hold in every intermediate state of execution of the program in the modality, including the initial state, thus, we need to assume that our strong invariant holds before the program is executed, and that is why the strong invariant is included in the precondition. The purchase log data structure in *Demoney* is basically two-dimensional byte array, where the first index points to a given log entry, and the second index points to the actual entry data. Since Java Card only allows only one-dimensional arrays, a workaround in the *Demoney* code has been introduced, namely, first a one-dimensional array of objects is allocated:

```
  Object[] records = new Object[...];
```

and then each entry in this array is associated with a byte array:

```
  records[i] = new byte[...];
```

Because of this, the `records` array lacks static type information. This results in (1) type casts in the *Demoney* code, and (2) necessity to express this hidden type information in our Java Card DL formulae. The way to do this is to use existential quantifiers in the preconditions and universal quantifiers in the postconditions, as in our example above. Those quantifier constructs are basically equivalents of type casts in Java Card DL.

Log records are stored in a cyclic file, i.e., the new entry overwrites the oldest one, thus, the need for cyclic indexing, using the modulo operator, of the array elements in the strong invariant.

The last element of the strong invariant to explain is the `short_compose` function symbol. It is an abstracted way to say that two `byte` values are composed to form a `short` value. This way one abstracts away from the actual Java Card Virtual Machine implementation of short data type (e.g., big or small endian) and avoids unnecessarily complicated Java integer expressions. Obviously, a small set of proof rules to deal with this abstracted representation is needed.

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 are not factored out by modularisation, so we consider this a relatively good result. Some modularisation using Java Card API specification has been used in the proof (e.g., a method specification for Java Card's `setShort` method, which makes use of the `short_compose` function symbol), however we have to point out here, that in case of proof obligations involving the throughout modality 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 the `performTransaction` method. Since a Java Card transaction is involved in this method it would be desirable to also show that no `TransactionException` is thrown and that no open transaction is left after this method is executed as stipulated in the previous Section. We intend to make this property even stronger and say that there is no exception thrown whatsoever. The proof obligation reads:

```
problem {
  // Mostly the same preconditions as before
  -> <{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 have also proved a similar consistency property about one of the methods in SafeApplet. There we specified that all the registered users have a properly defined set of private and public encryption keys at all times. The proof obligation is the following:

```
problem {
  // General preconditions
    !self = null & !apdu = null & !self.SafeApplet::temp = null
  & expLen = 1
  // APDUException, ISOException well formed
  // userPIN well formed
  & !self.userPIN = null & ...
  // General assumptions about the users table
  & !self.users = null
  & self.users.length = SafeApplet.MAX_USERS
  & all i:int.all j:int. (
     (i >= 0 & i < SafeApplet.MAX_USERS &
      j >= 0 & j < SafeApplet.MAX_USERS ) ->
     (!self.users[i] = null & !self.users[i].keydata = null &
         (!i=j ->
            (!self.users[i] = self.users[j] &
             !self.users[i].keydata = self.users[j].keydata &
             (
              (self.users[i].empty = FALSE &
               self.users[j].empty = FALSE) ->
                 !self.users[i].userID = self.users[j].userID
              )))))
  // Strong Invariant
  & all i:int.(
     i >= 0 & i < SafeApplet.MAX_USERS &
     self.users[i].empty = FALSE ->
      rsa_proper_key(self.users[i].keydata.privateExponent,
        self.users[i].keydata.publicExponent,
        self.users[i].keydata.modulus
      ) = TRUE)
-> [[{ method-frame(SafeApplet()):{
         self.dispatchGenerateKeyPair(apdu); }
   }]]
    // Strong Invariant
    all i:int.(
     i >= 0 & i < SafeApplet.MAX_USERS &
     self.users[i].empty = FALSE ->
      rsa_proper_key(self.users[i].keydata.privateExponent,
        self.users[i].keydata.publicExponent,
        self.users[i].keydata.modulus
      ) = TRUE)
```

The preconditions in the front are mostly the same as in the previous examples. The preconditions about the `users` table require more explanations. Due to lack of garbage collection the entries in the `users` table are reused, thus, each object of type `User` contains a boolean attribute `empty` to indicate if a given object is in use. Furthermore, we have to say that (1) each element in the `users` table contains a distinct `User` object, (2) users don't share key data objects, and (3) user IDs of non empty users are unique, i.e., a user with a given ID is registered only once. The strong invariant specifies that all non empty users contain a set of matching private and public keys at all times. The `rsa_proper_key` function symbol is used to specify that a key set contains matching keys. This function symbol has the same role as the `short_compose` function symbol in the previous example and is handled in a very similar way.

The actual implementation of the `dispatchGenerateKeyPair` does not use Java Card transactions to ensure data consistency. Instead, the method makes use of the `empty` attribute of each `User` object. When a new user is introduced, first the `User` object is initialised and then it is marked to be in use. When a user is deleted, the object is simply marked as empty. This way, the consistency property that applies to non empty objects only, holds at all times. However, such coding results in a more complex proof. Also, because of the numerous occurrences of quantifiers in the proof obligation, 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 purely related to integer arithmetic. It says that 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. 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, that is, 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.

Let us illustrate this with an example taken from the *SecSafe* document. First let us look at a proof obligation with a badly formed program with respect to overflow:

```
problem {
  inShort(balance) & inShort(maxBalance) & inShort(credit) &
  balance > 0 & maxBalance > 0 & credit > 0 ->
  <{ try {
       if (balance + credit > maxBalance)
         throw ie;
       else
         balance += credit;
     }catch(javacard.framework.ISOException e){}
  }> balance > 0
}
```

The problem in this program 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 in the following way:

```
...
      if (balance > maxBalance - credit)
...
```

This proof obligation is provable with both kinds of integer semantics. Further discussion about handling integer arithmetic in the KeY system can be found in [6].

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. This includes bounded dynamic memory allocation and no memory allocation in certain life stages of the applet. This seems like a problem strictly related to static analysis, because in general there is no need for precise analysis of the control flow, although in some cases such precise analysis would be required. For example, if memory allocation is performed inside a loop, the precise loop bound has to be known. 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. For example, each type of object contains an implicit reference `<next>`, which points to the object of the same type that was created next after this one—the Java Card DL rules that handle object creation are responsible for updating the state of the `<next>` reference in the proof. 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 seems to be a non trivial task and thorough treatment of this problem requires further research. For the moment, we would be only able to give approximate figures for memory consumption.

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 [11], 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

5.1 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. Getting the preconditions right 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 [16] 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 [16] 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 [25]. 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 failed proof obligations that are either small or at least contain 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, like the atomicity related properties (the throughout modality is partial), one should accompany such a proof with an additional termination property, like we did in Section 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 [23, 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.

5.2 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 (Section 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 number 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 [13]. 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] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, April 2004. Online First issue, to appear in print.

[2] Bernhard 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] Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Andreas Roth, Philipp Rümmer, and Steffen 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] Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In Mauro Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2621 of *LNCS*, pages 246–260, Warsaw, Poland, April 2003. Springer.

[5] Bernhard Beckert and Bettina 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] Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, April 2004.

[7] Bernhard Beckert and Peter H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.

[8] Robert Boyer. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.

[9] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Java Series. Addison-Wesley, June 2000.

[10] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proceedings, SPIN Software Model Checking Workshop*, LNCS, pages 205–223. Springer, 2000.

[11] Ádám Darvas, Reiner Hähnle, and Dave 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.

[12] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie 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.

[13] Tobias Gedell. Integrating static analysis into theorem proving. Available from `http://www.cs.chalmers.se/~gedell/publications/satp.ps`.

[14] Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *CASSIS'04 Post Workshop Proceedings*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[15] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[16] Bart Jacobs, Claude Marché, and Nicole 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.

[17] Bart Jacobs and Erik 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.

[18] Claude Marché, Christine Paulin-Mohring, and Xavier 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`.

[19] Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.

[20] Renaud Marlet and Daniel Le 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.

[21] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The JIVE system – Implementation description. Available from `http://softech.informatik.uni--kl.de/old/en/publications/jive.html`, 2000.

[22] Wojciech 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`.

[23] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[24] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[25] André Platzer. Using a program verification calculus for constructing specifications from implementations. Minor thesis, Karlsruhe University, Computer Science Department, February 2004.

[26] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.