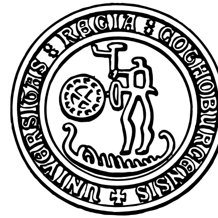
Thesis for the degree of Doctor of Philosophy

Formal Development of Safe and Secure Java Card Applets

Wojciech Mostowski

CHALMERS | GÖTEBORG UNIVERSITY



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

Göteborg, February 2005

Formal Development of Safe and Secure Java Card Applets
Wojciech Mostowski
ISBN 91-7291-575-7

Copyright © Wojciech Mostowski, 2005

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr 2257
ISSN 0346-718X

Technical Report 2D
Department of Computer Science and Engineering
Formal Methods Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31-772 1000

Cover: A schematic, SIM shaped Java Card device

Chalmers Reproservice
Göteborg, Sweden 2005

Formal Development of Safe and Secure Java Card Applets
Wojciech Mostowski
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

Abstract

This thesis is concerned with formal development of Java Card applets. Java Card is a technology that provides a means to program smart cards with (a subset of) the Java language. In recent years Java Card technology gained great interest in the formal verification community. There are two reasons for this. Due to the sensitive nature (e.g., security, maintenance costs) of Java Card applets, formal verification for Java Card is highly *desired*. Moreover, because of the relative simplicity of the programming language, Java Card is also a *feasible* target for formal verification. The formal verification platform that we used in our research is the KeY system developed in the KeY Project. One of the main objectives of our research is to find out how far formal verification for industrial size Java Card applets goes, in terms of usability, automation, and power (expressivity of constraints). Furthermore, we investigated practical and theoretical shortcomings of the verification techniques and development methods for Java Card applets. As a result, we adapted a program logic for Java Card to be able to express interesting, meaningful safety and security properties (*strong invariants*) and proposed design guidelines to support and ease formal verification (*design for verification*). We performed extensive practical experiments with the KeY system to justify and evaluate our work.

Formal aspects of our research concentrate on source code level verification of Java Card programs with interactive and automated theorem proving. Our work has been driven by certain assumptions, motivated by the KeY Project's philosophy: (1) formal verification should be accessible to software engineers without years of training in formal methods, (2) we should be able to perform full verification whenever needed, i.e., we want to handle complex Java Card applets that involve Java Card specific features, like atomic transactions and object persistency, (3) the verified code should not be subjected to translations, simplifications, intermediate representations, etc., and finally, (4) the properties that we prove should relate to important safety and security issues in Java Card development. We relate to these goals in our work.

Keywords: Java Card, object-oriented design, formal specification, formal verification, Dynamic Logic

List of Included Papers and Reports

This thesis is based on the publications listed below. My involvement in the work presented in these papers is described in detail in the thesis' introduction.

- [Mos02] 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.
- [BM03] 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, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.
- [LM04] Daniel Larsson and Wojciech Mostowski. Specifying JAVA CARD API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.
- [HM05] Reiner Hähnle and Wojciech Mostowski. Verification of Safety Properties in the Presence of Transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.
- [Mos05] Wojciech Mostowski. Formalisation and Verification of JAVA CARD Security Properties in Dynamic Logic. In Maura Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, LNCS. Springer, April 2005. To appear.

Table of Contents

Acknowledgements ix

Introduction 1

1 Overview . . . 1
2 Java Card . . . 2
3 The KeY System . . . 5
3.1 Architecture of the KeY Tool . . . 6
3.2 Dynamic Logic . . . 8
3.3 Syntax of Java Card DL . . . 9
3.4 Proof Obligations . . . 10
3.5 Deductive Calculus for Proving Obligations . . . 10
3.6 Taclets . . . 13
3.7 Implementation . . . 14
4 Description of the Papers . . . 14
5 Contributions . . . 18
6 Related Work . . . 20
7 Future Work . . . 23

Paper I: Rigorous Development of Java Card Applications 35

1 Introduction . . . 35
1.1 Java Card . . . 35
1.2 Analysis of the Current Situation . . . 36
1.3 Related Work . . . 37
1.4 Our Approach . . . 38
2 Case Study: `pam_iButton` . . . 39
3 Design Issues for Java Card Applications . . . 41
4 Developing Java Card Applications . . . 43
4.1 Applet Life States . . . 43
4.2 Applet Commands . . . 44
4.3 Command Invocation Protocol . . . 46
4.4 Command Processing . . . 48
5 The Framework . . . 53
5.1 Support from the CASE Tool . . . 54
5.2 Formal Specification and Verification . . . 54
5.3 Employing the KeY System . . . 57
6 Conclusions . . . 58

Paper II: A Program Logic for Handling Java Card's Transaction Mechanism **63**
1 Introduction . . . 63
2 Background . . . 64
3 Java Card Dynamic Logic . . . 67
3.1 Syntax of Java Card DL . . . 67
3.2 Semantics of Java Card DL . . . 68
3.3 State Updates . . . 69
3.4 Rules of the Sequent Calculus . . . 69
4 Extension for Handling "Throughout" and Transactions . . . 70
4.1 Additional Sequent Calculus Rules for the $[\![\cdot]\!]$ Modality . 71
4.2 Additional Sequent Calculus Rules for Transactions . . . 72
5 Examples . . . 76
6 Conclusions and Future Work . . . 80

Paper III: Specifying Java Card API in OCL **85**
1 Introduction . . . 85
2 Background . . . 86
2.1 The KeY Project . . . 86
2.2 Java Card and Java Card API . . . 87
2.3 Use Cases for OCL Specification of the Java Card API . 88
2.4 Related Work . . . 89
3 The Development of OCL Specification . . . 89
3.1 JML vs. OCL . . . 90
3.2 Exceptions . . . 91
3.3 The `null` value . . . 92
3.4 Integer Arithmetics . . . 92
3.5 JML `@assignable` clause . . . 93
4 The Specification . . . 93
4.1 Formal Verification . . . 97
5 Short Evaluation of OCL . . . 98
6 Conclusions . . . 98

Paper IV: Verification of Safety Properties in the Presence of Transactions **103**
1 Introduction . . . 103
2 Background . . . 104
2.1 The KeY Project . . . 104
2.2 Java Card Dynamic Logic . . . 105
2.3 Strong Invariants . . . 106
2.4 Java Card Atomic Transactions . . . 107
3 Case Study: Java Card Electronic Purse . . . 108
3.1 The `LogRecord` Class . . . 109
3.2 Specification and Verification of `setRecord` . . . 110
3.3 The `Purse` Class . . . 111
3.4 Specification and Verification of `processSale` . . . 111

3.5 *Post Hoc* Verification of Unaltered Code 111
3.6 Performance 113
4 Results 115
4.1 Verification Technology 115
4.2 Design for Specification and Verification 116
5 Related Work 119
6 Conclusions 121

Paper V: Formalisation and Verification of Java Card Security Properties in Dynamic Logic **129**
1 Introduction 129
2 Background 130
2.1 The KeY Project 130
2.2 Java Card 131
2.3 Java Card Dynamic Logic 132
2.4 Related Work 133
3 Case Studies 134
4 Security Properties 134
4.1 Only `ISOExceptions` at Top Level 135
4.2 No X Exceptions at Top Level 139
4.3 Well Formed Transactions 140
4.4 Atomic Updates 141
4.5 No Unwanted Overflow 144
4.6 Other Properties 145
5 Discussion 146
5.1 Lessons Learned 146
5.2 Static Analysis vs. Interactive Theorem Proving 148
6 Summary and Future Work 148

Acknowledgements

Believe it or not, but for me this the most difficult part of the thesis to write. The reason is that I will never know if I missed someone in the list below. But I will try to do my best to include everyone that deserves my gratitude.

Most of all, I would like to thank my Ph.D. supervisor, prof. Reiner Hähnle, for putting up with me and providing me with excellent guidance during my graduate studies at Chalmers. Without him, doing my research and writing this thesis would not be possible. I would also like to thank my opponent, prof. Arnd Poetzsch-Heffter, and the members of my grading committee, prof. Peter Dybjer, Mads Dam, and Erik Poll, for agreeing to participate in my Ph.D. defense. The members of my Ph.D. advising committee, prof. Mary Sheeran and Björn von Sydow, provided me with great support over the last four years.

Many people contributed to the quality of this thesis by commenting on the papers I (co-)wrote and giving me feedback on the thesis as such: prof. Reiner Hähnle, prof. Mary Sheeran, prof. Arnd Poetzsch-Heffter, Erik Poll, Steffen Schlager, Philipp Rümmer, and Martin Giese. At this point I would also like to thank my co-authors for excellent cooperation in writing papers: Bernhard Beckert, Daniel Larsson, and prof. Reiner Hähnle. Special thanks to Vladimir Klebanov for helping me give the thesis the "final touch".

Most of my work has been done in close cooperation with the KeY Project group. I would like to thank the following people, for always interesting discussions and for their friendship: prof. Peter H. Schmitt, Bernhard Beckert, Martin Giese, Wolfgang Ahrendt, Thomas Baar, Angela Wallenburg, Tobias Gedell, Daniel Larsson, Philipp Rümmer, Vladimir Klebanov, Andreas Roth, Steffen Schalger, and Richard Bubel. Special thanks to Andreas, Steffen, and Richard for exhibiting extreme patience in fulfilling my constant requests about missing features in the KeY system and fixing bugs.

Working in our department would not be possible without our great administrative staff. I would like to thank Jeanette Träff, Catharina Jerkbrant, Birgitta Magnusson, and Eva Löthman for all the help they have provided.

Life does not consist only of work. The list of people that made my social life in Sweden more enjoyable is very long. In particular my gratitude goes to my very good friend and former office mate, Angela Wallenburg, the current fifth floor corridor gang, Andrei Sabelfeld, Wolfgang Ahrendt, Rogardt Heldal, Reiner Hähnle, Philipp Rümmer, Martin Giese, and Jan-Willem Roorda, who sits just around the corner. Other people that specifically contributed to my life outside of work are Ádám Darvas, Erik Kilborn, Tuomo Takkula, Birgit Grohe, Boris Koldehofe, and Håkan Sundell. Tuomo, Birgit, Boris, and Håkan were

among the people I skied with (and/or I am about to ski with) on numerous occasions: Janna Khegai, Henrik Lindgren, Karin Hardell, Niklas Eén, Luige Eén, Jörgen Gustavsson, Jan-Willem Roorda, Alena Ostrovska, Karol Ostrovský, and Rogardt Heldal. Last in this list, but not least, I would like to thank all of my other office friends for making our department such a great place to work.

I would not have even thought of starting the Ph.D. studies in the first place, if I wasn't injected with the interest in research and formal methods by Tomasz Janowski during the pre-Chalmers era – many thanks for nursing me through the early stages of my research life.

Finally, I would be nowhere in my life without the love and support of my closest family. Most sincere thanks to Mama (also for proof reading large parts of this thesis!), Tata, and my dear brother Misio. Dziękuję Kochani!

Wojciech Mostowski

Göteborg
January 2005

Introduction

1 Overview

This thesis is concerned with formal development of Java Card applets. Java Card is a technology that provides a means to program smart cards with (a subset of) the Java language. In recent years Java Card technology gained great interest in the formal verification community. There are two main reasons for this. First of all, certain issues are critical in Java Card. Application areas like authentication or electronic cash require applets to be *safe* and *secure*. Also, because applets are usually distributed in large amounts, they are difficult and costly to *maintain*. Finally, *legal matters* may be critical in certain situations, for example, when the digital signature law is considered. Thus, formal verification of Java Card applets is highly *desired*. Secondly, due to the relative language simplicity, Java Card is also a *feasible* target for formal verification. The formal verification platform that we used in our research is the KeY system developed in the KeY Project[1] [ABB$^+$04]. One of the main objectives of our research is to find out how far formal verification, in particular using the KeY system, for industrial size Java Card applets goes, in terms of usability, automation, and power (expressivity of constraints). Furthermore, we investigated practical and theoretical shortcomings of the verification techniques and development methods for Java Card applets. As a result, we adapted the KeY system's Dynamic Logic to be able to express interesting, meaningful safety and security properties (*strong invariants* among others) and proposed design guidelines to support and ease formal verification (*design for verification*). We performed extensive practical experiments with the KeY system to justify and evaluate our work.

Formal aspects of our research concentrate on source code level verification of sequential Java programs, in particular Java Card, with interactive and automated theorem proving. Our work has been driven by certain assumptions, motivated by the KeY Project's philosophy. The basic one is that formal verification should be accessible to software engineers without years of training in formal methods. This turns out to be possible to achieve, however, in certain situations expertise in formal verification is required, in particular when verification of an already existing Java Card code is considered (*post hoc* verification). Another assumption is that we should be able to perform full verification whenever needed, i.e., we want to handle complex Java Card applets that involve

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

JAVA CARD specific features, like atomic transactions and object persistency. Moreover, the verified code should not be subjected to translations, simplifications, intermediate representations, etc. Finally, the properties that we prove should relate to important safety and security issues in JAVA CARD development. We relate to and comment on these goals throughout this work.

This work is a collection of papers that were published during the course of our research. The papers explore the subjects that we have just outlined. We explain the contents of these papers in detail later on in this chapter. First we start with some introductory material about JAVA CARD technology and the KeY system (Sections 2 and 3). We describe the papers included in this thesis in Section 4 and we discuss the contributions of this work in Section 5. Section 6 gives an overview of related work, and finally, Section 7 discusses possible future work.

2 JAVA CARD

Here we give a short, but for the purpose of this thesis complete, introduction to JAVA CARD technology and language [Che00, Sun03].

Smart Cards. Smart cards (chip cards) are small computers, providing 8, 16 or 32 bit CPU with clock speeds ranging from 5 up to 40 MHz, ROM memory between 32 and 64 KB, EEPROM memory (writable, persistent) between 16 and 32 KB and RAM memory (writable, non-persistent) between 1 and 4 KB. The ROM usually holds the card's operating system, the EEPROM is used to store persistent data of the applications residing on a smart card (for example, electronic cash) and the RAM is used for local computations. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816–4 standard). The communication is done in master-slave mode – it is always the master/terminal application that initialises the communication by sending a command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). There is no way for a smart card to initialise the communication (even though its CPU is active when the power is up), it can only reply to requests sent by the host system. APDUs are the only means to communicate with smart cards, which in practice means that the user of a smart card does not have any direct access to a smart card's "internals" (for example, there is no direct memory access).

JAVA Smart Cards. A smart card can be provided with functionality to run JAVA programs on it directly. Such cards are usually called JAVA powered cards (or simply JAVA cards) and the whole technology that provides JAVA functionality to smart cards (including a restricted subset of a JAVA language to program applets residing on a card) is called JAVA CARD. JAVA CARD's ROM, beside the operating system, includes a JAVA CARD virtual machine which implements the JAVA CARD language and allows applets to be run on the card.

Java Card Language Restrictions. Most of the Java Card language restrictions are related to the limited computing resources of smart cards. To start with, large primitive data types, like `int`, `long`, `double` or `float` are not available (although `int` is available on some Java Card platforms). Also characters, and thus strings, are excluded from the Java Card language. Furthermore multidimensional arrays, dynamic class loading, threads (concurrency) and garbage collection are not available in Java Card (again, garbage collection might be available on some platforms, but it is not required by the Java Card standard).

Otherwise Java Card is a fully functional Java with all object oriented features like interfaces, inheritance, virtual methods, overloading, dynamic object creation and scoping.

Java Card API and Applets. One other aspect where Java Card differs from Java is Java Card's API. The API is specific to the smart card environment and, thus, it provides support for handling APDUs, smart card Application IDentifiers (AIDs), PIN codes and Java Card specific system routines. Most of the "big" classes of the Java API, like `System`, `String` or `Vector`, are not available in Java Card.

The applications running in a Java Card environment are called Java Card applets. A proper applet should implement the `install` method responsible for the initialisation of the applet (one can see it as applet construction) and a `process` method for handling incoming command APDUs and sending the response APDUs back to the host. APDUs are the only means of communication with Java smart cards at the moment, but the upcoming versions of Java Card will also include support for Remote Method Invocation protocol. This will further decouple the hardware from the application level.

There can be more than one applet existing on a single Java Card, but there can be only one active at a time (the active one is the one most recently selected by the Java Card run-time environment).

Finally, we present a small example of a Java Card applet to show how applets work in practice. `CounterApplet` is an applet that simply returns the value of an internal counter when requested by the host. The counter is increased each time the applet is selected (activated) by the Java Card runtime environment. Here is the code:

```
import javacard.framework.*;

public class CounterApplet extends Applet {
    // CounterApplet APDU command codes
    final static byte CounterApplet_CLA = (byte)0xB2;
    final static byte GET_SELECT_COUNT = (byte)0x10;
    // (persistent) counter variable
    private byte counter;

    protected CounterApplet() {
        // applet initialisation
        counter = (byte)0;
```

```
        register();
    }

    public static void install(byte[] bArray,
      short bOffset, byte bLength) {
        new CounterApplet();
    }

    public void process(APDU apdu) {
        byte buffer[] = apdu.getBuffer();
        if ((buffer[ISO7816.OFFSET_CLA] == ISO7816.CLA_ISO7816) &&
            (buffer[ISO7816.OFFSET_INS] == ISO7816.INS_SELECT)) {
            // That was the SELECT APDU
            counter++;
        }else{
          if (buffer[ISO7816.OFFSET_CLA] != CounterApplet_CLA)
            ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
          if (buffer[ISO7816.OFFSET_INS] != GET_SELECT_COUNT)
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
          // That was the command to return the current value of
          // the counter.

          // Switch to 'send response' mode
          apdu.setOutgoing();
          apdu.setOutgoingLength((short)1);
          // Prepare the output buffer
          buffer[0] = counter;
          // Send the response
          apdu.sendBytes((short)0, (short)1);
        }
    }
}
```

During the initialisation of the applet (`install` method) a new instance of the applet is created, the counter is set to 0 and the applet is registered with the Java Card run-time environment. Then the `process` method takes care of processing the incoming APDUs. In case it is the select command APDU (the first `if` statement) the counter is increased. Otherwise the APDU is checked again whether it issues a command to return the current value of the counter. If that is the case the response is prepared and sent back to the host, otherwise an exception is thrown, which causes an APDU with a proper status word to be sent to the host informing that the request could not be handled.

Java Card Object Persistency, Atomicity and Transactions. Note that the value of the `counter` variable is not lost after the card's session is finished – all the instance variables of the applet are kept in the persistent (usually EEPROM) memory and, thus, their values are preserved when a power loss occurs. This is a specific feature of Java Card not present in Java.

When the execution of a Java Card applet is interrupted unexpectedly then all the updates to persistent objects performed so far are maintained. The atomicity level of the Java Card platform is quite "high" – all the updates to single variables and object fields are atomic. The user can perform atomic updates of a larger size using the transaction mechanism of Java Card. Inside a transaction the updates to persistent objects (only) are executed conditionally. When the transaction is committed, all the conditional updates are executed in one atomic step. In case the transaction is aborted (either by an abrupt termination of a program or by a system call) all the conditionally updated objects are rolled back to the state before the transaction started. Take a look at the following fragment of Java Card code:

```
counter = 100;
JCSystem.beginTransaction();
  counter = i;
  counter++;
  if(counter > 100)
    JCSystem.abortTransaction();
  else
    JCSystem.commitTransaction();
```

When the value of the counter inside the transaction rises above 100 the transaction will be aborted – the value of `counter` will be rolled back to its state before the transaction started, i.e., it will be 100 again. Otherwise the transaction will commit successfully and the value of `counter` will be equal to `i+1` after this piece of the program is executed.

3 The KeY System

The contents of this section is based on selected fragments of [ABB+04]. Here we explain the basic goals of the KeY Project and some technical details about the architecture of the KeY system, the Dynamic Logic used in interactive theorem proving, as well as some notes about the implementation and the current state of the system.

KeY is a tool for the development of high quality object-oriented software. The "KeY" idea behind this tool is to provide facilities for formal specification and verification of programs *within* a software development platform supporting contemporary design and implementation methodologies. The KeY tool empowers its users to perform formal specification and verification as part of software development based on the *Unified Modelling Language (UML)* [Obj03]. To achieve this, the system is realised as the extension of a commercial UML-based *Computer Aided Software Engineering Tool (CASE tool)*. As a consequence, specification and verification can be performed within the extended CASE tool itself. Such a deep integration of formal specification and verification into modern software engineering concepts serves two purposes. First, formal methods and object-oriented development techniques become *applicable* in a meaningful

combination. Second, formal specification and verification become more *accessible* to developers who are already using object-oriented design methodology. Moreover, KeY allows a *lightweight* usage of the provided formal techniques, as both, specification and verification, can be performed at any time, and to any desired degree.

UML based software development puts an emphasis on the activity of *designing* the targeted system. It is increasingly accepted that the design stage is very much where one actually has the power to prevent a system from failing. This suggests that formal specification and verification should (in different ways) be closely tied to the design phase, to design documents, and to design tools. One way of combining object-oriented design and formal specification is to attach *constraints* to *class diagrams*. An appropriate notation for such a purpose is already offered by the UML: the standard [Obj03] includes the *Object Constraint Language (OCL)* [WK03]. We briefly point out the three major roles of OCL constraints within KeY:

- The KeY tool supports the *creation* of constraints. While a user is free in general to formulate any desired constraint, he or she can also take advantage of the automatic generation of constraints, a feature which is realised in the KeY tool by extending the CASE tool's *design pattern instantiation* mechanism.

- The KeY tool supports the *formal analysis* of constraints. The relations between classes in the design imply relations between corresponding constraints, which can be analysed regardless of any implementation.

- The KeY tool supports the *verification* of implementations with respect to the constraints. A theorem prover with interactive and automatic operation modes can check consistency of JAVA implementations with the given constraints.

The target language of KeY-driven software development is JAVA. In particular, the verification facilities of KeY are focused on the JAVA CARD language [Che00, Sun03]. We have already given the description of JAVA CARD technology in the previous section. We have also pointed out why JAVA CARD is an *important* target for verification, and that the technical restrictions of JAVA CARD make verification of the full language *feasible*. However, KeY is not restricted to being used for the development of smart card applications – many features of JAVA that are not present in JAVA CARD are nevertheless supported by KeY, for example, `int` and `long` data types or characters and strings. In general, the KeY tool is applicable to most sequential JAVA programs.

3.1 Architecture of the KeY Tool

The KeY system is built on top of a commercial CASE tool. Integrating our system into an already existing tool has obvious advantages:

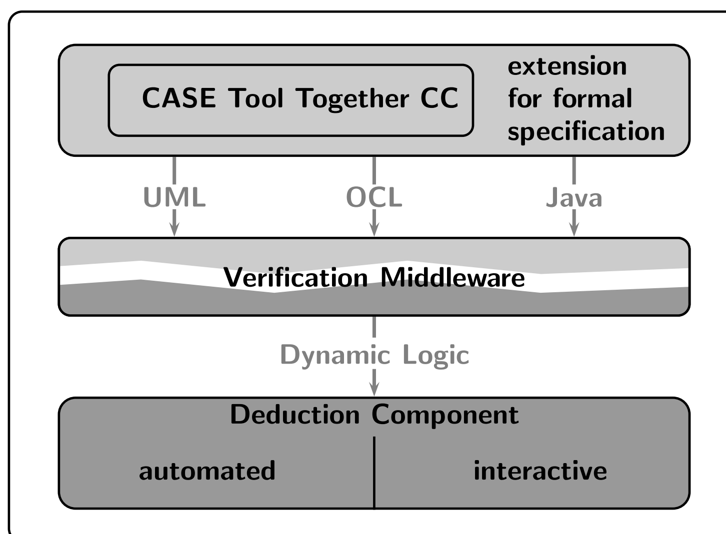


Figure 1. The architecture of the KeY system

1. All features of the existing tool can be used and do not need to be reimplemented.

2. The software developer does not have to become familiar with a new design and development tool. Furthermore, the developer is not required to change tools during development, everything that is needed is integrated into one tool.

A CASE tool that is well suited for our purposes has to be easily extensible and the extensions have to fit nicely into the tool providing a uniform user interface. At the moment we use *Together Control Center* from Borland.[2] Among all the tools on the market this one seems to be the most suitable for our purposes. It has state-of-the-art development and UML support (including some very basic support for textual specifications) and can be extended in almost any possible way by JAVA modules – TogetherCC offers access to most of its "internals" by means of a JAVA open API. There is, however, no fundamental obstacle to adding the KeY extensions to other, similar CASE tools, for example, work on integrating KeY into Eclipse[3] is underway.

The architecture of the KeY system is shown in Figure 1. In the following, we briefly describe the components and the interactions between them:

1. The *modelling component* (upper part in Figure 1) consists of the CASE tool with extensions for formal specification. While the CASE tool already

[2] `http://www.borland.com/together/`
[3] `http://www.eclipse.org`

allows the software model to contain OCL specifications, it does not have any support to create or process them in a formal way – OCL specifications are just textual annotations and are handled in the same way as comments. This is where the extension comes into play. It allows the user to create, process and prepare the OCL specifications (together with the model and its implementation) which can be later processed and passed to the deduction component. Manipulating OCL specifications is done by employing external programs and libraries as well as using TogetherCC's pattern mechanism to instantiate specifications from OCL specification templates [BHSS00, GHL04]. The CASE tool itself provides all the functionality for UML modelling and project development and is responsible for most of the user interactions with the project.

2. The *verification middleware* is the link between the modelling and the deduction component. It translates the model (UML), the implementation (Java) and the specification (OCL) into Java Card Dynamic Logic proof obligations which are passed to the deduction component. Java Card Dynamic Logic is a program logic used by the KeY prover (deduction component). The verification component is also responsible for storing and managing proofs during the development process.

3. The *deduction component* is used to construct proofs for Java Card Dynamic Logic proof obligations generated by the verification component. It is an interactive verification system combined with powerful automated deduction techniques. All those components are fully integrated and work on the same data structures.

3.2 Dynamic Logic

We use an instance of Dynamic Logic (DL) [KT90, HKT00] – which can be seen as an extension of Hoare logic – as the logical basis of the KeY system's software verification component. Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of Java. DL is used in the software verification systems KIV [BRS+00] and VSE [HLS+96] for (artificial) imperative programming languages. More recently, the KIV system supports also a fragment of the Java language [Ste01]. In both systems, DL was successfully applied to verify software systems of considerable size [BRS+00, Ste01].

DL can be seen as a modal logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of legal Java Card statements); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p. In standard DL there can be several such states (worlds) because the programs can be non-deterministic; but here, since Java Card programs are deterministic, there is exactly one such world (if p terminates) or there is no such world (if 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) [BS01].

The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\phi \rightarrow \langle p \rangle \psi$ is valid if for every state s satisfying pre-condition ϕ a run of the program p starting in s terminates, and in the terminating state the post-condition ψ holds.

Thus, the formula $\phi \rightarrow \langle p \rangle \psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators. In Hoare logic, the formulas ϕ and ψ are pure first-order formulas, whereas in DL they can contain programs. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Also, all Java constructs are available in our DL for the description of states (including `while` loops and recursion). It is, therefore, not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows one to concentrate on the relevant properties of a state.

3.3 Syntax of Java Card DL

As mentioned above, a dynamic logic is constructed by extending some non-dynamic logic with a modal operator $\langle \cdot \rangle$. In addition, we use the dual operator $[\cdot]$, for which $[p]\phi \equiv \neg\langle p \rangle \neg\phi$. The non-dynamic base logic of our DL is typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical to the Java types) nor how exactly terms and formulas are built. The definitions can be found in [Bec01]. Note that terms (which we often call "logical terms" in the following) are different from Java expressions; the former never have side effects.

In order to reduce the complexity of the programs occurring in formulas, we introduce the notion of a *program context*. The context can consist of any Java Card program, i.e., it is a sequence of class and interface definitions. Syntax and semantics of Java Card DL formulas are then defined with respect to a given context; and the programs in Java Card DL formulas are assumed not to contain class definitions.

The programs in Java Card DL formulas are basically executable statements of Java Card code. The verification of a given program can be thought of as *symbolic code execution* [HK76]. As will be detailed below, each rule of the calculus for Java Card DL specifies how to execute one particular Java statement, possibly with additional restrictions. When a loop or a recursive method call is encountered, it is necessary to perform induction over a suitable data structure.

Given that we follow the symbolic execution paradigm for verification, it is evident that a certain amount of runtime infrastructure must be represented in Java Card DL. It would be possible, but clumsy and inefficient, to achieve this by purely logical means. Therefore, we introduced an additional construct for handling of method calls that is not available in plain Java Card. Methods are invoked by syntactically replacing the call by the method's implementation. To

handle the `return` statement in the right way, possible exceptions, and dynamic binding it is necessary (a) to record the object field or variable x that the result is to be assigned to, (b) to record the object o and its type C that the method is applied to, and (c) to mark the boundaries of the implementation *body* when it is substituted for the method call. For that purpose, we allow statements of the form `method-frame(`x`,`C`(`o`)){`*body*`}` to occur in JAVA CARD DL programs. In practice, before the `method-frame` construct is introduced in a proof, rules that establish the run-time type of o and factor out method calls appearing inside expressions are applied first. Note, that the `method-frame` construct is a "harmless" extension because this construct is only used for proof purposes and never occurs in the verified JAVA CARD programs.

3.4 Proof Obligations

Let us now turn to the translation of OCL constraints into JAVA CARD DL proof obligations. To prove that a method $\mathtt{m(arg_1,\ldots,arg}_n\mathtt{)}$ of class `C` satisfies a pre-/post-condition pair, the OCL conditions are first translated into first-order formulas $pre(\mathtt{self}, \mathtt{arg}_1, \ldots, \mathtt{arg}_n)$ and $post(\mathtt{self}, \mathtt{arg}_1, \ldots, \mathtt{arg}_n)$, respectively. From these formulas, KeY constructs the JAVA CARD DL proof obligation:

$$pre(\mathtt{self}, \mathtt{arg}_1, \ldots, \mathtt{arg}_n) \rightarrow \\ \langle\, \mathtt{self.m(arg_1,\ldots,arg}_n\mathtt{);}\,\rangle post(\mathtt{self}, \mathtt{arg}_1, \ldots, \mathtt{arg}_n)$$

where now `self` and $\mathtt{arg}_1, \ldots, \mathtt{arg}_n$ are program variables, which are implicitly universally quantified with respect to their initial value.

Similarly, to prove that a method $\mathtt{m(arg_1,\ldots,arg}_n\mathtt{)}$ preserves an invariant, the proof obligation

$$(inv(\mathtt{self}) \wedge pre(\mathtt{self}, \mathtt{arg}_1, \ldots, \mathtt{arg}_n)) \rightarrow \\ \langle\, \mathtt{self.m(arg_1,\ldots,arg}_n\mathtt{);}\,\rangle inv(\mathtt{self})$$

is constructed, where $inv(\mathtt{self})$ is the first-order translation of the invariant.

3.5 Deductive Calculus for Proving Obligations

As usual for deductive program verification, we use a sequent-style calculus. A *sequent* is of the form $\Gamma \vdash \Delta$, where Γ, Δ are duplicate-free lists of formulas. Intuitively, its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

Rules of a sequent calculus are often represented by rule schemata, such as the example rules that will follow. In the KeY system, rules are implemented using the taclet mechanism (see the next section).

A *proof* for a goal (a sequent) S is an upside-down tree with root S. In practice, rules are applied from bottom to top. That is, proof construction starts with the initial proof obligation at the bottom and ends with axioms (rules with an empty premise tuple). In the following we describe some exemplary rules of the calculus. The whole JAVA CARD DL calculus contains (at least) one rule for each JAVA CARD programming construct, in total there are about 250 rules for handling the JAVA part of the logic.

The Active Statement in a Program

The rules of our calculus operate on the first *active* command p of a program $\pi p\omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.[4] The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following JAVA block operating on its first active command "`i=0;`", then the non-active prefix π and the "rest" ω are the indicated parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

The Assignment Rule and Handling State Updates

In JAVA (like in other object-oriented programming languages), different object variables can refer to the same object. This phenomenon, called aliasing, causes serious difficulties for handling of assignments in a calculus for JAVA CARD DL.

For example, whether or not a formula "$\texttt{o1.a} = 1$" still holds after the (symbolic) execution of the assignment "`o2.a = 2;`", depends on whether or not `o1` and `o2` refer to the same object.

Therefore, JAVA assignments cannot be symbolically executed by syntactic substitution. Solving this problem naively – by doing a case split if the effect of an assignment is unclear – is inefficient and leads to heavy branching of the proof tree.

In our JAVA CARD DL calculus we use a different solution. It is based on the notion of *updates*. These (state) updates are of the form $\{loc := val\}$ and can be put in front of any formula. The semantics of $\{loc := val\}\phi$ is the same as that of $\langle\texttt{loc = val;}\rangle\phi$. The difference between an update and an assignment is syntactical. The expressions *loc* and *val* must be simple in the following sense: *loc* is (a) a program variable `var`, or (b) a field access `obj.attr`, or (c) an array access `arr[i]`; and *val* is a logical term (that is free of side effects). More complex expressions are not allowed in updates.

The syntactical simplicity of *loc* and *val* has semantical consequences. In particular, computing the value of *val* has no side effects. The KeY system uses special simplification rules to compute the result of applying an update to logical terms and formulas not containing programs. Computing the effect of an update to a program p (and a formula $\langle p\rangle\phi$) is delayed until p has been symbolically

[4] In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form $\langle pq\rangle\phi$ can be replaced by $\langle p\rangle\langle q\rangle\phi$. In our calculus, splitting of $\langle \pi pq\omega\rangle\phi$ into $\langle\pi p\rangle\langle q\omega\rangle\phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle\pi p\omega\rangle\langle\pi q\omega\rangle\phi$ cannot be used either because its semantics is in general different from that of $\langle\pi pq\omega\rangle\phi$.

executed using other rules of the calculus. Thus, case distinctions are not only delayed but they can often be avoided completely, because (a) updates can be simplified *before* their effect is computed and (b) their effect is computed when a maximal amount of information is available (namely after the symbolic execution of the program).

The assignment rule now takes the following form ($\mathcal{U}$ stands for an arbitrary sequence of updates):

$$\frac{\Gamma \ \vdash \ \mathcal{U}\{loc := val\}\langle\, \pi\ \omega\,\rangle\phi}{\Gamma \ \vdash \ \mathcal{U}\langle\, \pi\ \texttt{loc = val;}\ \omega\,\rangle\phi} \tag{R1}$$

That is, it just adds the assignment to the list of updates $\mathcal{U}$. Of course, this does not solve the problem of computing the effect of the assignment. This problem is postponed and solved by rules for simplifying updates.

This assignment rule can, of course, only be used if the expression *val* is a logical term. Otherwise, other rules have to be applied first to evaluate *val* (as that evaluation may have side effects). For example, these rules replace the formula $\langle\, \texttt{x = ++i;}\,\rangle\phi$ with $\langle\, \texttt{i = i+1; x = i;}\,\rangle\phi$. One can view these rules as on-the-fly program transformations. Their effect is always local and fairly obvious, so that the user's understanding of the proof is not obfuscated.

The Rule for if-else

One more example that shows how the calculus corresponds to symbolic program execution, is the rule for the `if` statement:

$$\frac{\Gamma,\, \mathcal{U}(b \doteq \mathsf{true}) \ \vdash \ \mathcal{U}\langle\, \pi\ p\ \omega\,\rangle\phi \qquad \Gamma,\, \mathcal{U}(b \doteq \mathsf{false}) \ \vdash \ \mathcal{U}\langle\, \pi\ q\ \omega\,\rangle\phi}{\Gamma \ \vdash \ \mathcal{U}\langle\, \pi\ \texttt{if(}b\texttt{)}\ p\ \texttt{else}\ q\ \omega\,\rangle\phi} \tag{R2}$$

The two premises of this rule correspond to the two cases of the `if` statement. The semantics of rules is that, if all the premises are true in a state, then the conclusion is true in that state. In particular, if the premises are valid, then the conclusion is valid. As the rule demonstrates, applying it (from bottom to top) corresponds to a symbolic execution of the program to be verified.

The Rules for try/throw

Finally, the following rules allow to handle `try-catch-finally` blocks and the `throw` statement. These are simplified versions of the actual rules that apply to the case where there is exactly one `catch` clause and one `finally` clause.

$$\frac{\Gamma \ \vdash \ \mathit{instanceof}(exc, T) \qquad \Gamma \ \vdash \ \langle\, \pi\ \texttt{try\{e=exc;}\ q\texttt{\}finally\{}r\texttt{\}}\ \omega\,\rangle\phi}{\Gamma \ \vdash \ \langle\, \pi\ \texttt{try\{throw exc;}\ p\texttt{\}catch(}T\ \texttt{e)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\,\rangle\phi} \tag{R3}$$

$$\frac{\Gamma \ \vdash \ \neg\mathit{instanceof}(exc, T) \qquad \Gamma \ \vdash \ \langle\, \pi\ r\texttt{; throw exc;}\ \omega\,\rangle\phi}{\Gamma \ \vdash \ \langle\, \pi\ \texttt{try\{throw exc;}\ p\texttt{\}catch(}T\ \texttt{e)\{}q\texttt{\}finally\{}r\texttt{\}}\ \omega\,\rangle\phi} \tag{R4}$$

The predicate $\mathit{instanceof}(exc, T)$ has the same semantics as the `instanceof` operator in Java. It evaluates to *true* if the value of *exc* is assignable to a program variable of type T, i.e., if its dynamic type is a sub-type of T.

Rule (R3) applies if an exception *exc* is thrown that is an instance of exception class T, i.e., the exception is caught; otherwise, if the exception is not caught, rule (R4) applies.

3.6 Taclets

Most existing interactive theorem provers are "tactical theorem provers". The tactics for which these systems are named are programs which act on the proof tree, mostly by many applications of primitive rules, of which there is a small, fixed set. The user constructs the proof by selecting the tactics to run. Writing a new tactic for a certain purpose, for example, to support a new data type theory requires expert knowledge of the theorem prover.

In the KeY prover, both tactics and primitive rules are replaced by the *taclet* concept [BGH+04, Hab00]. A taclet combines the logical content of a sequent calculus rule with pragmatic information that indicates when and for what it should be used. In contrast to the usual fixed set of primitive rules, taclets can easily be added to the system. All the Java Card DL calculus rules are implemented in the form of taclets. Moreover, since rules can be dynamically added to the prover, it is possible to have alternative sets of rules for certain purposes. For example, when dealing with integer arithmetics, different integer semantics can be chosen to handle all possible overflow scenarios: ideal integer arithmetics (no overflow), finite integer types (overflow prohibited), or Java integer semantics (types are finite, overflow is allowed and modelled accurately as it happens in JVM) [BS04].

Taclets are formulated as simple pattern matching and replacement schemas. For instance, a typical taclet might read as follows:

```
find (b -> c ==>)
if (b ==>) replacewith (c ==>)
heuristics(simplify)
```

This means that an implication `b -> c` on the left side of a sequent may be replaced by `c`, if the formula `b` also appears on the left side of that sequent. The clause `heuristics(simplify)` indicates that this rule should be part of the heuristic named `simplify`, meaning that it should be applied automatically whenever possible if that heuristic is activated.

While taclets can be more complex than the typically minimalistic primitive rules of tactical theorem provers, they do not constitute a tactical programming language. There are no conditional statements, no procedure calls and no loop constructs. This makes taclets easier to understand and easier to formulate than tactics. In conjunction with an appropriate mechanism for application of heuristics, they are nevertheless powerful enough to permit interactive theorem proving in a convenient and efficient way.

In principle, nothing prevents one from formulating a taclet that represents an unsound proof step. It is possible, however, to generate a first-order proof obligation from a taclet, at least for taclets not involving programs. If that formula can be proven using a restricted set of "primitive" taclets, then the

new taclet is guaranteed to be a correct derived rule. As for the primitive taclets for handling Java programs in Java Card DL, it is possible to show their correctness using the Isabelle formalisation of Java by Oheimb [vO01].

3.7 Implementation

The KeY tool is an extension of the commercial case tool TogetherCC. The open API of TogetherCC allows the KeY tool to add items to the CASE tool's contextual menus for classes, methods, etc. The API also makes it possible to modify the currently open UML model. All KeY system extensions can optionally be switched on and off in TogetherCC and thus it is the developer's decision to use them or not.

The KeY tool is implemented in the Java programming language. This choice has several advantages, besides the obvious one of portability. Using the Java language makes it easy to link the KeY tool to TogetherCC, which is also written in Java. More generally, Java is well suited for interaction with other tools, whether they are written in Java or not. In particular, the imperative nature of the language leads to a comparatively natural native code interface, in contrast to the logic or functional programming languages often preferred for deduction purposes.

Java was also a good choice for the construction of the graphical user interface, which is an important aspect of the KeY tool. Finally, previous experiments with both interactive [Hab00] and automated [Gie01] theorem proving have shown that the advantages of Java outweigh the additional effort for the implementation of term data structures, unification, etc.

Some components of the KeY tool can be used stand alone. In particular, it is possible to use the KeY prover to prove properties about Java programs without the presence or availability of the CASE tool. The KeY system is still being developed; however it can already handle all of Java Card features, including Java Card transactions and object persistency. The current publicly available version of KeY can be downloaded from the KeY Project web page.[5] Development versions are available on request.

4 Description of the Papers

In the following we give a description of the papers included in this thesis. All the papers have been revised and are self contained. Thus, some introductory material in the papers is repeated and overlaps with the material already presented in this introduction. For each of the papers we shortly present its contents, describe its relation to this thesis and the author's involvement in the presented work.

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

Paper I: Rigorous Development of Java Card Applications

This paper is the starting point to develop a rigorous, well defined development process for Java Card applications with formal methods support. It is based on a real-life case study – a system for authenticating users in the Linux system with Java Powered iButtons[6] instead of the password mechanism. The first step was to identify all the problems and deficiencies of the Java Card applet caused by an unorganised development process and "too relaxed" use of the Java Card language. Then a development process is proposed, which aims at overcoming the problems discovered. Among other things, it ensures a well defined and self-controlled (by the applet itself) message exchange protocol disallowing tampering with the applet, it enforces integrity checks on the incoming APDU data (wherever necessary) and constrained usage of memory making the applet "memory safe". The development process is based on UML and OCL giving the basis for formal verification. The new development process applied to the case study (i.e., the case study was reengineered) produced a robust, self-controlled, memory safe Java Card applet. Moreover, most of the actual applet code was derived from UML/OCL specifications.

The paper also describes the problem of a card "rip-out" in some detail. The problem occurs when the execution of the applet is abruptly terminated by ripping out the card from the reader (terminal) or simply by power loss. In such a case the applet's memory may be left in an undefined state, disabling the proper functioning of the applet in some cases. To handle this problem one needs to be able to specify and prove a property that should hold throughout the whole execution of a Java Card program, so that in case of a "rip-out" at any point the property is maintained. The theoretical aspects of handling "rip-out" properties is the subject of the next paper.

The work described in this paper was carried out solely by the author. Originally, this paper was presented at the Rigorous Object Oriented Methods Workshop in London, March 2002 [Mos02]. Later it was submitted to the Software and Systems Modelling Journal and it is currently undergoing the review process.

Paper II: A Program Logic for Handling Java Card's Transaction Mechanism

This paper extends the Java Card Dynamic Logic used in the KeY system's interactive prover to handle the mentioned "rip-out" properties. The new modal operator "throughout" is introduced to the logic, which can be used to prove that a property holds throughout the whole execution of a Java Card program (in all the intermediate states). We call the "rip-out" properties *strong invariants*. Such an invariant on the objects' data is maintained at any time during applet execution and, in particular, in case of abrupt termination. The main challenge in this work was to handle Java Card's transaction mechanism

[6] "iButtons" are particular Java Card devices embedded in a button shaped case, see `http://www.ibutton.com/`.

in connection with object persistency (which is specific to Java Card) in the sequent calculus rules. It should be noted here that transactions and object persistency affect the semantics of the $\langle\cdot\rangle$ and $[\cdot]$ modal operators (specifically programmatic abortion of a transaction) – the necessary rules to handle transactions for diamond and box operators are also presented in the paper. The paper also contains examples of simple proofs using the new rules. To the best of our knowledge, currently there is only one other research effort to deal with Java Card transactions and "rip-out" properties in Java Card program verification, see Section 6.

This paper was presented (and published in the proceedings) at the Fundamental Approaches to Software Engineering Conference (part of ETAPS) held in Warsaw, April 2003 [BM03]. The theoretical aspects of introducing the extensions to the Java Card DL calculus were carried out in close cooperation with Bernhard Beckert from the University of Karlsruhe, Germany[7] during the author's six months research visit to Karlsruhe. Later on, the author implemented all the extensions in the KeY prover by himself. The version of the paper included in this thesis is a slightly extended version of the one published in the conference proceedings.

Paper III: Specifying Java Card API in OCL

After implementing the "throughout" and transactions extensions the author engaged in practical experiments. At that point it was already known that to properly verify real Java Card applications one needs a full Java Card API specification. Such a specification is used in proofs to handle method calls to the Java Card library. In particular, when the implementation of a given method in the Java Card API is not known, the prover should "replace" the method call with the specification of that method. Since OCL is the main design and specification language in KeY, it was a natural decision to develop the specification of the Java Card API in OCL. The starting point for this work was the already existing specification of the API written in JML (Java Modelling Language) [LBR99] developed for the LOOP tool (see Section 6). The paper reports on the problems that were encountered when writing the specification and their solutions. This work was also treated as a Java Card verification case study for the KeY system, successful verification attempts of some small parts of the reference implementation of the API library are reported. Finally, the paper evaluates OCL and compares it to JML in the context of Java Card program specification and verification.

Later on, when the KeY prover was equipped with the ability to use OCL method specifications (or method contracts, as they are referred to in KeY) in proofs, certain practical deficiencies of OCL have been discovered. In short, proper verification of Java Card programs using method contracts requires attribute modification information to be given in the specification [Mül01, BS03].

[7] Currently at the Department of Computer Science, University of Koblenz-Landau, Germany, e-mail: `beckert@uni-koblenz.de`

Pure OCL does not support this, KeY introduces its own OCL extensions to handle such information ("modifies" clauses). Yet another problem is OCL's limited capability to deal with primitive arrays, which are the basis of the Java Card communication and API. Finally, the OCL parser (external) implementation that KeY uses, does not handle Java package information. Given all this, it turned out that OCL in its pure form is too high level for low level applications like Java Card. Currently, the experience gathered during this work serves as a case study to develop a more flexible OCL parser and type checker[8] [Joh04]. Such a general purpose parser could be used to interface OCL to a formal verification tool like KeY. The last paper of this thesis gives some brief insights on how we have overcome the deficiencies of OCL by defining method contracts directly in Java Card DL.

The work described in this paper has been carried out mostly by Daniel Larsson[9] as his Master's Thesis project [Lar03]. He has been supervised by the author of this thesis. Together, we wrote a paper based on Daniel's work, which was presented at the OCL 2.0 Satellite Workshop of the UML Conference, held in San Francisco, October 2003 [LM04]. Although the author's involvement in the actual development of the Java Card API specification was limited, we still consider this work an important element of further research that took place. Therefore, we decided to include this paper in the thesis.

Paper IV: Verification of Safety Properties in the Presence of Transactions

This paper presents the first practical case study, based on a real Java Card application, that involve strong invariants and Java Card transactions. Some difficulties has been discovered when trying to prove safety properties related to transactions. It is argued that, to overcome such difficulties, formal verification has to be taken into account during the design and coding phase of the Java Card applet development. This gives raise to practical Java Card *design for verification* guidelines. As a result of applying such guidelines, safety properties for Java Card applets can be proven easily – automatically and in a relatively short time. To support this claim, benchmarks of the KeY system are given in the paper.

All the practical work described in this paper was done by the author of this thesis. The author's supervisor, prof. Reiner Hähnle,[10] was invited to the CASSIS (Construction and Analysis of Safe, Secure and Interoperable Smart devices) Workshop (Marseille, March 2004), where he presented this work. The paper that we have written together, was accepted to be published in the post-workshop LNCS proceedings [HM05].

[8] `http://www.cs.chalmers.se/~krijo/ocltc/`
[9] Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: `danla@cs.chalmers.se`
[10] Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: `reiner@cs.chalmers.se`

Paper V: Formalisation and Verification of Java Card Security Properties in Dynamic Logic

This paper presents how common Java Card security properties can be formalised in Dynamic Logic and verified with the KeY system. The properties under consideration originate from [MM01], a document listing Java Card security properties to be studied in the SecSafe Project.[11] It consists of ten security properties that are of importance to the smart card industry. In the paper we show how we formalised and verified a large part of those properties. For the remaining properties we give concrete suggestions on how they can be treated in interactive theorem proving. The formalisation of the properties is accompanied by concrete examples based on real-life, industrial size, Java Card applications. The first one is the authentication applet described in the first paper of this thesis. The second is an electronic purse applet *Demoney*, provided to us by Trusted Logic.[12] The *Demoney* applet is also the motivating example for the previous paper. The SecSafe Project focuses on applying static analysis techniques to establish the Java Card security properties. We managed to formalise the properties in Java Card DL and verify them with the KeY prover. Therefore, we give an assessment of interactive theorem proving as an alternative to static analysis. Finally, all the verification described in the paper, was performed on an *unaltered* Java Card code, which was not designed to support verification (i.e., we used Java Card code "as is"). Thus, we discuss the practical experience that we gathered during the course of this work. The main conclusion is that, for the scenario of Java Card applet verification, while automation is still achievable and the time requirements stay within reasonable limits, certain user expertise is required.

The work described in this paper was carried out solely by the author. The paper was accepted to be published in the proceedings of the Fundamental Approaches to Software Engineering Conference, to be held in Edinburgh, Scotland, April 2005 [Mos05]. The version of the paper included in this thesis is an extended version of the conference paper, published as a technical report [Mos04].

5 Contributions

The work presented in this thesis brings a number of benefits to the area of formal Java Card development and, in particular, to the KeY Project. To start with, the development process for Java Card has been investigated. Together with the *design for verification* guidelines presented in the fourth paper and other case study work (for example, see [BH04]) it gives a solid basis for defining a general formal verification oriented development process that would bring formal methods to contemporary software engineering.

[11] `http://www.doc.ic.ac.uk/~siveroni/secsafe/`

[12] `http://www.trusted-logic.com`. We thank Renaud Marlet for providing the *Demoney* source code.

Secondly, a lot of practical experiments and formal verification have been performed during the course of this work. In particular, this gives the KeY system a real usability test in the context of industrial size Java Card applets. Moreover, we believe that the practical experience that we gathered could be useful for Java Card verification attempts using other tools as well. Our experiments have been driven by the following aspects: expressiveness, automation, and performance. To enhance expressiveness the Java Card DL has been extended with the throughout modality, which enables us to express and verify Java Card specific properties (strong invariants for rip-out situations). The ideas behind our extensions to handle the throughout modality and transactions could be used in other Hoare-like logics for Java Card programs. The last two papers of this thesis show that a tool equipped with such extensions can be used to verify many of the interesting Java Card safety and security properties.

The main obstacle to reach full automation in the verification process are loops and recursion. In principle, handling loops and recursive method calls requires using the induction rule in the proof. Although work on supporting inductive proofs is underway [HW03, Wal04], induction still requires user interaction with the prover. However, in practice, recursion is non existent in Java Card programs, so this is not considered to be a problem. As for loops, the situation is also manageable. In Java Card, most of the loops have constant bounds and can be symbolically executed step by step without user interaction (although the proofs tend to grow large). Moreover, the loops that are not bounded by a constant, are usually in a special form, i.e., each execution of loop body is independent of the other loop body executions (for example, simple array traversal or initialisation). In such situations, loops can be handled by a dedicated loop rule. Our practical experience regarding the loops currently serves as a basis for research on such Java Card specific loop handling. What also helps with loops, is the proper design and coding of the routines involving loops. Such design and coding guidelines should be incorporated into the general Java Card development process described above.

Our research has identified performance bottlenecks of the KeY prover. One of the bottlenecks is the handling of loops described above. The other one is the efficient use of Java Card API method contracts in proofs. The current contract mechanism implemented in KeY turns out not to be flexible enough in some situations. Our research has identified the deficiencies in this respect and issues that need to be addressed in the implementation of the contract mechanism to reach full support for Java Card specific method contracts. Such improvements are currently underway. Despite the discovered performance deficiencies, we found the KeY prover flexible enough to handle complex Java Card applets, mostly automatically, in reasonable time. Given that, according to the literature [PvdBJ00, JMR04], other Java Card verification tools support method contracts quite well, we consider regular use of formal methods in software engineering, and, in particular, Java Card development, absolutely feasible in the near future.

Finally, we introduced the throughout modality for the sole purpose of handling the rip-out properties. However, there is no obstacle to introduce other

temporal modal operators to the Java Card DL. The theoretical research regarding the transaction mechanism, and the schematic modal operators framework implemented by us in the KeY prover, make implementing a new modality in the prover literally a one day process, given that the set of the rules for the new modality is known. Additional temporal operators could be used to specify and prove correctness of non-terminating programs, for example, control software in Java2 Micro Edition devices. Given the close relation of the Dynamic Logic to Hoare calculus and relative genericity of our solution, we believe that extensions to handle temporal properties could also be added to other Java Card verification tools in a similar fashion.

6 Related Work

In recent years formal approaches to Java Card development gained considerable interest, both in academia and industry. Work in Java Card verification can be classified according to several criteria. Working on byte code avoids the problems of source code availability and compiler trustworthiness, but makes full verification more difficult due to information loss during compilation. An overview of work done on the byte code level is provided in [Boy03]. For us the most interesting projects and approaches are those that aim at source code verification of Java and Java Card programs. To start with, [HM01] contains an overview of the existing literature (until end of year 2001) on Java and Java Card safety with emphasis on formal approaches (including the ones targeted at byte code level). Here we list the most relevant (in our opinion) tools and approaches to formal verification of Java and Java Card programs.

The iContract tool [Kra98, Ens01], based on the idea of Design-By-Contract [Mey92], represents the most lightweight category of tools to enforce reliability of Java programs. In this approach, a Java program is decorated with annotations from a restricted language (based on OCL). Such specifications are checked during runtime, giving raise to exceptions when the specification is violated. Some other tools for Java based on the idea of run-time assertion checking are Jass[13] (uses a language resembling JML), the JML compiler – `jmlc` [CL02a] (assertions are written in a subset of JML), or `jmlunit` [CL02b], which combines JML assertion checking with unit testing [BG98].

The ESC/Java tool [FLL^{+}02] also relies on annotations from a restricted specification language, based on JML. The tool employs extended static checking techniques to validate Java programs. Annotated programs (via an intermediate representation) undergo a dynamic analysis that produces first-order verification conditions for a theorem prover (Simplify). The analysis does not attempt to be sound or complete, but it is fully automatic and produces warnings, when annotations are potentially violated. With ESC/Java it is possible to express and check properties similar to our strong invariants – an arbitrary code location can be annotated with an object invariant [LS97]. Annotating every program statement with an object invariant is an equivalent of expressing

[13] `http://csd.informatik.uni-oldenburg.de/~jass/`

a strong invariant. The development of ESC/Java2, the successor of ESC/Java, started at the University of Nijmegen, Netherlands, as part of the VerifiCard Project[14] [JMP01]. The following tools that we discuss are also related to this project.

The Jive system [MPH00, MMPH00] uses an extended Hoare-style calculus [PHM99] to prove properties about programs written in subdialect of Java Card (Diet Java Card). Jive is an interactive theorem prover with a dedicated user interface that uses an associated general purpose theorem prover (Isabelle [Pau94] or PVS [ORSvH95]).

The LOOP tool [JP03] employs a Hoare-like logic formalised in Type Theory [JP01, vdBHJP00, HJ00] to verify sequential Java programs. The backend theorem prover of the LOOP tool is PVS and the input specification language is JML. The LOOP tool is capable of verifying advanced Java programs [vdBJP01, JMR04, BCHJ04]. As a part of the work on the LOOP tool, a Java Card API specification in JML has been developed [MP01, PvdBJ00].

Yet another system associated with the VerifiCard Project is the Krakatoa tool [MPMU04]. Here, Java programs and their JML specifications are translated into an intermediate, mostly functional, language, then proof obligations are generated, which in turn are proved with the Coq proof assistant [CDT04].

The VerifiCard Project tools that we mention, have been used to perform serious verification of a commercial Java Card applet [JMR04]. The considered property is one of the properties listed in [MM01]. The main result of the paper is that subtle bugs were found in the verified applet. In the last two papers of this thesis we give a more thorough comparison to the tools and results described in [JMR04]. We should also mention here that recently some preliminary work to reason about rip-out properties and Java Card transactions has been initiated in the context of the VerifiCard Project [HP04] (there, ripouts are called card tears). This is the only work that we are aware of that considers Java Card's transaction mechanism. The approach taken in [HP04] differs from ours in certain respects, we describe that in the fourth paper of this thesis.

The JACK (Java Applet Correctness Kit) tool [BRL03] is also associated with the VerifiCard project, but initially it has been developed outside of the project by Gemplus Research Labs. Currently, it is maintained as part of the Everest project at INRIA, Sophia-Antipolis, France.[15] The motivation and goals of JACK are closely related to those of the KeY system – JACK is designed for Java programmers that do not have background in formal methods. It provides intuitive, integrated user interface to developers (like KeY, JACK is integrated into a commercial IDE tool). Specifications for JACK are written in JML. The tool implements fully automated weakest precondition calculus. It can generate proof obligations for the B [Abr96] method's prover, Simplify theorem prover (the prover used by the ESC/Java), or Coq proof assistant.

[14] `http://www.verificard.org`
[15] `http://www-sop.inria.fr/everest/software.php`

Typical security properties related to information flow (confidentiality and integrity of data) are very important in Java Card applet verification. Here, on the source code level, mostly static analysis techniques are used. [GD03] presents a Java Card case study, that involves confidentiality properties for Java Card applets. Two tools used in this case study are ESC/Java, that we have already discussed, and Jif[16] [Mye99] – a compiler for a security-typed programming language that extends Java with support for static information flow control. In [DHS04] it is shown how interactive theorem proving can also be used to establish properties related to information flow.

Finally, [BCC+04] and [BCHJ04] contain a comprehensive overview of JML tools and applications. In the former, further references to tools and techniques targeted at Java verification based on JML can be found, the latter reports on specification and verification of yet another case study (Gemplus electronic purse applet) in the context of the VerifiCard project.

It is clear from the description of the different tools above, that JML is the most commonly used language for specifying behaviour of Java and Java Card programs. The main reason is that JML was designed with Java in mind – it clearly states what the semantics of class invariants and method contracts are in the context of Java program. Also, the exceptions are handled in a clean and uniform way in JML [LBR99]. On the other hand, OCL [WK03], which is the main specification language in the KeY system, has a more open architecture – it has been designed with UML in mind, but not any specific programming language. Thus, the semantics of OCL can be interpreted differently depending on the actual programming language it is used with [HHB02]. In the KeY system the OCL specifications are interpreted in a very similar way to JML semantics. More discussion on the practical differences between JML and OCL in the context of the KeY system can be found in the third paper of this thesis. We should add here, that recently the KeY system also supports JML [Eng05], and that in KeY it is possible to specify the behaviour of the program directly in Java Card DL, the approach presented in the last paper of this thesis.

Apart from the VerifiCard Project, we should mention the following. The Bandera project [CDHR00] uses abstraction of the execution model of Java and the requirements to prove properties about Java programs using model checking. The advantages are full automation of the model checking phase, trace generation for counter models, and treatment of concurrent Java programs. The drawback is the need for abstraction which poses difficulties for programs containing Java arithmetic and other inductive data structures. In [SSB01] a full formalisation of Java and the Java Virtual Machine using Abstract State Machines is presented.

Finally, we should mention again the KeY system's formalisation of Java/ Java Card – Java Card Dynamic Logic [Bec01, BS01]. Dynamic Logic is also used to handle a fragment of the Java language in the KIV system [Ste01]. We have also pointed out that a formalisation of Java in Isabelle [vO01] could be used to verify the correctness of the KeY's Java Card DL calculus rules.

[16] `http://www.cs.cornell.edu/jif/`

7 Future Work

We are encouraged by the results of our work. We are able to formally verify interesting safety and security properties for Java Card applets with full support for the transaction mechanism and strong invariants. Our experiments have been performed on applications of considerable size and we think that the current degree of automation and time performance are already satisfactory. We have also identified some of the Java Card design and development aspects that make the verification easier. However, some of the issues that we discussed in this thesis should be investigated further. An important step to proceed with this work is to try to integrate our ideas into an industrial context. The ultimate goal would be to develop a commercial Java Card application from scratch, following our design process ideas, and verify its correctness. Such an experiment would definitely confirm the usefulness of the design process and design for verification guidelines. A comparative study to regular Java Card development techniques could be performed with respect to the number of discovered bugs, cost and time spent on development, etc. Also, by cooperating with industry and investigating a wider diversity of Java Card applets, the design process would be made more precise. We have only investigated two types of Java Card applications in our research. Studying other applications would definitely add to the design process and guidelines – more Java Card specific design and specification patterns would be identified to form a comprehensive Java Card design pattern library and new ways to formally specify domain specific behaviour of Java Card applets would be investigated, for example, how to efficiently specify and verify interactions between applets through sharable interfaces. Those goals require access to commercial applets, which is difficult to obtain without serious commitment from industry.

Regardless of whether the goal above can be achieved or not, it would be interesting to investigate the applicability of formal development techniques to a wider range of Java applications targeted at small consumer devices. Here, the Java2 Micro Edition platform is an obvious choice for further research. Although being still relatively "small", Java2 ME already poses certain difficulties not found in Java Card – Java2 ME supports graphical user interfaces (for example, in mobile phones) and concurrency.

In the last paper of the thesis we point out that, so far, we are not able to perform global verification of Java Card applets. By global, we mean that parts (single methods) of an applet can be verified, but not the whole applet as such. This is strictly a performance issue. Although we performed our experiments with the KeY system only, the literature reports that the other formal Java Card tools, like LOOP [JP03] or Krakatoa [MPMU04], also suffer from such performance problems, in particular, the underlying theorem provers (PVS [ORSvH95], Coq [CDT04]) seem to be the performance bottlenecks. Here, the research direction would be to push the tools towards such global verification and try to eliminate the performance deficiencies. Further modularisation of proofs is one candidate for investigation. Based on our experience, we also believe that, in the long run, full automation in verification of

certain classes of Java Card applets can be achieved, without sacrificing expressiveness of properties. To reach this goal, further ways to improve automation should be researched, for example, automated rules to handle simple loops.

Finally, we have mentioned that the idea of introducing the throughout operator could be extended to other temporal operators. As an initial step it would be desirable to investigate what kind of useful properties one could specify and verify using such temporal operators, in the context of Java Card or Java2 ME environments.

Bibliography

[ABB+04] 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.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[BCC+04] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Springer, December 2004. Online First issue, to appear in print.

[BCHJ04] Cees-Bart Breunesse, Néstor Cataño, Marieke Huisman, and Bart Jacobs. Formal methods for smart cards: An experience report. In *Science of Computer Programming*, ENTCS. Elsevier, 2004. To appear.

[Bec01] 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.

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[BGH+04] 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.

[BH04] Richard Bubel and Reiner Hähnle. Integration of informal and formal development of object-oriented safety-critical software – A

case study with the KeY system. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.

[BHSS00] Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, pages 389–404. Springer, September 2000.

[BM03] 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 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.

[Boy03] 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.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In *Proceedings, Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[BRS+00] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[BS01] 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.

[BS03] 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.

[BS04] 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.

[CDHR00] 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.

[CDT04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.0*, January 2004. Available from `http://coq.inria.fr`.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, 2000.

[CL02a] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *The International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[CL02b] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255. Springer, June 2002.

[DHS04] 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.

[Eng05] Christian Engel. A translation from JML to JavaDL. Minor thesis, Karlsruhe University, Computer Science Department, Karlsruhe, Germany, 2005.

[Ens01] Oliver Enseling. iContract: Design by contract in Java, 2001. Available from `http://www.javaworld.com/archives/index-jw-02-2001.html`.

[FLL+02] 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.

[GD03] Pablo Giambiagi and Mads Dam. Verification of confidentiality properties for Java Card applets. Manuscript. Report on additional work in WP4.3. Available from `http://www.cs.ru.nl/VerifiCard/files/verConf-T4.3-item4.pdf`, 2003.

[GHL04] Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-based simplification of OCL constraints. In Octavian Patrascoiu et al.,

editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 84–98, 2004.

[Gie01] Martin Giese. Incremental closure of free variable tableaux. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning (IJCAR), Siena, Italy*, volume 2083 of *LNCS*, pages 545–560. Springer-Verlag, 2001.

[Hab00] Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000. Available at `http://i12www.ira.uka.de/~key/doc/2000/stsr.ps.gz`.

[HHB02] Rolf Hennicker, Heinrich Hußmann, and Michel Bidoit. On the precise meaning of OCL constraints. In T. Clark and J. Warmer, editors, *Advances in Object Modelling with the OCL*, volume 2263 of *LNCS*, pages 69–84. Springer, 2002.

[HJ00] Marieke Huisman and Bart Jacobs. Java program verification via a Hoare logic with abrupt termination. In *Proceedings, Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *LNCS*, pages 284–303. Springer, 2000.

[HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HLS+96] Dieter Hutter, Bruno Langenstein, Claus Sengler, Jörg H. Siekmann, and Werner Stephan. Deduction in the Verification Support Environment (VSE). In Marie-Claude Gaudel and James Woodcock, editors, *Proceedings, Formal Methods Europe: Industrial Benefits Advances in Formal Methods*. Springer, 1996.

[HM01] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java virtual machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.

[HM05] Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[HP04] Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in Java Card. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.

[HW03] Reiner Hähnle and Angela Wallenburg. Using a software testing technique to improve theorem proving. In Alex Petrenko and Andreas Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES), Montréal, Canada*, volume 2931 of *LNCS*, pages 30–41. Springer, 2003.

[JMP01] Bart Jacobs, Hans Meijer, and Erik Poll. VerifiCard: A european project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001. Available from `http://www.cs.kun.nl/~bart/PAPERS/nvti01.ps.Z`.

[JMR04] 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.

[Joh04] Kristofer Johannisson. Disambiguating implicit constructions in OCL. In Octavian Patrascoiu et al., editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 30–44, 2004.

[JP01] Bart Jacobs and Erik Poll. A logic for the Java modelling language. In Heinrich Hußmann, editor, *4th Fundamental Approaches to Software Engineering, Genova, Italy*, volume 2029 of *LNCS*, pages 284–299. Springer-Verlag, April 2001.

[JP03] 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.

[Kra98] Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[KT90] Dexter Kozen and Jerzy Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.

[Lar03] Daniel Larsson. OCL specifications for the Java Card API. Master's thesis, Chalmers University of Technology, Department of Computing Science, Göteborg, Sweden, 2003.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*. Kluwer Academic Publishers, 1999.

[LM04] Daniel Larsson and Wojciech Mostowski. Specifying Java Card API in OCL. In Peter H. Schmitt, editor, *OCL 2.0 Workshop at UML 2003*, volume 102C of *ENTCS*, pages 3–19. Elsevier, November 2004.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note #1997-007, Digital Systems Research Center, Palo Alto, USA, January 1997. Available from `ftp://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-007.ps.gz`.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.

[MM01] 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.

[MMPH00] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The Jive system – Implementation description. Available from `http://softech.informatik.uni-kl.de/downloads/publications/jive.pdf`, 2000.

[Mos02] 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`.

[Mos04] Wojciech 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.

[Mos05] Wojciech Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In Maura Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference 2005, Edinburgh, Scotland*, LNCS. Springer, April 2005. To appear.

[MP01] Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*, volume 2140 of *LNCS*, pages 165–178. Springer, September 2001.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors,

Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000, Berlin, Germany, volume 1785 of *LNCS*, pages 63–77. Springer, April 2000.

[MPMU04] 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`.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[Mye99] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, January 1999.

[Obj03] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[Pau94] Larry C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[PHM99] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential JAVA. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

[PvdBJ00] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JAVA CARD API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Academic Publishers, 2000.

[SSB01] Robert F. Stärk, Joachim Schmid, and Egon Börger. *JAVA and the JAVA Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.

[Ste01] Kurt Stenzel. Verification of JAVA CARD programs. Technical Report 2001–5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available from `http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/`.

[Sun03] Sun Microsystems, Inc., Santa Clara/CA, USA. *JAVA CARD 2.2.1 Platform Specification*, October 2003.

[vdBHJP00] Joachim van den Berg, Marieke Huisman, Bart Jacobs, and Erik Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 of *LNCS*, pages 1–21. Springer-Verlag, 2000.

[vdBJP01] Joachim van den Berg, Bart Jacobs, and Erik Poll. Formal Specification and Verification of Java Card's Application Identifier Class. In I. Attali and Th. Jensen, editors, *Proceedings of the Java Card 2000 Workshop*, volume 2041 of *LNCS*, pages 137–150. Springer-Verlag, 2001.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, January 2001.

[Wal04] Angela Wallenburg. Induction rules for proving correctness of imperative programs. Licentiate Thesis 1L, Chalmers University of Technology and Göteborg University, Department of Computer Science and Engineering, Göteborg, Sweden, December 2004.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.

Paper I — Rigorous Development of Java Card Applications

Fourth Workshop on Rigorous Object-Oriented Methods, London, U.K., March 2002

Rigorous Development of Java Card Applications

Wojciech Mostowski

Abstract

We present an approach to rigorous, tool supported design and development of Java Card applications. We employ the Unified Modelling Language (UML) and formal methods for object oriented software development in our approach. Our goal is to make Java Card applications robust "by design", to make the development process independent of the Java Card platform used and to enable applications to be verified formally by the KeY system. First we analyse the current situation of Java Card application development, then we present a real life Java Card case study and describe the problems we found that should be addressed by rigorous development. Finally we propose some solutions to selected problems by using UML specifications, software design patterns, formal specifications and a modern CASE tool support.

1 Introduction

In this article we present an approach to rigorous, tool supported design and development of Java Card applications. Our goal is to make Java Card applications robust "by design", to make the development process independent of the Java Card platform used and to enable applications to be formally verified by the KeY system [ABB+04]. First we analyse the current situation of Java Card application development, then we present a real life Java Card case study (`pam_iButton` [Bol]) and describe the problems we found that should be addressed by rigorous development and formal verification. We propose solutions to selected problems by presenting a framework that incorporates the use of UML [Obj03] specifications, software idioms and design patterns, a modern CASE tool support, as well as formal specification and verification to provide a systematic, rigorous development process for Java Card applications.

1.1 Java Card

We start with a short introduction to Java Card technology [Che00]. Java Card provides a means of programming smart cards with (a subset of) the Java programming language. Today's smart cards are small computers, providing 8, 16 or 32 bit CPU with clock speeds ranging from 5 up to 40 MHz, ROM memory between 32 and 64 KB, EEPROM memory (writable, persistent) between 16

and 32 KB and RAM memory (writable, non-persistent) between 1 and 4 KB. Smart cards communicate with the rest of the world through application protocol data units (APDUs, ISO 7816–4 standard). The communication is done in master-slave mode – it is 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 (possibly with empty contents). In the case of Java powered smart cards (Java Cards) besides the operating system the card's ROM contains a Java Card virtual machine which implements a subset of the Java programming language and allows running Java Card applets on the card. The following are the features not supported by the Java Card language compared to full Java: large primitive data types (`int`, `long`, `double`, `float`), characters and strings, multidimensional arrays, dynamic class loading, threads and garbage collection. Some of the actual Java Card devices go beyond those limitations and support, for example, the `int` data type 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. The card also contains the standard Java Card API, which provides support for handling APDUs, Application IDentifiers (AIDs), Java Card specific system routines, PIN codes, etc. A proper Java Card applet should implement the `install` method responsible for the initialisation of the applet (usually it just calls the applet constructor) and a `process` method for handling APDU communication with the host. There can be more than one applet existing on a single Java Card, but there can be only one active at a time.

1.2 Analysis of the Current Situation

Java Card technology is relatively young and still developing and so are design and development techniques for Java Card applications. Although the Java Card language is based on full Java, the nature of the Java Card environment (for example, constrained memory, no garbage collection) makes Java Card programming quite different from normal Java programming. Powerful development and modelling tools for Java are not Java Card "aware". Such a Java tool can become helpful provided it can be customised to Java Card needs. This however is not the common approach being taken. Instead, each Java Card vendor provides its own development environment and proposes its own Java Card specific solutions. The provided tools try to ease the actual process of writing Java Card programs, installing them to the card and testing, but they hardly ever provide the support for the design of Java Card applications in a more abstract sense. Our experience is based on using the Java-powered iButtons,[1] which we use in our research, and the development environment, iB-IDE, provided for this platform, but most of the following statements apply to other environments too. The iB-IDE tool provides the following functionality: automatic creation of the skeleton code for both the card (iButton) application

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

and Open Card Framework [Ope] compliant Java host application with convenience methods for dispatching user defined command APDUs and converting data types, debugging tools with the possibility of running the card applet in an emulated environment, and finally a very handy APDU sender which is used to communicate with the card applets without a host application and to provide some card administration services – downloading applets to the card, erasing card's memory, etc. The tool however does not provide any kind of modelling or design support for building Java Card applications, nor does it provide any support for formal specification and verification. Moreover, the Java Card virtual machine in iButton devices implements garbage collection and the iB-IDE skeleton code and example applets make heavy use of that fact. Thus the solutions provided by iB-IDE are not (easily) portable to other Java Card platforms. In contrast to this, SUN's Java Card reference development kit[2] provides very nice examples which take into account common Java Card limitations and proposes a very elegant way of writing Java Card applets, but the development kit itself does not contain any user friendly tools to create the applications, the only support available are the command line tools for compiling and running the applets in a simulated/emulated environment.

The next issue is the need for the use of formal methods in Java Card application development. There are two reasons for this. First of all, smart card application are usually security critical, secondly, in contrast to normal computer software, making updates on the cards distributed in large amounts is not possible, thus correctness of the card application should be done by best means possible. At the same time Java Card applications seem to be suitable for formal verification because they are small in size and the Java Card programming language lacks some of the complications of the full Java language that makes formal verification difficult (like threads, graphical user interfaces, complex data types). Finally a controlled software development process in general (like the one we want to propose, or an industrial one, like Nokia OK process) will benefit from adding the formal methods support to it.

Taking all this into account it is clear that the development of Java Card applications needs to be uniform, platform independent, and should be done in a controlled, systematic, well defined, rigorous way giving the possibility to formally verify the application's properties. Our further experience [HM05] shows that properly designed and developed Java Card application substantially eases the formal verification effort.

1.3 Related Work

Most of the efforts related to Java Card program development are concentrated on the verification issues. Those can be generally divided into low level (byte code) and high level (source code) verification. An overview of work done on the byte code level can be found in [Boy03]. For the source code verification of Java Card application we only name the most important tools and projects. The

[2] `http://java.sun.com/products/javacard/`

LOOP tool [JP03] employs the PVS theorem prover to prove properties about JAVA CARD programs annotated with JML (JAVA Modelling Language) specifications. JML is also used as a specification language in ESC/JAVA2 [FLL+02], KRAKATOA [MPMU04], and JIVE [MMPH00] tools. ESC/JAVA2 provides full automation, trading off the completeness and soundness of the static checking used to verify JAVA CARD programs. The KRAKATOA tool, which is part of the VerifiCard project,[3] uses the COQ theorem prover as the basis for verification. The JIVE system employs an extended Hoare style calculus implemented in Isabelle and PVS theorem provers through a dedicated graphical user interface.

Our work has been done in the context of the KeY Project [4] [ABB+04]. One of its main objectives is to integrate formal methods with object oriented software design to provide user friendly formal verification environment for JAVA CARD, thus, the KeY tool seems to be most suited for our purposes. The design and specification languages used in KeY are respectively UML [Obj03] (Unified Modelling Language) and OCL [WK03] (Object Constraint Language), which is part of the UML standard. During the course of the paper we show how we use the KeY tool to support formal development of JAVA CARD applications.

Apart from the formal approaches to JAVA CARD development the following should be mentioned. The Open Card organisation [Ope] bundles efforts to create a common and unified programming framework for writing host/terminal applications for JAVA CARD devices coming from different manufacturers (Open Card Framework). JAVA CARD applications are quite often security critical, [Jür01] shows how UML can be used to express security requirements during system development.

1.4 Our Approach

In our approach to the development of JAVA CARD applications we use UML modelling techniques, software patterns and incorporate formal methods in an incremental way. By incremental we mean that the use of formal methods should be optional and it should be up to the developer (who might be unfamiliar with formal methods) at which level of detail formal methods are used, a view stressed in [ABB+04, BML01]. To enable and ease the usage of formal methods we try to provide means of creating certain kinds of formal specifications semi automatically in two ways. The first by applying software and specification patterns solving some common problem to the application design [GHL04]. Such patterns usually need to be provided with parameters during instantiation to create a proper specification, but giving the parameters is the only job that is required from the developer. The second way is to create a specification out of certain kinds of UML diagrams (possibly taking some parameters from the user). Both enable creating partial specifications without detailed knowledge about the formal specification language. The created specifications are well formed by design and ready to be formally verified.

[3] `http://www.verificard.org`
[4] `http://www.key-project.org`

Having all this support we can make Java Card applications robust and secure by design and easier for verification. To achieve our goals we need support from a modern, fully customisable UML CASE tool as well as a suitable formal verification system. As we already mentioned, we use the KeY system [ABB+04] as a basis in our framework. The KeY system extends a commercial UML CASE tool with formal verification modules in a seamless way. The CASE tool currently used by the KeY system is Together Control Center from Borland.[5] It provides state of the art support for UML and is fully extensible through its Java open API. Therefore, we can use powerful UML support as well as formal verification tools in one framework. One of the KeY extensions to the CASE tool provides a library of common design and specification patterns to support creation of OCL specifications [GHL04]. Yet another extension we introduced to the CASE tool supports low level Java Card development tasks, like compiling, installing, or testing applets, either in a simulated environment or on real Java Card devices [Mos]. Such framework allows us to make our solutions to Java Card design issues independent of the actual Java Card platform and vendor specific development environment. At the same time we obtain generic, powerful UML support and direct access to the state of the art verification system.

In this work we limit ourselves to the card applications (Java Card applets), that is, we do not consider the problems of developing the host application. The main reason is that host applications are mostly regular Java programs and usually are part of a bigger project, to which existing development techniques can be applied, for example, the Unified Process [JBR99]. Moreover, suitable efforts to support uniform development of host applications are carried out, for example the Open Card Framework [Ope]. The applets themselves seem to be too small applications to be subject to "big" process like UP, we believe that developing a Java Card applet should be seen as a small subprocess, which needs specific approach.

In this paper we present a motivating Java Card case study (Section 2) based on which we identify Java Card specific design issues and problems we want to tackle (Section 3). In Section 4 we walk through and reengineer our example to present our framework. In Section 5 we discuss the solutions in the framework that address the problems we identified. Finally, Section 6 summarises the paper.

2 Case Study: pam_iButton

It is time to present our case study, upon which we build up some of the common design requirements for Java Card applications. The `pam_iButton` package was written by Dierk Bolten and is available free of charge [Bol]. The package allows a Linux user to authenticate himself to the system by inserting an iButton device into the reader instead of giving the password. A Java-powered iButton is a Java Card device implementing Java Card API version 2.0 (which differs

[5] http://www.borland.com/together/

substantially from the current Java Card API 2.2) with `int` data type support and garbage collection. The most recent Java-powered iButton has an 8 bit processor, cryptographic (RSA and SHA1) coprocessor and 130 KB of non-volatile RAM memory. The `pam_iButton` package consists of a PAM (Pluggable Authentication Module) Linux system library which is responsible for authentication on the system side, a setup utility to configure the necessary system files and administrate the iButton and the Java Card applet (`Safe_Applet`) which performs the actual authentication on the iButton device.

The following is an example `pam_iButton` usage scenario. First a Linux user account needs to be setup to be able to use the iButton authentication. The user is assigned a unique ID number and a pair of private and public RSA keys is generated on the iButton and stored together with the user's ID in the iButton's memory (many different users can be registered on one iButton). The public key is then retrieved by the system from the iButton and stored in the system configuration file together with the user's ID number. The iButton is ready to be used for authentication. When the user wants to be authenticated he types in his login name. The system looks up his ID number and encrypts a random message with the user's public key. The encrypted message and the user's ID number are sent to the iButton applet. The applet checks if the user is registered and if so, it decrypts the message with the private key, computes the SHA1 hash value from the decrypted message and sends it back to the system. The system compares the received SHA1 value with its own and if they match the user is authenticated successfully.

The following is the list of the most important and interesting command APDUs that `Safe_Applet` accepts:

- Store data – stores temporary data for a subsequent command.
- Authenticate user – given the user's ID performs the challenge-response authentication described earlier. In response sends back the SHA1 code of the message. The encrypted message has to be sent beforehand with the 'store data' command.
- Set PIN code (PIN code protected) – sets a new PIN code for PIN protected commands.
- Generate key pair (PIN code protected) – given the user's ID generates an RSA key pair (the generation is done on the card) and stores it together with the user's ID in the applet's memory. In response sends back the public part of the key.
- Get public key – given the user's ID sends back the public part of the key.
- Delete key pair (PIN code protected) – given the user's ID removes this user's key pair entry from applet's memory.
- Get key information – sends back the ID numbers of users registered in the applet.

Any command (except for the first and the last) sent to the applet can cause an error condition in which case instead of the expected answer the error code (status word) is sent back to the host indicating what the error was caused by. Internally in the Java Card applet this is done by throwing an appropriate exception (`ISOException`).

3 Design Issues for Java Card Applications

In the following section we will describe what issues came up while we were studying the example and we will try to list some common requirements that a Java Card application should satisfy.

One of the first questions that came to mind were the following. Who is the owner of the applet PIN code, Linux system administrator or the user? Who is the person to setup iButton for authentication, the system administrator, the user, both? What are the applet deployment steps, who is responsible for installing the applet to iButton, when is iButton ready to be passed to the user for regular usage (that is, when does the applet get personalised)? Should it be possible for one iButton applet to be used on two different Linux systems? Answers to some of the questions imply answers to some of the other questions, for example, if a single applet can be used on many different systems then it certainly should be the user owning the applet's PIN code and it should be the user that sets up the system configuration, probably through some administrator privileged system tool, which itself needs to be very carefully designed.

One way or the other, the answers to the posed questions are not provided by the design of the applet, at least not explicitly, and since this kind of application is security critical, things like those mentioned above need to be well defined and carefully thought through.

Secondly, we took a closer look at the protocol that is used to exchange information between the host application and the iButton applet and we discovered the following. There is no order imposed on command sequences: in one possible type error attack scenario first the 'store temporary data' command is sent to the applet with the intention for this data to be used with a given subsequent command call (say 'authenticate user'), but then a different command is sent which also relies on 'store temporary data', in which case the latter gets wrong data (for example, intended for 'authenticate user'), which may cause corruption of the applet data. Apart from 'authenticate user' there are other commands that rely on 'store temporary data'. In case of this particular applet we did not find a sequence of command calls that could put the applet in an unrecoverable state, but it is definitely possible to corrupt the applet state with wrong data, causing some (recoverable) malfunctioning. Connected to this problem as well as to the next one, is the fact that there are no integrity checks on the data being sent along. Some of the commands may require input that does not fit into a single APDU, so there are multiple APDUs being sent. However, there is no control whether the proper number of APDUs in a proper order is sent (in particular this may cause the applet to run out of memory). The last thing we

found strange about the protocol is that the PIN code is sent along with each command that requires PIN code authentication. Generally there is nothing wrong with it, but it produces overhead and it is different from the commonly used solution of establishing the PIN code once per command exchange (card) session.

Another thing which we found problematic (and it applies to iButton applets in general, not only the one presented) is the unconstrained memory usage. The iButton applets make heavy use of garbage collection and do a lot of dynamic memory allocation. Not watching for memory usage makes life much easier for the developer, but it also makes the applet much less robust – due to lack of memory the applet may refuse to function properly at any point of execution. Such an approach to Java Card programming also makes the applications not portable to other Java Card devices that do not support garbage collection.

Extensive testing of `Safe_Applet` revealed one more problem. If the user rips out the iButton from the reader during authentication, the applet is not functioning properly any more during subsequent authentication sessions. The main reason for this is the loose command exchange protocol, which does not take the possibility of rip out into account. The design of a Java Card application should take such possibilities under consideration and try to make the applets as robust and rip-out proof as possible. Our further research [HM05] shows that formal verification is the right technique to ensure that applets are rip-out safe.

The last thing we want to point out is that `Safe_Applet` allows two different key pairs registered with the same user ID number. While this is the author's deliberate design decision, we think the applet should forbid to make double entries of this kind, instead of making the user responsible for controlling the state of the key pair entries.

Some of the problems we mentioned may seem not to be an issue for such a small application as `Safe_Applet`, but we want to make the Java Card design and development process scalable, and for bigger applications the problems raised here definitely become serious issues which need to be addressed. Based on what we have already described we now list some of the common design issues in Java Card development we will try to face and give some support to in the next section:

- the applet has to be robust in the sense that it should be protected against malicious host application, tampering with and against ripping out the card from the reader,

- the applet deployment steps and life cycle should be well defined and controlled by the applet itself disabling improper applet usage,

- the message exchange protocol should be well defined, constrained and controlled by the applet to disable illegal command invocation sequences (this also includes proper support for the commands requiring data to be sent in multiple APDUs),

- the applets should be very careful about the memory (to say the least), here we would like to take the safest approach of allocating all the memory an applet may ever want to use during applet installation time [Che00].

To end this section we want to stress that in our work we do not want to impose any ways of taking design decisions for Java Card applications (for example, what actual deployment steps the applet should have, whether a certain command should be PIN code protected, etc.), we only want to support the design process and provide the developer with means and tools to make those design decisions and control the development process in a rigorous way. The design decisions we present in the next section are only examples among many possible, the design decisions in real-life Java Card world should be done by a domain expert.

4 Developing Java Card Applications

We now present how one can go about designing and developing a Java Card application by going through the case study again and reengineering it in a well defined way. The large parts of the example that we present should give the reader the complete overview of the development process. In the next Section we will discuss the crucial features of our framework that make the developed application robust and free of the problems we identified earlier.

4.1 Applet Life States

First we define the life states of the applet (deployment steps). These are the distinguished states that the applet will go through during its life time. For our application we can limit ourselves to the following:

- applet is selectable, this is the state of the applet just after installing (downloading) it to the card, but before setting some data in the applet that is necessary for proper functioning of the applet,
- applet is personalised, this is the state after setting the data on the applet. This is also the applet's "normal operation" state,
- applet is locked, this is the state after something goes wrong during normal applet usage, for example, the user entered the wrong PIN code a number of times and the applet access is blocked temporarily,
- applet is dead, this is the state after an unrecoverable misusage of the applet, in our case when the user enters a wrong master PIN code, which can only be presented for verification once and is only allowed to be presented in locked state.

An applet goes only once through the selectable state during its life and also it can never leave the dead state after entering it. It can however move between

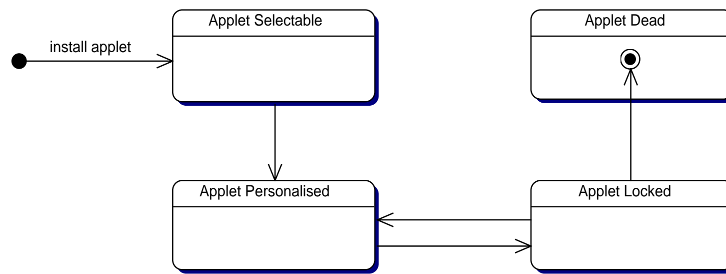


Figure 1. `Safe_Applet` life states

personalised and locked states many times during its life time. We will show later what the exact conditions that cause an applet's life state change are. One last thing that we will require from the applet is that it enforces the card terminal session to be restarted after the applet has moved from one life state to another. Figure 1 shows a UML state diagram presenting the life states idea we have just described.

4.2 Applet Commands

We are now at the point where we can start defining the command APDUs that the applet should support. To avoid some of the problems described earlier (for example, unnecessary PIN code sending, see verifyUserPIN below), the commands have been redesigned, so they differ slightly from the ones described in Section 2. For each of the commands we give it a name, we say if it can be invoked in a given applet life state and if it is a user or master PIN code protected command (for each state separately). Table 1 shows the list of commands we are interested in. Without specifying formally what are a given command's parameters and responses we now give an informal description of the intended meaning of the commands:

authenticateUser This command is used to authenticate a given user through a challenge-response protocol. A single person owns one Java Card device with a single `Safe_Applet`, however there can be more than one system user registered in the applet. Hence, the command has to specify, by giving a user ID, which user is to be authenticated.

updateUserPIN This command changes the user's PIN code to a new one. Depending in which life state the applet is, different security measures are taken to protect the command. For example, since the personalisation step should be taken in the issuer's trusted area it is not necessary to require PIN authentication for updating the user's PIN in selectable state.

setMasterPIN This command sets the master PIN for the applet. It's the only command required to make the applet personalised, hence after successful

Name/State	Selectable	Personalised	Locked	Dead
authenticateUser	No	Yes	No	No
updateUserPIN	Yes	Yes (PIN)	Yes (Master PIN)	No
setMasterPIN	Yes	No	No	No
verifyUserPIN	No	Yes	No	No
verifyMasterPIN	No	No	Yes	No
generateKeyPair	No	Yes (PIN)	No	No
deleteKeyPair	No	Yes (PIN)	No	No
getPublicKey	No	Yes	No	No
disableUser	No	Yes (PIN)	No	No
enableUser	No	Yes (PIN)	No	No
getKeysInfo	No	Yes	No	No

Table 1. Possible `Safe_Applet` commands

invocation it should move the applet from the state selectable to personalised.

verifyUserPIN This command performs the verification of the user's PIN which after successful verification stays validated until the end of the terminal session. All the commands that are PIN code protected can check the PIN code validity flag.

verifyMasterPIN Same as the previous one, just for the master PIN. This command can only be invoked in the locked state to enable special behaviour to unlock the applet. Usually the master PIN is only allowed to be presented once, after an unsuccessful try the applet becomes dead.

generateKeyPair This command generates a pair of keys (public and private) for a given user's ID and stores this information in the applet's memory for future use.

deleteKeyPair This command removes the information about the keys for a given user's ID from applet's memory.

getPublicKey This command retrieves the public part of a key for a given user's ID.

disableUser, enableUser These commands are used to disable and enable the authentication of a given user specified by a user's ID. The user may wish to block the usage of `Safe_Applet` when he has to pass the Java Card device (iButton) to somebody else (for example, to download some other applets).

getKeysInfo This command should inform the owner of the applet about all user IDs registered in it (for administrative purposes).

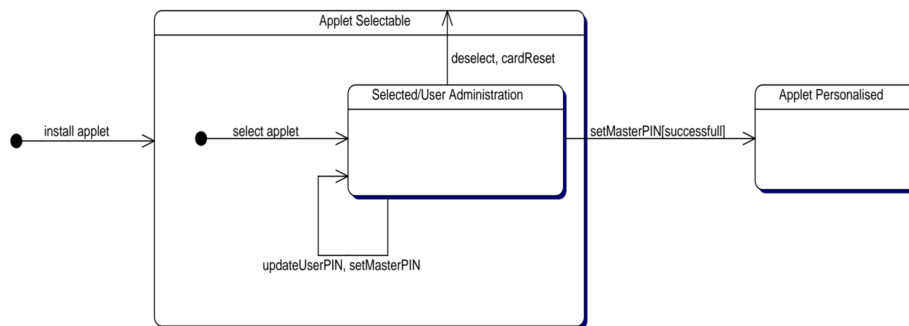


Figure 2. Command states in the selectable life state

The commands that can be invoked during the operational mode of the applet (personalised) fall into certain categories, which in turn define possible sequences of command invocations. For example authenticateUser is the only application command that is going to be used on a daily basis, while updateUserPIN is a user administration command, which is invoked rarely (if ever) and should not be mixed with application mode commands. Commands like generateKeyPair or getPublicKey fall into system administration category.

4.3 Command Invocation Protocol

The information we gathered so far is sufficient to define the protocol that `Safe_Applet` should follow. We do this by presenting further state charts, one inside each state representing a single applet life state. We will call the new substates the command states. In our application we distinguish four different command states. The initial one is the selected state. This is after the applet is selected by the Java Card run time environment (this is usually by host application). Then, depending on the commands invoked, the applet can be in one of the three command states: application, user administration or system administration.

Both in selectable and locked life states the command states selected and user administration are in some sense equivalent and we put them together as one state. At this stage we also define precisely under what conditions the applet changes its life state.

Let us start with the selectable life state. Figure 2 shows the corresponding state chart diagram. The black dot represents the state in which the applet is not active and needs to be selected. When the applet gets deselected by the Java Card run-time environment or a card reset event occurs the applet has to be selected again. There is only one command state inside the life state selectable and only two commands possible. The invocation of updateUserPIN is optional during the personalisation process – the applet issuer may wish to release the applet without user PIN code set. Once setMasterPIN is invoked

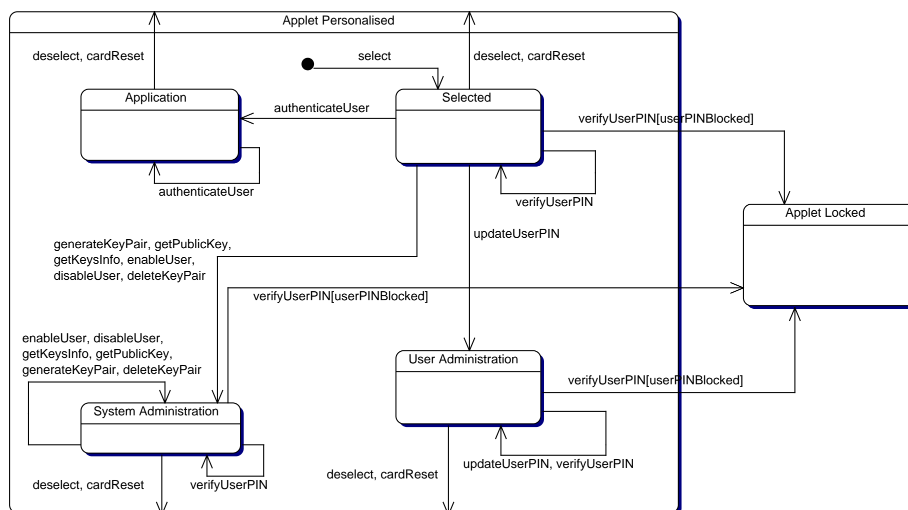


Figure 3. Command states in the personalised life state

successfully (no error occurs and the input data for setting the master PIN is not corrupted) the applet changes its life state to personalised and never goes back to selectable. The card/terminal session has to be restarted after a life state change, which means that no further commands can be invoked after a successful setMasterPIN until the applet is selected again.

Figure 3 shows the details of the personalised life state. This is the applet's main operational state in which most of the application and administration commands are enabled. As before, after selection the applet is in selected command state. Once a command belonging to one of the three classes (application, system administration, user administration) is invoked the command state is changed accordingly and the applet stays in this state until the end of the session. To enter a different command mode the session has to be restarted. The verifyUserPIN command is treated in a special way – since the PIN code is required by the commands both in system and user administration modes invoking verifyUserPIN does not change the command state of the applet. However if the PIN verification fails the maximum allowed number of times (userPINBlocked) the applet's life state is changed to state locked where special rules apply for unblocking the PIN code. The only application mode command is authenticateUser, the only user administration command is updateUserPIN and in system administration mode we have the following commands enabled: generateKeyPair, deleteKeyPair, getPublicKey, getKeysInfo, disableUser and enableUser.

Finally, we describe the command protocol for the applet life state locked (Figure 4). As in the case of life state selectable there are two equivalent command states – selected and user administration. The only two commands that are allowed here are verifyMasterPIN and updateUserPIN. After successful master

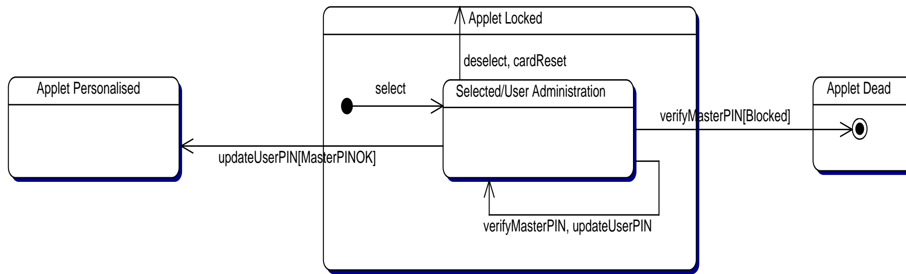


Figure 4. Command states in the locked life state

Name	Input parameters	Length	Integrity	APDUs
authenticateUser	User ID, the challenge	1+256	No	Many
updateUserPIN	New PIN data	8	Yes	1
setMasterPIN	PIN data	16	Yes	1
verifyUserPIN	PIN data	8	Yes	1
verifyMasterPIN	PIN data	16	Yes	1
generateKeyPair	User ID	1	No	1
deleteKeyPair	User ID	1	No	1
getPublicKey	User ID	1	No	1
disableUser	User ID	1	No	1
enableUser	User ID	1	No	1
getKeysInfo	None	0	No	1

Table 2. Command parameters

PIN verification (MasterPINOK) the updateUserPIN command sets the new user PIN code and unblocks it moving the applet back to personalised life state. In case the master PIN verification failed the applet life state changes to dead from which there is no return – the applet becomes unoperational.

All the command invocation sequences that are not defined by the diagrams are forbidden – in case of any attempt to violate the defined protocol the applet should end the communication immediately by throwing a suitable exception.

Note that we already gave a lot of semi formal information about the applet we are building without writing or presenting a single line of Java Card code. This shows how proper documenting can be useful to understand the intended behaviour of the applet.

4.4 Command Processing

It is time to focus on the actual command processing. For each of the commands we listed we now define which parameters a given command takes, whether there should be extra integrity checks on the delivered data, if the command is allowed

Name	Response data	Length	Integrity
authenticateUser	SHA1 code	20	No
updateUserPIN	None	0	No
setMasterPIN	None	0	No
verifyUserPIN	None	0	No
verifyMasterPIN	None	0	No
generateKeyPair	None	0	No
deleteKeyPair	None	0	No
getPublicKey	User's public key	131	Yes
disableUser	None	0	No
enableUser	None	0	No
getKeysInfo	User IDs	0...Max Users	No

Table 3. Command responses

to spread across multiple APDUs and what is the response data (again with the indication of whether extra integrity checks are required). Tables 2 and 3 show the complete list.

Taking into account everything we have said so far about commands we now show how the actual dispatching of the commands can be done inside the Java Card applet based on the examples of updateUserPIN, getPublicKey and authenticateUser. Let us start with updateUserPIN. Recall that this command had a conditional PIN check depending on the current applet's life state. It also expects 8 bytes of input data and there is a required integrity check on the data. There is no response data, just a status word is sent back to the host indicating the (un)successful invocation of the command. The command should also follow the protocol we defined. Here is the code:

```
/** @param apdu the incoming apdu packet to dispatch */
public void dispatchUpdateUserPin(APDU apdu) {
  updateCommandState(UPDATE_USER_PIN);
  switch (curr_applet_state) {
    case AS_SELECTABLE:   break;
    case AS_PERSONALISED: checkPIN(); break;
    case AS_LOCKED:       checkMasterPIN(); break;
  }
  readInput(apdu, (short)28); // modifies temp
  verifyInput((short)8);
  userPIN.update(temp, (short)0, (byte)8);
  if (curr_applet_state == AS_LOCKED) {
    setAppletState(UPDATE_USER_PIN, AS_PERSONALISED);
  }
}
```

The call to `updateCommandState` makes sure that the command is invoked according to the protocol. The `updateCommandState` implements a state machine

that follows the diagrams shown. The switch statement performs the conditional PIN check (the AS prefix stands for applet state). Then the input is read, which has to be 8 bytes long plus 20 bytes for the SHA1 code for data integrity verification. After the data is retrieved from the APDU packet it is stored in the temp array, which is allocated once during applet installation and is sufficiently big to serve all command dispatching methods, thus keeping memory consumption fixed. The method verifyInput performs the actual verification of the data stored in the temp array. Then the actual user PIN update happens. If the applet happens to be in locked life state then it switches back to personalised state after successful update (setAppletState).

Let us take a look at getPublicKey now. This command does not require any PIN checks, expects 1 byte of input data without integrity verification and sends back 131 bytes of response plus additional 20 bytes of SHA1 code for integrity verification on the host side. We skip the actual key retrieval code as it is not relevant at this point. Here is the code:

```
public void dispatchGetPublicKey(APDU apdu) {
    updateCommandState(GET_PUBLIC_KEY);
    readInput(apdu, (short)1);
    // retrieve the key, prepare the response data in temp
    integrifyOutput((short)131);
    sendResponse(apdu, (short)151);
}
```

The sendResponse method simply sends the data prepared in the temp array back to the host.

Now let us see how the code for authenticateUser command is constructed. This command is the only one that is allowed to be sent in parts in multiple APDUs. There is no PIN check nor input data integrity verification required. The response is 20 bytes of SHA1 code calculated from the received message. Here is the code:

```
/** @param apdu the incoming apdu packet to dispatch */
public void dispatchAuthUser(APDU apdu) {
  updateCommandState(AUTH_USER);
  readBigInput(apdu, (short)257);
  if (multiple_package == (byte)0) { // everything read
    // process bigtemp, prepare the response in temp
    sendResponse(apdu, (short)20);
  }
}
```

Methods readBigInput and updateCommandState make sure that the data parts contained in different APDUs are sent in proper order and are not interleaved by any other commands. This is done by using global applet variables and requiring the multiple APDUs sent over to the applet to be properly marked as we will show shortly.

Now we give some more details about the auxiliary methods that are used by dispatch methods. The readInput method reads the input from the incoming

APDU into the `temp` array in a standard way reporting any possible data length mismatches by throwing an appropriate exception, which in turn causes a status word indicating an error condition to be sent back to the host:

```
/**
  * @param apdu the incoming apdu to read data from
  * @param expectedLength the expected data length to read
  */
public void readInput(APDU apdu, short expectedLength) {
  byte buffer[] = apdu.getBuffer();
  short apduDataOffset = 0;
  short dataLength =
    (short)(buffer[ISO7816.OFFSET_LC] & (byte)0xFF);
  if (dataLength != expectedLength) {
    ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
  }
  short bytesRead = apdu.setIncomingAndReceive();
  while (bytesRead > 0) {
    if ((short)(bytesRead + apduDataOffset) > expectedLength) {
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    }
    Util.arrayCopyNonAtomic(buffer, ISO7816.OFFSET_CDATA,
                            temp, apduDataOffset, bytesRead);
    apduDataOffset += bytesRead;
    bytesRead = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
  }
}
```

Methods `setAppletState`, `updateCommandState`, and `readBigInput` are more interesting. The first one is responsible for setting and changing the applet's life state. It is the calling method's responsibility to ensure that a proper condition for changing this state is satisfied (for example, that master PIN is verified when updateUserPIN changes the state from locked to personalised). The method manipulates the global applet variable called `curr_applet_state`. Here is a small part of `setAppletState`:

```
/**
  * @param command the code of the command changing the state
  * @param newstate the new state to be set
  */
public void setAppletState(byte command, short newstate) {
  switch (command) {
    // ...
    case UPDATE_USER_PIN:
      // 'masterPIN.isValidated() == true' should hold here
      if (curr_applet_state == AS_LOCKED) {
        curr_applet_state = newstate;
        curr_command_state = CS_START;
      }
      break;
```

```
    // ...
  }
}
```

Finally, we get to the updateCommandState and readBigInput methods that share some global applet variables to ensure that the protocol is followed. One of them is multiple_package which indicates whether a multiple APDU command is being processed – when equal to 0 there is no multiple APDU command process in progress, when greater than 0 it is equal to the code of the multiple command being processed. The updateCommandState method first checks if the life state of the applet is the dead state and if so, it throws an exception interrupting the communication. Then it checks if there is multiple APDU processing in progress and if so if the current command belongs to the sequence of currently processed multiple APDUs throwing an exception if there is a mismatch. Finally, the method checks if the command invocation is according to the protocol defined. The global applet variable curr_command_state stores the current command state (selected, application, user administration or system administration). The code follows the diagrams shown before, throwing an exception if the command is invoked out of the allowed sequence:

```
/** @param command the code of the invoking command */
public void updateCommandState(byte command) {
  if (curr_applet_state == AS_DEAD) {
    ISOException.throwIt(SW_APPLET_DEAD);
  }
  if (multiple_package != (byte)0 && command != multiple_package) {
    ISOException.throwIt(SW_COMMAND_OUT_OF_SEQUENCE);
  }
  switch (command) {
    case VERIFY_USER_PIN:
      if (curr_applet_state != AS_PERSONALISED) {
        ISOException.throwIt(SW_COMMAND_OUT_OF_SEQUENCE);
      } else {
        if (curr_command_state == CS_APPLICATION) {
          ISOException.throwIt(SW_COMMAND_OUT_OF_SEQUENCE);
        } else {
          // do nothing, there is no state change
        }
      }
      break;
    case UPDATE_USER_PIN:
      // ...
  }
}
```

The readBigInput method uses both global variables and the form of the APDU to control the multiple APDU communication. The p2 header byte of the incoming APDU indicates the total number of APDUs to come, the p1 header byte indicates which APDU packet is being received ("p1-th out of p2 packets").

The global variables `multiple_curr` and `multiple_total` are used to control this. Whenever a multiple APDU packet is received `p1` and `p2` are checked against global variables to verify that the proper sequence is maintained. Then the data from the APDU is appended to the `bigtemp` array which collects the data from the multiple APDUs. The code for `readBigInput` is the following:

```
/**
  * @param apdu the incoming apdu to read data from
  * @param expectedLength the expected data length to read
  */
public void readBigInput(APDU apdu, short expectedLength) {
  byte buffer[] = apdu.getBuffer();
  byte ins = buffer[ISO7816.OFFSET_INS];
  byte p1 = buffer[ISO7816.OFFSET_P1];
  byte p2 = buffer[ISO7816.OFFSET_P2];
  if (p1 == (byte)0 && multiple_total == (byte)0) {
    multiple_total = p2;
    multiple_package = ins;
  } else {
    if (p1 >= p2 || p2 != multiple_total ||
        p1 != (byte)(multiple_curr + (byte)1)) {
      ISOException.throwIt(ISO7816.SW_WRONG_DATA);
    }
  }
  multiple_curr = p1;
  // append the data from APDU to bigtemp array
  multiple_readnum = apduDataOffset;
  if ((byte)(multiple_curr + (byte)1) == multiple_total) {
    resetMultiple(); // data in bigtemp ready for use
  }
}
```

5 The Framework

We did not discover any problems during extensive testing of the resulting, reengineered applet. In particular, the “rip-out” problem is no longer present. This was achieved mostly by enforcing the applet to follow strict command exchange protocol we defined during the design. Thus, the most important guideline in our framework is to define the applet life and communication protocols with UML state chart diagrams and enforce the applet to adhere to these protocols by embedding corresponding state machines into the applet itself. Moreover, the actual command processing (types of parameters, responses, etc.) should be defined and implemented in a strict way. To achieve that, we used dedicated methods (`readInput`, `sendResponse`, etc.) in our applet responsible for reading the input and sending the output. Such methods are in most part independent of the actual applet and can be used in other applications.

The artifacts of the design produced documentation that makes understanding the workings of the applet much easier for the developers and prospective

users of the applet. Further support for the developers can be provided by the CASE tool and the use of formal specification and verification techniques. We discuss these issues in the following sections.

5.1 Support from the CASE Tool

We used the support of the Together Control Center tool in many places. For the low level task like compiling, installing and testing the applet we used the extension module we wrote ourselves [Mos]. Some parts of the applet were created with the same module by using Java Card specific code patterns, which were used to introduce skeletons of the command dispatching methods automatically. Furthermore, the KeY extensions were used to introduce formal specifications into the design and to perform formal verification (see next subsections). The remaining code was engineered "by hand", however we still see possibilities to introduce further automation to the process with the support of a CASE tool in the following ways:

- Having methods like `readInput`, `readBigInput`, `verifyInput`, etc. among the standard set of Java Card helper methods, idioms and design patterns, together with the specifications. This can be easily implemented in Together Control Center.
- Generating (possibly with a little of developers help) the code for `setAppletState` and `updateCommandState` methods from the state chart diagrams like the ones presented here also incorporating formal specifications for verification. Again this should be implementable in Together Control Center, for example there are existing tools to create code from sequence diagrams and vice versa.
- The PIN check routines seem to be a good candidate for a pattern, too, as it is done in a very similar way in every Java Card applet: there is a global applet object representing the PIN, there is one APDU command that verifies the delivered PIN, sets the validation flag of the PIN object accordingly for the current terminal session and returns the result of PIN verification back to the host also indicating the number of tries left in case of failure. Then any command requiring PIN authentication can refer to PIN object by a single method call.

5.2 Formal Specification and Verification

It is almost clear that the presented dispatching methods follow the semi formal specifications we gave earlier. The `setAppletState`, `updateCommandState` and `readBigInput` and possibly `readInput` methods require a bit more attention and this is where we turn to formal specification.

First we can define the state chart behaviour more formally by giving OCL specifications like the following. Those specifications do not reflect the whole diagram set that we have shown, they are just examples. First we can tie a

given applet life state to a condition that causes the applet to be in a given state, for example:

```
context Safe_Applet
inv: self.curr_applet_state = AS_LOCKED implies
     self.userPIN.getTriesRemaining() = 0

inv: self.curr_applet_state = AS_DEAD implies
     self.masterPIN.getTriesRemaining() = 0
```

Next we can limit a set of possible command states in a given life state by the following expression:

```
context Safe_Applet
inv: self.curr_applet_state = AS_LOCKED implies
       self.curr_command_state = CS_START or
       self.curr_command_state = CS_SELECTED
```

Finally, we can describe some of the behaviour of `setAppletState` and `update-CommandState` with the following expressions:

```
context Safe_Applet::setAppletState(command: byte, newstate: short)
pre: command = UPDATE_USER_PIN and newstate = AS_PERSONALISED implies
     self.masterPIN.isValidated() and
     self.curr_applet_state = AS_LOCKED
post: self.curr_applet_state = AS_PERSONALISED and
      self.curr_command_state = CS_START

context Safe_Applet::updateCommandState(command: byte)
post: command = VERIFY_USER_PIN and
      self.curr_applet_state@pre = AS_PERSONALISED and
      self.curr_command_state@pre <> CS_APPLICATION and
      self.curr_command_state@pre <> CS_START
           implies
      self.curr_command_state = self.curr_command_state@pre
```

As mentioned before, such specifications follow exactly the diagrams and it should be possible to just generate them automatically, possibly with a little bit of user intervention.

The second set of specifications makes sure that the `readInput` and `read-BigInput` methods behave in a consistent and safe way. The following OCL invariants express the consistency conditions that the global applet variables used by the read methods should satisfy:

```
context Safe_Applet
inv: self.multiple_readnum <= self.bigtemp->size()
inv: self.multiple_package <> 0 implies
       self.multiple_curr < self.multiple_total
inv: self.multiple_package = 0 or self.multiple_package = AUTH_USER
inv: self.multiple_total > 0 implies self.multiple_package <> 0
```

Here we also stated that the authenticateUser command is the only one that can spread over multiple APDUs. The next are two preconditions that make sure the read methods do not exceed the temporary array space they operate on:

```
context Safe_Applet::readInput(apdu: APDU, expectedLength: short)
pre: self.temp <> null and expectedLength <= self.temp->size()

context Safe_Applet::readBigInput(apdu: APDU, expectedLength: short)
pre: self.bigtemp <> null and expectedLength <= self.bigtemp->size()
```

Such specifications should be associated with a general Java Card pattern that produces the read methods and put into design automatically together with the actual code.

Of course one may want to give some more in-depth specifications of the application describing its functionality or some safety properties [MM01]. In the next subsection we show how the already existing KeY tool features can be used to produce such a specification, here we briefly discuss other situations where formal specification can prove itself helpful. Suppose we would like to extend our application to keep track of unsuccessful authentication attempts and disable the access once a certain number of unsuccessful attempts has been reached (similarly to PIN code verification). This is quite straightforward to program – a counter variable needs to be increased after each failed attempt and once some threshold value is reached the following access attempts are rejected. However, when coded uncarefully, the counter may get increased during rejected attempts as well. After reaching the maximal value for a data type used (say `byte`) it will leap back to 0 ending up in an undesired, security breaching state. A typical security related specification idiom that could be used here would be that a card stays blocked after the maximum number of tries has been reached until it is explicitly released, for example, by giving the master PIN. To verify such a property one needs formalisation of Java integer arithmetics that handles properly the overflow behaviour of Java integer types. The KeY system both supports the specification idioms [GHL04] and contains formalisation of Java integer arithmetics as part of the KeY specification library [BS04].

Another area, where we turn to formal methods is to ensure that the applet data stays consistent in the case when the applet's execution terminates abnormally by ripping out the card from the reader. This requires to specify a kind of invariant for our program that holds at any point of execution of the program, not only before and after the program is executed. This is not possible to express in plain OCL, but the Dynamic Logic for Java Card used in the KeY system can handle such properties (*strong invariants*) [BM03].

The specifications we have shown and discussed are subject to formal verification. Extensive research based on the case study presented here as well as on other Java Card programs has been done in this direction [HM05, Mos04]. Here we should say that verification of quite advanced properties (including the "rip-out" properties) can be performed mostly automatically by the KeY system in a matter of minutes. Simpler properties, like the specification of the state

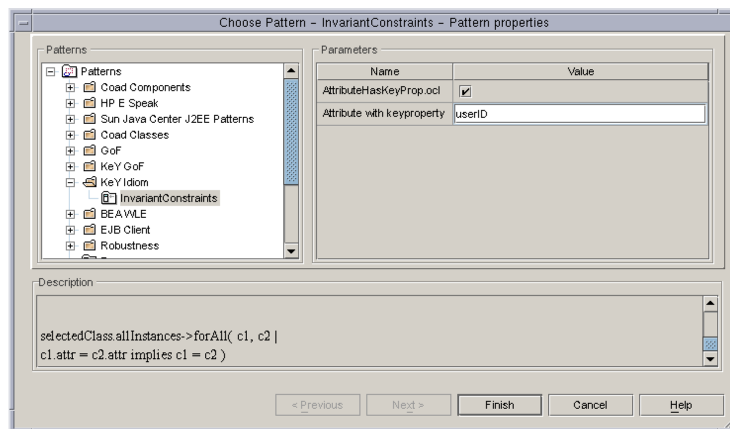


Figure 5. Applying specification patterns in the KeY system

machines controlling the life cycle and the protocol of the applet, are easily verified in the KeY system.

5.3 Employing the KeY System

Here we show how the KeY system can be used to support creation of the formal specification. Recall that one of the problems we found in `Safe_Applet` was that a single user ID can be registered more than once in the applet. First let us look at the class representing a single user record in the applet:

```
public class User {
    boolean empty = true;
    boolean enabled = true;
    byte userID = (byte)0;
    KeyData keydata = null;
}
```

Given this we would like to specify that there should not exist two (non empty) objects of this class in our applet having the same user ID. Then it can be verified formally that any code that operates on those records does not violate this condition. The condition just mentioned is a slight modification of a standard specification pattern in the KeY system called AttributeHasKeyProp as Figure 5 shows. After the pattern is applied the following invariant is produced for `User` class:

```
context User:
inv: User.allInstances->forAll(c1, c2 |
       c1.userID = c2.userID implies c1 = c2)
```

After a small modification we get what we want:

```
context User:
inv: User.allInstances->forAll(c1, c2 |
       not c1.empty and not c2.empty and
       c1.userID = c2.userID implies c1 = c2)
```

The KeY system provides a whole library of such specification patterns (some also based on the GoF patterns [GHJV99, BHSS00]) applicable to any Java Card program and, more generally, any Java program.

6 Conclusions

We presented an approach to rigorous development of Java Card applications. We have shown how UML can be used to specify an applet's behaviour and how such specifications can be translated into actual code. We have also presented how we can support formal specification and verification in Java Card development. A modern CASE tool plays an important role in our approach giving support for UML specifications, software patterns, formal verification (KeY system) and last but not least easy testing of Java Card applets. Large parts of the code we have shown were developed by hand, but we were precisely following the UML diagrams we constructed, the coding was quite straightforward and almost a one pass process – we made the applet work in the expected way in a very short time and extensive testing revealed no problems in the applet. Further research showed that formal verification of Java Card applets is feasible and can formally ensure robustness of the applet.

References

[ABB+04] 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.

[BHSS00] Thomas Baar, Reiner Hähnle, Theo Sattler, and Peter H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Infomatik*, pages 389–404. Springer, September 2000.

[BM03] 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 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.

[BML01] Dominique Bolignano, Daniel Le Métayer, and Claire Loiseaux. Formal Methods in Practice: the Missing Link. A Perspective from the

Security Area. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan, editors, *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19–23, 2000*, volume 2067 of *LNCS*. Springer-Verlag, 2001.

[Bol] Dierk Bolten. PAM authentication with an iButton. `http://www--users.rwth-aachen.de/dierk.bolten/pam_ibutton.html`.

[Boy03] 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.

[BS04] 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.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, 2000.

[FLL+02] 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.

[GHJV99] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1999.

[GHL04] Martin Giese, Reiner Hähnle, and Daniel Larsson. Rule-based simplification of OCL constraints. In Octavian Patrascoiu et al., editor, *Workshop on OCL and Model Driven Engineering at UML2004, Lisbon*, pages 84–98, 2004.

[HM05] Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[JP03] 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.

[Jür01] Jan Jürjens. Towards development of secure systems using UMLsec. In Heinrich Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE, 4th International Conference, Part of ETAPS)*, volume 2029 of *LNCS*, pages 187–200. Springer, 2001.

[MM01] 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.

[MMPH00] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The Jive system – Implementation description. Available from `http://softech.informatik.uni-kl.de/downloads/publications/jive.pdf`, 2000.

[Mos] Wojciech Mostowski. Java Card tools for Together Control Center. `http://www.cs.chalmers.se/~woj/papers/jctools.pdf`.

[Mos04] Wojciech 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.

[MPMU04] 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`.

[Obj03] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[Ope] Open Card homepage. `http://www.opencard.org`.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, Reading/MA, 2003.

Paper II

A Program Logic for Handling Java Card's Transaction Mechanism

Fundamental Approaches to Software Engineering Conference 2003, Warsaw, Poland, April 2003

A Program Logic for Handling Java Card's Transaction Mechanism

Bernhard Beckert* Wojciech Mostowski

Abstract

In this paper we extend a program logic for verifying Java Card applications by introducing a "throughout" operator that allows us to prove "strong" invariants. Strong invariants can be used to ensure "rip out" properties of Java Card programs (properties that are to be maintained in case of unexpected termination of the program). Along with introducing the "throughout" operator, we show how to handle the Java Card transaction mechanism (and, thus, conditional assignments) in our logic. We present sequent calculus rules for the extended logic.

1 Introduction

Overview. The work presented in this paper is part of the KeY project[1] [ABB+04]. One of the main goals of KeY is to provide deductive verification for a real world programming language. Our choice is the Java Card language [Che00] (a subset of Java) for programming smart cards. This choice is motivated by the following reasons. First of all Java Card applications are subject to formal verification, because they are usually security critical (e.g., authentication) and difficult to update in case a fault is discovered. At the same time the Java Card language is easier to handle than full Java (for example, there is no concurrency and no GUI). Also, Java Card programs are smaller than normal Java programs and thus easier to verify. However, there is one particular aspect of Java Card that does not exist in Java and which requires the verification mechanism to be extended with additional rules and concepts: the persistency of the objects stored on a smart card in combination with Java Card's transaction mechanism (ensuring atomicity of bigger pieces of a program) and the possibility of a card "rip out" (unexpected termination of a Java Card program by taking the smart card out of the reader/terminal). Since we want to have support for the full Java Card language in the KeY system we have to handle this aspect.

* Department of Computer Science, University of Koblenz-Landau, Germany, e-mail: beckert@uni-koblenz.de

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

To ensure that a Java Card program is "rip-out safe" we need to be able to specify "strong" invariants – invariants that must hold throughout the whole execution of a Java Card program (except when a transaction is in progress). The KeY system's deduction component uses a program logic, which is a version of Dynamic Logic modified to handle Java Card programs (Java Card DL) [Bec01, BS01a]. An extension to pure Dynamic Logic to include trace modalities "throughout" and "at least once" is presented in [BS01b]. Here we extend that work and introduce the "throughout" operator to Java Card DL (we do not introduce "at least once" since it is not necessary for handling "rip out" properties). Then we add techniques necessary to deal with the Java Card transaction mechanism (specifically conditional assignments inside the transactions). We present the sequent calculus rules for our extensions. So far we have not implemented the new rules in the KeY system's interactive prover (the implementation for the unextended Java Card DL is fully functional). But considering the extensibility and open architecture of the KeY prover it is not a difficult task.

Related Work. As said above, the work presented here is based on [BS01b], which extends pure Dynamic Logic with trace modalities "throughout" and "at least once". There exist a number of attempts to extend OCL with temporal constructs, see [BFS02] for an overview. In [TH02] temporal constructs are introduced to the Java Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Structure of the Paper. The rest of this paper is organised as follows. Section 2 gives some more details on the background and motivation of our work and some insights into the Java Card transaction mechanism. Section 3 contains a brief introduction to Java Card Dynamic Logic. Section 4 introduces the "throughout" operator in detail and presents sequent calculus rules to handle the new operator and the transaction mechanism. Section 5 shows some of the rules in action by giving simple proof examples and finally Section 6 summarises the paper.

2 Background

The KeY Project. The main goal of the KeY project [ABB+04] is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

- The specification language should be usable by people who do not have years of training in formal methods. The Object Constraint Language (OCL), which is incorporated into current version of the Unified Modelling Language (UML), is the specification language of our choice.

- The programs that are verified should be written in a "real" object-oriented programming language. We decided to use Java Card (we already stated our reasons for this in the introduction).

For verifying Java Card programs, the already mentioned Java Card Dynamic Logic has been developed within the KeY project (Section 3 contains a detailed description of this logic). The KeY system translates OCL specifications into Java Card DL formulas, whose validity can then be proved with the KeY system's deduction component.

Motivation. The main motivation for this work resulted from an analysis of a Java Card case study [Mos02]. In short, the case study involves a Java Card applet that is used for user authentication in a Linux system (instead of a password mechanism). After analysing the application and testing it, the following observation was made: the Java Card applet in question is not "rip-out safe". That is, it is possible to destroy the applet's functionality by removing (ripping out) the Java Card device from the card reader (terminal) during the authentication process. The applet's memory is corrupted and it is left in an undefined state, causing all subsequent authentication attempts to be unsuccessful (fortunately this error causes the applet to become useless but does not allow unauthorised access, which would have been worse).

It became clear that, to avoid such errors, one has to be able to specify (and if possible verify) the property that a certain invariant is maintained at all times during the applet's execution, such that it holds in particular in case of an abrupt termination. Standard UML/OCL invariants do not suffice for this purpose, because their semantics is that if they hold before a method is executed then they hold after the execution of a method. Normally it is not required for an invariant to hold in the intermediate states of a method's execution. To solve this problem, we introduce "strong" invariants, which allow to specify properties about all intermediate states of a program.

For example, the following "strong" invariant (expressed in pseudo OCL) says that we do not allow partially initialised `PersonalData` objects at any point in our program. In case the program is abruptly terminated we should end up with either a fully initialised object or an uninitialised (empty) one:

```
context PersonalData throughout:
  not self.empty implies
    self.firstName <> null and self.lastName <> null and self.age > 0
```

Since the case study was explored in the context of the KeY project, we extended the existing Java Card DL with a new modality to handle strong invariants.

The Java Card Transaction Mechanism. Here we describe the aspects of transaction handling in Java Card relevant to this paper. A full description of the transaction mechanism can be found in [Che00, Sun02a, Sun02b, Sun02c].

The memory model of Java Card differs slightly from Java's model. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which holds its contents between card sessions, and transient memory

(RAM), whose contents disappear when power loss occurs, i.e. when the card is removed from the card reader. Thus every memory element in Java Card (variable or object field) is either persistent or transient. The Java Card language specification gives the following rules (this is a slightly simplified view of what is really happening):

- All objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Thus, in Java Card all assignments like "`o.attr = 2;`", "`this.a = 3;`" and "`arr[i] = 4;`" have permanent character; that is, the assigned values will be kept after the card loses power.
- A programmer can create an array with transient elements by calling a certain method from the Java Card API (`JCSystem.makeTransient...`), but currently there is no possibility to make objects (fields) other than array elements transient.
- All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by Java Card's transaction mechanism. The following are the Java Card system calls for transactions with their description:

- `JCSystem.beginTransaction()` begins an atomic transaction. From this point on, all assignments to fields of persistent objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally (immediately).
- `JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).
- `JCSystem.abortTransaction()` aborts the transaction. All the conditional assignments are rolled back to the state in which the transaction started. Assignments to transient variables and array elements remain unchanged (as if there had not been a transaction in progress).

As an example to illustrate how transactions work in practice, consider the following fragment of a Java Card program:

```
this.a = 100;
int i = 0;
JCSystem.beginTransaction();
  i = this.a;
  this.a = 200;
JCSystem.abortTransaction();
```

After the execution of this program, the value of `this.a` is still 100 (value before the transaction), while the value of `i` now is 100 (the value it was updated to during the transaction).

Transactions do not have to be nested properly with other program constructs, e.g., a transaction can be started within one method and committed within another method. However, transactions must be nested properly with each other (which is not relevant for the current version of JAVA CARD, where the nesting depth of transactions is restricted to 1).

The whole program piece inside the transaction is seen by the outside world as if it were executed in one atomic step (considering the persistent objects). By introducing strong invariants we want to ensure the consistency of the persistent memory of a JAVA CARD applet, thus strong invariants will not (and should not) be checked within a transaction – in case our program is terminated abruptly during a transaction, the persistent variables will be rolled back to the state before the transaction was started for which the strong invariant was established.

3 JAVA CARD Dynamic Logic

Dynamic Logic [Har84, HKT00, KT90, Pra77] can be seen as an extension of Hoare logic. It is a first-order modal logic with modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of JAVA CARD statements). In the semantics of these modalities a world w (called state in the DL framework) is accessible from the current world, if the program p terminates in w when started in the current world. 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 such final states (worlds). Here, since JAVA CARD programs are deterministic, there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). 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 post-condition ψ holds. The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, i.e., ψ must only hold *if* p terminates.

The formula $\phi \rightarrow [p]\psi$ is similar to the Hoare triple $\{\phi\}p\{\psi\}$. But in contrast to Hoare logic, the set of formulas of DL is closed under the usual logical operators. In Hoare logic, the formulas ϕ and ψ are pure first-order formulas. DL allows to involve programs in the descriptions ϕ resp. ψ of states. For example, using a program, it is easy to specify that a data structure is not cyclic, which is impossible in pure first-order logic. Because all JAVA constructs are available in DL for the description of states (including `while` loops and recursion) it is not necessary to define an abstract data type *state* and to represent states as terms of that type; instead DL formulas can be used to give a (partial) description of states, which is a more flexible technique and allows one to concentrate on the relevant properties of a state.

3.1 Syntax of JAVA CARD DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators of the form $\langle \cdot \rangle$ and $[\cdot]$. The non-dynamic base logic of

our DL is a typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical with the JAVA types) nor how exactly terms and formulas are built. The definitions can be found in [Bec01]. Note that terms (which we often call "logical terms" in the following) are different from JAVA expressions – they never have side effects.

In order to reduce the complexity of the programs occurring in DL formulas, we introduce the notion of a *program context*. The context can consist of any JAVA CARD program, i.e. it is a sequence of class and interface definitions. Syntax and semantics of DL formulas are then defined with respect to a given context; and the programs in DL formulas are assumed not to contain class definitions.

The programs in DL formulas are basically executable JAVA CARD code. However, we introduced an additional construct not available in plain JAVA CARD, whose purpose is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To treat the `return` statement in the right way, it is necessary (a) to record the object field or variable x that the result is to be assigned to, and (b) to mark the boundaries of the implementation *prog* when it is substituted for the method call. For that purpose, we allow statements of the form `method-frame(`x`){`*prog*`}` to occur. This is a "harmless" extension because the additional construct is only used for proof purposes and never occurs in the verified JAVA CARD programs.

3.2 Semantics of JAVA CARD DL

The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since JAVA CARD is deterministic, that set either contains exactly one state (if p terminates normally) or is empty (if p does not terminate or terminates abruptly).

For formulas ϕ that do not contain programs, the notion of ϕ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p\rangle\phi$ is satisfied by a state s if the program p, when started in s, terminates normally in a state s' in which ϕ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models.

As mentioned above, we consider programs that terminate abruptly to be non-terminating. Thus, for example, $\langle\texttt{throw x;}\rangle\phi$ is unsatisfiable for all ϕ. Nevertheless, it is possible to express and (if true) prove the fact that a program p terminates abruptly. For example, the formula

$$e \doteq \mathsf{null} \;\rightarrow\; \langle\texttt{try\{}p\texttt{\}catch(Exception e)\{\}}\rangle(\neg\,(e \doteq \mathsf{null}))$$

is true in a state s if and only if the program p, when started in s, terminates abruptly by throwing an exception (as otherwise no object is bound to `e`).

Sequents are notated following the scheme $\phi_1,\ldots,\phi_m \;\vdash\; \psi_1,\ldots,\psi_n$ which has the same semantics as the formula $(\forall x_1)\cdots(\forall x_k)((\phi_1 \wedge \ldots \wedge \phi_m) \rightarrow (\psi_1 \vee \ldots \vee \psi_n))$, where $x_1,\ldots,x_k$ are the free variables of the sequent.

3.3 State Updates

We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ to be attached to terms and formulas, where x is a program variable, o is a term denoting an object with attribute a, and t is a term. The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\{x := t\}\phi$ has the same semantics as $\langle \texttt{x = t;} \rangle \phi$.

3.4 Rules of the Sequent Calculus

Here we only present a small number of rules necessary to get proper intuition of how the JAVA CARD DL sequent calculus works.

Notation. The rules of our calculus operate on the first *active* statement p of a program $\pi p \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately.[2] The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the following JAVA block operating on its first active command `i=0;` then the non-active prefix π and the "rest" ω are the marked parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \}finally\{ k=0; \}\}}}_{\omega}$$

In the following rule schemata, $\mathcal{U}$ stands for an arbitrary update.

The Rule for if. As the first simple example, we present the rule for the `if` statement:

$$\frac{\Gamma,\, \mathcal{U}(b \doteq \mathsf{true})\ \vdash\ \mathcal{U}\langle\, \pi\ p\ \omega\,\rangle\phi \quad \Gamma,\, \mathcal{U}(b \doteq \mathsf{false})\ \vdash\ \mathcal{U}\langle\, \pi\ q\ \omega\,\rangle\phi}{\Gamma\ \vdash\ \mathcal{U}\langle\, \pi\ \texttt{if}\,(b)\ \{p\}\,\texttt{else}\,\{q\}\ \omega\,\rangle\phi} \qquad \text{(R1)}$$

The rule has two premises, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premises hold in a state, then the conclusion is true in that state. In particular, if the two premises are valid, then the conclusion is valid. In practice, rules are applied from bottom to top: from the old proof obligation new proof obligations are derived. As the `if` rule demonstrates, applying a rule from bottom to top corresponds to a symbolic execution of the program to be verified.

[2] In DL versions for simple artificial programming languages, where no prefixes are needed, any formula of the form $\langle pq \rangle \phi$ can be replaced by $\langle p \rangle \langle q \rangle \phi$. In our calculus, splitting of $\langle \pi p q \omega \rangle \phi$ into $\langle \pi p \rangle \langle q \omega \rangle \phi$ is not possible (unless the prefix π is empty) because πp is not a valid program; and the formula $\langle \pi p \omega \rangle \langle \pi q \omega \rangle \phi$ cannot be used either because its semantics is in general different from that of $\langle \pi p q \omega \rangle \phi$.

The Assignment Rule and Handling State Updates. The assignment rule

$$\frac{\Gamma \vdash \mathcal{U}\{loc := expr\}\langle\, \pi\ \omega\,\rangle\phi}{\Gamma \vdash \mathcal{U}\langle\, \pi\ \texttt{loc = expr;}\ \omega\,\rangle\phi} \qquad \text{(R2)}$$

adds the assignment to the list of updates $\mathcal{U}$. Of course, this does not solve the problem of computing the effect of an assignment, which is particularly complicated in Java because of aliasing. This problem is postponed and solved by rules for simplifying updates that are attached to formulas whenever possible (without branching the proof).

The assignment rule can only be used if the expression *expr* is a logical term. Otherwise, other rules have to be applied first to evaluate *expr* (as that evaluation may have side effects). For example, these rules replace the formula $\langle\texttt{x = ++i;}\rangle\phi$ with $\langle\texttt{i = i+1; x = i;}\rangle\phi$.

4 Extension for Handling "Throughout" and Transactions

In some regard Java Card DL (and other versions of DL) lacks expressivity – the semantics of a program is a relation between states; formulas can only describe the input/output behaviour of programs. Java Card DL cannot be used to reason about program behaviour not manifested in the input/output relation. Therefore, it is inadequate for verifying strong invariants that must be valid throughout program execution.

Following [BS01b], we overcome this deficiency and increase the expressivity of Java Card DL by adding a new modality $[\![\cdot]\!]$ ("throughout"). In the extended logic, the semantics of a program is the sequence of all states its execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of the intermediate states of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the Java Card DL calculus extended with additional sequent rules for $[\![\cdot]\!]$ presented in Section 4.1.

A "throughout" property (formula) has to be checked after every single field or variable assignment, i.e., the sequent rules for the throughout modality will have more premises and branch more frequently. According to the Java Card runtime environment specification [Sun02b], each single field or variable assignment is atomic. This matches exactly Java Card DL's notion of a single update. Thus, a "throughout" property has to hold after every single Java Card DL update. However, additional checks have to be suspended when a transaction is in progress. This will require marking the modality (resp. the program in the modality) with a tag saying that a transaction is in progress, so that different rules apply. Since transactions do not have to be nested properly with other program constructs, enclosing a transaction in a block with a separate set of rules for that kind of block (like the `method-frame` blocks) is not possible.

In addition, we have to cover conditional assignments and assignment roll-back (after `abortTransaction`) in the calculus. This not only affects the

"throughout" modality, but the $\langle\cdot\rangle$ and $[\cdot]$ modalities as well, since rolling back an assignment affects the final program state.

In practice only formulas of the form $\phi \to [\![p]\!]\phi$ will be considered. If transient arrays are involved in ϕ (explicitly or implicitly), one also has to prove $\phi \to \langle \texttt{initAllTransientArrays();} \rangle \phi$, i.e., that after a card rip-out the reinitialisation of transient arrays preserves the invariant.

4.1 Additional Sequent Calculus Rules for the Throughout Modality

Below, we present the assignment and the `while` rules for the $[\![\cdot]\!]$ modality. Due to space restrictions, we cannot list all additional rules. However, the other loop rules are very similar to the `while` rule, and all other $[\![\cdot]\!]$ rules are essentially the same as for $[\cdot]$ – except for the transaction rules which we present in the next subsection.

The Assignment Rule for $[\![\cdot]\!]$. An assignment `loc = expr;` is an atomic program, if *expr* is a logical term (and, in particular, is free of side effects and can be computed in a single step). By definition, its semantics is a trace consisting of the initial state s and the final state $s' = \{loc := val_s(expr)\}s$. Therefore, the meaning of $[\![\texttt{loc = expr;}]\!]\phi$ is that ϕ is true in both s and s', which is what the two premises of the following assignment rule express:

$$\frac{\Gamma \vdash \mathcal{U}\phi \quad \Gamma \vdash \mathcal{U}\{loc := expr\}[\![\,\pi\ \omega\,]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\,\pi\ \texttt{loc = expr;}\ \omega\,]\!]\phi} \tag{R3}$$

The left premise states that the formula ϕ has to hold in the state s before the assignment takes place. The right premise says that ϕ has to hold in the state s' after the assignment – and in all states thereafter during the execution of the rest ω of the program. As for the other modalities, the precondition for an application of the assignment rule is that *expr* is a logical term (and, in particular, free of side effects).

It is easy to see that using this rule causes some extra branching of the proofs involving the $[\![\cdot]\!]$ modality. This branching is unavoidable due to the fact that the strong invariant has to be checked (evaluated) for each intermediate state of the program execution. However, many of those branches, which do not involve Java Card programs any more, can be closed automatically.

The while Rule for $[\![\cdot]\!]$. Another essential programming construct, where the rule for the $[\![\cdot]\!]$ modality differs from the corresponding rule for the $[\cdot]$ modality, is the `while` loop. As in the case of the `while` rule for the $[\cdot]$ modality a user has to supply a loop invariant *Inv*. Intuitively, the rule establishes three things:

1. In the state before the loop is executed, some invariant *Inv* holds.
2. If the body of the loop terminates normally (there is no `break` and no exception is thrown but possibly `continue` is used) then at the end of a single execution of the loop body the invariant *Inv* has to hold again.

3. Provided *Inv* holds, the formula ϕ has to hold during and continuously after loop body execution in all of the following cases: (i) when the loop body is executed once and terminates normally, (ii) when the loop body is not executed (the loop condition is not satisfied), and (iii) when the loop body terminates abruptly (by `break`, `continue`, or throwing an exception) resulting in a termination of the whole loop.

Formally, the `while` rule for $[\![\cdot]\!]$ is the following:

$$\frac{\Gamma \vdash \mathcal{U}Inv \quad Inv \vdash \langle\alpha\rangle true,\ [\beta]Inv \quad Inv \vdash [\![\pi\ \beta\ \omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\pi\ \lambda\mathtt{while(}a\mathtt{)\ \{}p\mathtt{\}}\ \omega]\!]\phi} \tag{R4}$$

where

$\alpha \equiv \mathtt{if(}a\mathtt{)\ \{}l_{break}\mathtt{:\ \{try\ \{}l_{cont}\mathtt{:\ \{}p'\mathtt{\}\ abort;\}\ catch(Exception\ e)\{\}\}\}}$
$\beta \equiv \mathtt{if(}a\mathtt{)}\ l_{cont}\mathtt{:}\ l_{break}\mathtt{:\ \{}p'\mathtt{\}}$

In the above rule, λ is a (possibly empty) sequence "$l_1 : \ldots l_n :$" of labels, and p' is p with (a) every "`continue;`" and every "`continue` l_i`;`" changed to "`break` l_{cont}`;`" and (b) every "`break;`" and every "`break` l_i`;`" changed to "`break` l_{break}`;`". The three premises establish the three conditions listed above, respectively. When the program p' terminates normally, the `abort` in α is reached and, thus, the formula $\langle\alpha\rangle true$ evaluates to *false* and $[\beta]Inv$ has to be proved. Enclosing program p' in "`if(`a`)`..." takes care of both cases, where the loop body is executed (intermediate loop body execution) and where it is not executed (loop exit). They are later in the proof considered separately by applying the rule for `if`.

4.2 Additional Sequent Calculus Rules for Transactions

Additional Syntax. Before presenting the sequent rules for transactions, we first have to introduce some new programming constructs (statements) and transaction markers to Java Card DL. The three new statements we need are the following:

- `bT` – Java Card beginning of a transaction,
- `cT` – Java Card end of a transaction (commit),
- `aT` – Java Card end of a transaction (abort).

These statements are used in the proof when the transaction is started resp. finished in the Java Card program. The statements are only part of the rules and not the Java Card programming language. Thus for example, when a transaction is started in a Java Card program by a call to `JCSystem.beginTransaction()` the calculus assumes the following implementation of `beginTransaction()`:

```
public class JCSystem {
  private static int _transDepth = 0;
  public static void beginTransaction() throws TransactionException {
    if(_transDepth > 0)
      TransactionException.throwIt(TransactionException.IN_PROGRESS);
    _transDepth++;
    bT;
  }
  ...
```

Thus, when we encounter any of `bT`, `cT` or `aT` in our proof we can assume they are properly used (nested).

The second thing we need is the possibility to mark modalities (resp. the programs they contain) with a tag saying that a transaction is in progress. We will use two kinds of tags and make them part of the inactive program prefix π in the sequent. The two markers are:

- TRcommit: – a transaction is in progress and is expected to be committed (`cT`),
- TRabort: – a transaction is in progress and is expected to aborted (`aT`).

This distinction is very helpful in taking care of conditional assignments – since we know how the transaction is going to terminate "beforehand" we can treat conditional assignments correspondingly, commit them immediately in the first case or "forget" them in the second case. Shortly we will show exactly how this is done in the rules.

Rules for Beginning a Transaction. For each of the three operators ($\langle\cdot\rangle$, $[\cdot]$, $[\![\cdot]\!]$) there is one "begin transaction" rule. The rules for $\langle\cdot\rangle$ and $[\cdot]$ are identical, so we only show one of them:

$$\frac{\Gamma \vdash \mathcal{U}\phi \quad \Gamma \vdash \mathcal{U}[\![\textsf{TRcommit:}\, \pi\ \omega\,]\!]\phi \quad \Gamma \vdash \mathcal{U}[\![\textsf{TRabort:}\, \pi\ \omega\,]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\,\pi\ \texttt{bT;}\ \omega\,]\!]\phi} \tag{R5}$$

$$\frac{\Gamma \vdash \mathcal{U}\langle\textsf{TRabort:}\, \pi\ \omega\,\rangle\phi \quad \Gamma \vdash \mathcal{U}\langle\textsf{TRcommit:}\, \pi\ \omega\,\rangle\phi}{\Gamma \vdash \mathcal{U}\langle\,\pi\ \texttt{bT;}\ \omega\,\rangle\phi} \tag{R6}$$

In case of the $[\![\cdot]\!]$ operator the following things have to be established. First of all, ϕ has to hold before the transaction is started. Then we split the sequent into two cases: the transaction will be terminated by a commit, or the transaction will be terminated by an abort. In both cases the sequent is marked with the proper tag, so that corresponding rules can be applied later, depending on the case. The $\langle\cdot\rangle$ and $[\cdot]$ rules for "begin transaction" are very similar to $[\![\cdot]\!]$ except that ϕ does not have to hold before the transaction is started.

Rules for Committing and Aborting Transactions. These rules are the same for all three operators, so we only show the $[\![\cdot]\!]$ rules.

The first two rules apply when the expected type of termination is encountered (“TRcommit:” for commit resp. “TRabort:” for abort). In that case, the corresponding transaction marker is simply removed, which means that the transaction is no longer in progress. These are the rules:

$$\frac{\Gamma \ \vdash \ \mathcal{U}[\![\pi\ \omega]\!]\phi}{\Gamma \ \vdash \ \mathcal{U}[\![\mathsf{TRcommit\!:} \pi\ \mathtt{cT;}\ \omega]\!]\phi} \tag{R7}$$

$$\frac{\Gamma \ \vdash \ \mathcal{U}[\![\pi\ \omega]\!]\phi}{\Gamma \ \vdash \ \mathcal{U}[\![\mathsf{TRabort\!:} \pi\ \mathtt{aT;}\ \omega]\!]\phi} \tag{R8}$$

We also have to deal with the case where the transaction is terminated in an unexpected way, i.e., a commit is encountered when the transaction was expected to abort and vice versa. In this case we simply use an axiom rule, which immediately closes the proof branch (one of the proof branches produced by the “begin transaction” rule will always become obsolete since each transaction can only terminate by either commit or abort). The rules are the following:

$$\frac{}{\Gamma \ \vdash \ \mathcal{U}[\![\mathsf{TRabort\!:} \pi\ \ cT;\ \omega]\!]\phi} \tag{R9}$$

$$\frac{}{\Gamma \ \vdash \ \mathcal{U}[\![\mathsf{TRcommit\!:} \pi\ \mathtt{aT;}\ \omega]\!]\phi} \tag{R10}$$

Rules for Conditional Assignment Handling within a Transaction. Finally, we come to the essence of conditional assignment handling in our rules. In case the transaction is expected to commit, no special handling is required – all the assignments are executed immediately. Thus, the rule for an assignment in the scope of $[\![\mathsf{TRcommit\!:} \ldots]\!]$ is the same as the rule for an assignment within $[\cdot]$ (the same holds for all other programming constructs). Note that, even using the $[\![\mathsf{TRcommit\!:} \ldots]\!]$ modality, ϕ only has to hold at the end of the transaction, which is considered to be atomic.

$$\frac{\Gamma \ \vdash \ \mathcal{U}\{loc := expr\}[\![\mathsf{TRcommit\!:} \pi\ \omega]\!]\phi}{\Gamma \ \vdash \ \mathcal{U}[\![\mathsf{TRcommit\!:} \pi\ \mathtt{loc\ =\ expr;}\ \omega]\!]\phi} \tag{R11}$$

In case a transaction is terminated by an abort, all the conditional assignments are rolled back as if they were not performed. If we know that the transaction is going to abort because of a TRabort: marker, we can deliberately choose not to perform the updates to persistent objects as we encounter them. However, we cannot simply skip them since the new values assigned to (fields of) persistent objects during a transaction may be referred to later in the same transaction (before the abort). The idea to handle this, is to assign the new value to a copy of the object field or array element while leaving the original unchanged, and to replace – until the transaction is aborted – references to persistent fields and array elements by references to their copies holding the new value. Note that if an object field to which no new value has been assigned is referenced (and for which therefore no copy has been initialised), the original reference is used.

Making this work in practice, requires changing the assignment rule for the cases where a transaction is in progress and is expected to abort (i.e., where the "TRabort:" marker is present). Also the rules for update evaluation change a bit, which changes the semantics of an update as well, see description of the rule below. The following is the assignment rule for the $[\![\cdot]\!]$ modality with the "TRabort:" tag present. The corresponding rules for $\langle\cdot\rangle$ and $[\cdot]$ are the same:

$$\frac{\Gamma \ \vdash \ \mathcal{U}\{loc' := expr'\}[\![\textsf{TRabort:}\,\pi\ \omega\,]\!]\phi}{\Gamma \ \vdash \ \mathcal{U}[\![\textsf{TRabort:}\,\pi\ \texttt{loc = expr;}\ \omega\,]\!]\phi} \qquad \text{(R12)}$$

As usual *expr* has to be a logical term. To handle objects fields persistent arrays elements, all sub-expressions such as $obj.a_1.arr[e].a_2\ldots$ in *expr* are replaced by $obj.a'_1.arr'[e']'.a'_2\ldots$ in $expr'$ (for object fields the prime denotes a copy of that field and for array access function [] the prime denotes a "shadow" access function that operates on copies of elements of a given array). The first reference *obj* or *arr* (as in $arr[i].a$) in *expr* is not primed, since it is either a local variable, which is not persistent, or the `this` reference, which is not assignable, or a static class reference, like `SomeClass`, which also can be viewed as not assignable. All subexpressions that are local variables are left unchanged in $expr'$. The expression *loc* on the left side of the assignment and the subexpression e are changed into loc' resp. e' in the same way as all the subexpressions in *expr*.

As mentioned, the semantics of an update has to be changed to take care of the cases when a copy of an object's field has not been initialised. In the new semantics, if the value of $obj.a'$ or $arr[i]'$ is referred to in an update but is not known (i.e., there was no such value assigned in the preceding updates) then it is considered to be equal to $obj.a$ or $arr[i]$, respectively.

The assignments to the copies are not visible outside the transaction, where the original values are used again – the effect of a roll-back is accomplished. Each separate transaction has to have its own copies of fields or array elements, so the second encountered transaction can, for example, use $''$, the third one $'''$, etc.

One more thing that we have to handle here is the case when the programmer explicitly defines an array to be transient (the above rule assumes that it was not the case). It is not possible to know beforehand which arrays are transient and which are not, since they are defined to be transient by reference and not by name. This problem can be treated by adding an extra field to each array (only in the rules) indicating whether the given array is transient or persistent (rules for initialising arrays can set this field). Then for each occurrence of array reference *arr* in *loc* and *expr* in rule (R12) we can split the proof into two cases, following the schema:

$$\frac{\begin{array}{c}\Gamma,\,\mathcal{U}(o.arr'.trans \doteq \textsf{true}) \ \vdash \ \mathcal{U}\{o.arr'[e'] := expr'\}[\![\textsf{TRabort:}\,\pi\ \omega\,]\!]\phi \\ \Gamma,\,\mathcal{U}(o.arr'.trans \doteq \textsf{false}) \ \vdash \ \mathcal{U}\{o.arr'[e']' := expr'\}[\![\textsf{TRabort:}\,\pi\ \omega\,]\!]\phi\end{array}}{\Gamma \ \vdash \ \mathcal{U}[\![\textsf{TRabort:}\,\pi\ \texttt{o.arr[e] = expr;}\ \omega\,]\!]\phi} \qquad \text{(R13)}$$

The remaining rules for $[\![\mathsf{TRabort}{:}\,\cdot]\!]$ (i.e., for other programming constructs) are the same as for $[\cdot]$, and the remaining rules for $[\mathsf{TRabort}{:}\,\cdot]$ and $\langle\mathsf{TRabort}{:}\,\cdot\rangle$ are the same as if there were no transaction marker.

5 Examples

In the following, we show two examples of proofs using the above rules. The first example shows how the $[\![\cdot]\!]$ assignment and `while` rules are used, the second example shows the transaction rules in action. The formula we are trying to prove in the second example is deliberately not provable and shows the importance of the transaction mechanism when it comes to "throughout" properties.

The proofs presented here may look like tedious work, but most of the steps can be done automatically, in fact the only place where user interaction is required, is providing the loop invariant. The KeY system provides necessary mechanisms to perform proof steps automatically whenever possible.

Example 1. Consider the following program p:

```
x = 3;
while (x < 10) {
  if(x == 2) x = 1;
  else x++;
}
```

We show that throughout the execution of this program, the strong invariant $\phi \equiv x \geq 2$ holds, i.e., we prove the formula $x \geq 2 \rightarrow [\![p]\!]x \geq 2$.

Proof. We start the proof with the sequent

$$x \geq 2 \;\vdash\; [\![\texttt{x = 3;}\ldots]\!]x \geq 2 \tag{1}$$

Applying the assignment rule (R3) to (1) produces two proof obligations:

$$x \geq 2 \;\vdash\; x \geq 2 \tag{2}$$

$$x \geq 2 \;\vdash\; \{x := 3\}[\![\texttt{while}\ \ldots]\!]x \geq 2 \tag{3}$$

Sequent (2) is valid. Applying the `while` rule (R4) to (3) with $x \geq 3$ as the loop invariant *Inv* gives us the three proof obligations below. Note that here it is necessary to use $x \geq 3$ as the invariant. Using $Inv' = \phi = x \geq 2$ would not be enough, because the statement `x = 1` inside the `if` statement could not be discarded and x would be assigned 1, which would break the $x \geq 2$ property.

$$x \geq 2 \;\vdash\; \{x := 3\}x \geq 3 \tag{4}$$

$$x \geq 3 \;\vdash\; [\![\texttt{if(x<10)}\lambda\texttt{\{}\beta\texttt{\}}]\!]x \geq 2 \tag{5}$$

$$x \geq 3 \;\vdash\; \langle\alpha\rangle true,\ [\texttt{if(x<10)}\lambda\texttt{\{}\beta\texttt{\}}]x \geq 3 \tag{6}$$

where:

$$\alpha \equiv \texttt{if(x < 10) \{...}\beta\texttt{; abort;...\}}$$
$$\beta \equiv \texttt{if(x == 2) x = 1; else x++;}$$
$$\lambda \equiv l_{cont}\colon l_{break}\colon$$

Reducing (4) results in $x \geq 2 \vdash 3 \geq 3$ which is valid. In the program α, `abort` will be reached (for $x < 10$) after some proof steps that we do not show here due to space restrictions. Since `abort` is a non-terminating program formula $\langle\texttt{abort;}\rangle\phi$ is always false. Thus, (6) can be reduced to:

$$x \geq 3,\ x < 10 \ \vdash\ [\texttt{if(x<10)}\lambda\texttt{\{}\beta\texttt{\}}]x \geq 3 \tag{7}$$

We are left with (5) and (7) to prove. Applying the `if` rule to (5) gives two proof obligations:

$$x \geq 3,\ x < 10 \ \vdash\ [\![\lambda\texttt{\{}\beta\texttt{\}}]\!]x \geq 2 \tag{8}$$
$$x \geq 3,\ x \geq 10 \ \vdash\ [\![\ ]\!]x \geq 2 \tag{9}$$

Sequent (9) is reduced to $x \geq 3,\ x \geq 10 \ \vdash\ x \geq 2$, which is valid. After simplifying and applying the `if` rule to (8) we get:

$$x \geq 3,\ x < 10,\ x \doteq 2 \ \vdash\ [\![\texttt{x = 1;}]\!]x \geq 2 \tag{10}$$
$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ [\![\texttt{x = x + 1;}]\!]x \geq 2 \tag{11}$$

Sequent (10) is valid by contradiction in the antecedent. Applying the assignment rule to (11) gives two proof obligations:

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ x \geq 2 \tag{12}$$
$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ \{x := x + 1\}[\![\ ]\!]x \geq 2 \tag{13}$$

Sequent (12) is valid. Sequent (13) is reduced to:

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ x + 1 \geq 2 \tag{14}$$

Sequent (14) is valid. We can go back to (7) and apply the `if` rule yielding two proof obligations:

$$x \geq 3,\ x < 10,\ x < 10 \ \vdash\ [\lambda\texttt{\{}\beta\texttt{\}}]x \geq 3 \tag{15}$$
$$x \geq 3,\ x \geq 10,\ x < 10 \ \vdash\ [\ ]x \geq 3 \tag{16}$$

Sequent (16) is valid by contradiction in the antecedent. Applying the `if` rule to (15) gives us:

$$x \geq 3,\ x < 10,\ x \doteq 2 \ \vdash\ [\texttt{x = 1;}]x \geq 3 \tag{17}$$
$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ [\texttt{x = x + 1;}]x \geq 3 \tag{18}$$

Again (17) is valid by contradiction in the antecedent. Applying the assignment rule to (18) gives:

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ \{x := x + 1\}[\ ]x \geq 3 \tag{19}$$

$$\dfrac{\dfrac{}{(2)} \quad \dfrac{\dfrac{}{(4)} \quad \dfrac{\dfrac{\dfrac{}{(10)} \quad \dfrac{\dfrac{}{(12)} \quad \dfrac{\dfrac{}{(14)}}{(13)}}{(11)}\,\text{(R3)}}{(8)}\,\text{(R1)} \quad \dfrac{}{(9)}}{(5)}\,\text{(R1)} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{(17)}\quad \dfrac{\dfrac{\dfrac{}{(20)}}{(19)}}{(18)}\,\text{(R2)}}{(15)}\,\text{(R1)} \quad \dfrac{}{(16)}}{(7)}}{(6)}}{(3)}\,\text{(R4)}}{(1)}\,\text{(R3)}$$

Figure 1. The proof from Example 1

which is reduced to:

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \ \vdash\ x + 1 \geq 3 \tag{20}$$

Sequent (20) is valid and thus we have proved the initial formula. Figure 1 shows the proof tree for this example.

Example 2. Now consider the following program p (fields of o are persistent):

```
bT;
  o.x = 60;
  o.y = 40;
cT;
t = o.x;
o.x = o.y;
o.y = t;
```

We try to prove that the strong invariant $o.x + o.y \doteq 100$ holds throughout the execution of this program The formula to prove is the following:

$$o.x + o.y \doteq 100 \ \vdash\ [\![\texttt{bT;}\ldots]\!] o.x + o.y \doteq 100 \tag{1}$$

Note that this is not provable.

Proof. We start our proof by applying the begin transaction rule to (1) yielding three proof obligations:

$$o.x + o.y \doteq 100 \ \vdash\ o.x + o.y \doteq 100 \tag{2}$$

$$o.x + o.y \doteq 100 \ \vdash\ [\![\textsf{TRcommit:}\,\texttt{o.x = 60;}\ldots]\!] o.x + o.y \doteq 100 \tag{3}$$

$$o.x + o.y \doteq 100 \ \vdash\ [\![\textsf{TRabort:}\,\texttt{o.x = 60;}\ldots]\!] o.x + o.y \doteq 100 \tag{4}$$

Sequent (2) is obviously valid. Applying the assignment rule to (4) gives:

$$o.x + o.y \doteq 100 \ \vdash\ \{o.x' := 60\}[\![\textsf{TRabort:}\,\texttt{o.y = 40;}\ldots]\!] o.x + o.y \doteq 100 \tag{5}$$

Notice that since we are inside a transaction the assignment rule does not branch. Again the assignment rule to (5) gives:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x' := 60\}\{o.y' := 40\}[\![\mathsf{TRabort}\colon \texttt{cT;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{6}$$

Applying the exit transaction rule (R9) (transaction commits unexpectedly) to (6) proves (6) to be valid. Applying the assignment rule to (3) gives:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}[\![\mathsf{TRcommit}\colon \texttt{o.y = 40;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{7}$$

Again the assignment rule to (7) gives:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}[\![\mathsf{TRcommit}\colon \texttt{cT;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{8}$$

Applying the exit transaction rule to (8) gives:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}[\![\texttt{t = o.x;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{9}$$

Applying the assignment rule to (9) gives two proof obligations:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}o.x + o.y \doteq 100 \tag{10}$$

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}\{t := o.x\}[\![\texttt{o.x = o.y;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{11}$$

Sequent (10) is reduced to:

$$o.x + o.y \doteq 100 \ \vdash \ 60 + 40 \doteq 100 \tag{12}$$

which is valid. Applying the assignment rule to (11) gives two proof obligations:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}\{t := o.x\}o.x + o.y \doteq 100 \tag{13}$$

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}[\![\texttt{o.y = t;}\ \ldots]\!]o.x + o.y \doteq 100 \tag{14}$$

Sequent (13) is reduced to:

$$o.x + o.y \doteq 100 \ \vdash \ 60 + 40 \doteq 100 \tag{15}$$

which is valid. Applying the assignment rule to (14) gives again two proof obligations:

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}o.x + o.y \doteq 100 \tag{16}$$

$$o.x + o.y \doteq 100 \ \vdash \ \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}\{o.y := t\}[\![\ ]\!]o.x + o.y \doteq 100 \tag{17}$$

$$\dfrac{\dfrac{}{(2)} \quad \dfrac{\dfrac{\dfrac{}{(6)}\,(R9)}{(5)}\,(R11)}{(4)}\,(R11) \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(12)}}{(10)} \quad \dfrac{\dfrac{\dfrac{}{(15)}}{(13)} \quad \dfrac{\dfrac{\mathbf{(18)}}{(16)} \quad \dfrac{\dfrac{}{(19)}}{(17)}}{(14)}\,(R3)}{(11)}\,(R3)}{(9)}\,(R3)}{(8)}\,(R7)}{(7)}\,(R12)}{(3)}\,(R12)}{(1)}\,(R5)$$

Figure 2. The proof from Example 2

Sequents (16) and (17) are reduced to, respectively:

$$o.x + o.y \doteq 100 \ \vdash \ 40 + 40 \doteq 100 \tag{18}$$

$$o.x + o.y \doteq 100 \ \vdash \ 60 + 40 \doteq 100 \tag{19}$$

Sequent (19) is obviously valid. Sequent (18) is not provable. Inspecting our program closely shows that indeed both $o.x$ and $o.y$ are equal to 40 at some point (after line 6 is executed) and their sum is 80, which violates the property we wanted to prove. Figure 2 shows the proof tree for this example with an open proof goal.

6 Conclusions and Future Work

We introduced the "throughout" modality (and, thus, strong invariants) to Java Card Dynamic Logic and presented the necessary sequent calculus rules to handle this modality and conditional assignments in Java Card transactions. Introduction of this modality was a manageable task and the set of presented rules is quite easy to use in theorem proving as shown in the examples. Our future plan is to implement our rules in the KeY prover and then try our calculus with "real" examples.

References

[ABB+04] 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.

[Bec01] 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.

[BFS02] Julian C. Bradfield, Juliana Küster Filipe, and Perdita Stevens. Enriching OCL using observational mu-calculus. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[BS01a] 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.

[BS01b] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.

[Che00] Zhiqun Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, 2000.

[Har84] David Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*. Reidel, 1984.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[KT90] Dexter Kozen and Jerzy Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.

[Mos02] 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`.

[Pra77] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.

[Sun02a] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Application Programming Interface*, September 2002.

[Sun02b] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Runtime Environment Specification*, September 2002.

[Sun02c] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Virtual Machine Specification*, September 2002.

[TH02] Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.

Paper III | Specifying Java Card API in OCL

OCL 2.0: Industry Standard or Scientific Playground? Satellite Workshop at UML Conference 2003, San Francisco, U.S.A., October 2003

Specifying Java Card API in OCL

Daniel Larsson* Wojciech Mostowski

Abstract

We discuss the development of an OCL specification for the Java Card API. The main purpose of this specification is to support and aid the verification of Java Card programs in the KeY system. The main goal of the KeY system is to integrate object oriented design and formal methods. The already existing specification written in JML (Java Modelling Language) has been used as a starting point for the development of the OCL specification. In this paper we report on the problems that we encountered when writing the specification and their solutions, we present the most interesting parts of the specification, we report on successful verification attempts and finally we evaluate OCL and compare it to JML in the context of Java Card program specification and verification.

1 Introduction

This paper reports on the development of an OCL specification for the Java Card API [Sun02]. Java Card [Che00] is a subset of the Java programming language and is used to program smart cards. The Java Card API (Application Programming Interface) is a set of library classes used in Java Card programs. Java Card API is a much smaller version of the standard Java API and is specifically designed for smart card programming. The OCL specification is necessary to perform formal verification of such programs when the implementation of the API classes is not available. Even if the API implementation is available, having the OCL specification helps to avoid repetitive work of proving the API implementation each time API method is used in a Java Card program. The secondary purpose of writing the specification is to document the behaviour of the Java Card API in a formal way. We discuss the problems we encountered when writing the specification in OCL and their solutions. We present some of the most interesting parts of the specification and report on successful verification attempts of the reference implementation of Java Card API with respect to our specification. Finally, we evaluate OCL and compare it to JML in the context of this work. This paper summarises results from [Lar03].

In the following section we give more details about the background and motivation of this work. In Section 3 we give a detailed report on the development

* Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: `danla@cs.chalmers.se`

of the specification, in Section 4 we present some interesting parts of our specification, in Section 5 we evaluate OCL in the context of the presented work and finally we conclude in Section 6.

2 Background

2.1 The KeY Project

The work presented in this paper has been carried out as part of the KeY project[1] [ABB+04]. The main goal of the KeY project is to enhance a commercial CASE (Computer Aided Software Engineering) tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are:

1. The specification language should be usable by people who do not have years of training in formal methods. The Object Constraint Language (OCL) [WK03], which is incorporated into the current version of the Unified Modelling Language (UML) [Obj03], is the specification language of our choice.

2. The programs that are verified should be written in a "real" object-oriented programming language. We decided to use Java Card. This choice is motivated by the following reasons. First of all, many Java Card applications are subject to formal verification, because they are usually security critical (for example, authentication) and difficult to update in case a fault is discovered. At the same time the Java Card language is easier to handle than full Java (for example, there is no concurrency and no GUI – see Section 2.2). Also, Java Card programs are smaller than normal Java programs and thus easier to verify.

The architecture of the KeY system is shown in Figure 1. It is built on top of a commercial CASE tool (Borland Together Control Center)[2] and extends it with facilities for formal specification and verification of Java programs in the following ways:

- It supports creation and manipulation of OCL constraints, for example, the KeY system can automatically create a partial OCL specification by instantiating an OCL template (commonly used OCL specification schema) or use a syntax based editor to create OCL expressions.

- The deduction component is used to actually construct proofs for Java Dynamic Logic proof obligations generated from the UML model, OCL constraints and Java implementation. The deduction component is an interactive verification system based on Java Dynamic Logic, a logic specifically designed for formal verification of Java programs [Bec01].

[1] `http://www.key-project.org`
[2] `http://www.borland.com/together/`

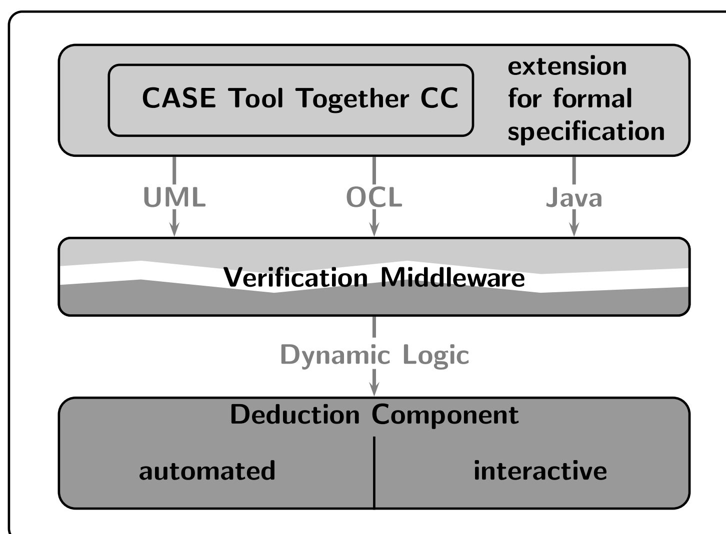


Figure 1. The architecture of the KeY system

2.2 Java Card and Java Card API

Java Card is a technology that provides means to program smart cards with (a subset of) the Java programming language. Due to limited resources of smart cards, the Java Card language is limited in a number of ways as compared to full Java. The following is the list of features that are not supported in Java Card: 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.

The Java Card API is a library that handles smart card specific features, like Application Protocol Data Units (APDUs – used for communication between the card and the rest of the world), Application IDentifiers (AIDs), Java Card specific system routines, PIN codes, etc. [Sun02]. Some of the packages included in the Java Card API 2.2 are the following:

- `java.lang` – provides classes that are fundamental to the design of the Java Card technology subset of the Java programming language. The classes in this package are derived from `java.lang` in the standard Java programming language and represent the core functionality required by the Java Card Virtual Machine.

- `javacard.framework` – provides a framework of classes and interfaces for building, communicating and working with Java Card applets. These classes and interfaces provide the minimum required functionality for a

Java Card environment. The key classes and interfaces in this package are:

- `AID` – encapsulates the Application IDentifier (AID) associated with an applet.
- `APDU` – provides methods for controlling card input and output.
- `Applet` – the base class for all Java Card applets on the card. It provides methods for working with applets to be loaded onto, installed into and executed on a Java Card compliant smart card.
- `JCSystem` – provides methods for controlling system functions such as transaction management, transient objects, object deletion mechanism, resource management, and inter-applet object sharing.
- `Util` – provides convenience methods for working with arrays and array data.

The whole Java Card API consists of 57 classes and interfaces, many of which are very simple (for example, exception classes).

2.3 Use Cases for OCL Specification of the Java Card API

One of the purposes of the KeY system is the possibility to formally verify Java Card applications. To successfully verify a program that uses the Java Card API one has to have access to either the implementation of the API or its formal specification. Since the implementation of the API is usually not available (especially when the methods are native), the latter is the solution we are aiming for. Let us look at an example to illustrate how the Java Card API specification is used in the verification process. Suppose we have implemented a method `aMethod` in our Java Card program. We now want to verify that the implementation satisfies the formal specification (the pair of pre- and postconditions) of method `aMethod`:

```
/**
  * @preconditions <pre>
  * @postconditions <post>
  */
public void aMethod(...) {
  ...
  APIClass.apiMethod(...);
  ...
}
```

Our method invokes a method from the Java Card API, which we assume has been already specified. The specification of `aMethod` and its implementation is translated into a proof obligation, which in turn is passed to the KeY deduction component (prover). When trying to construct a proof for this proof obligation, we sooner or later have to apply a rule that takes care of the invocation of the

API method `apiMethod`. If we had no specification of this method we would have to replace the method call with the actual method body. In case the specification of `apiMethod` is available it is enough to verify that the precondition of `apiMethod` is satisfied in the state before `apiMethod` is executed and then we can simply "replace" the method call to `apiMethod` with its postcondition. This however is not as straightforward as it sounds, there is ongoing work in the KeY project which investigates when and under what conditions such a replacement can be safely done [BS03].

In addition to this, having an OCL specification of the API saves a lot of work during verification of JAVA CARD programs in the long run. When there is no specification available, the same API method call has to be replaced by the method's implementation and proved each time the method in question is used. In practice it can happen that the same piece of API implementation is going to be placed in the proof more than one time in one program.

The secondary purpose of writing the OCL specification for JAVA CARD API is for documentation purposes – an OCL specification can serve as formal documentation of the JAVA CARD API. This is very useful, because the informal specification does not always contain all the necessary information about the behaviour of the API.

2.4 Related Work

As already mentioned, the starting point for this work was the formal specification of the JAVA CARD API written in JAVA Modelling Language (JML) [LBR99, MP01, PvdBJ00]. That work has been done for similar reasons as stated above, the main difference is the specification language used. The LOOP tool presented in [vdBJ01] uses JML and PVS as the means to formally verify JAVA CARD programs, thus, the necessity for the API specification written in JML. As we use the industry standard OCL as a specification language in the KeY project we need to have the JAVA CARD API specification formulated in OCL. We also made an effort to have more complete coverage of the JAVA CARD API in our specification.

3 The Development of OCL Specification

As stated above, we based our specification on the JML specification of the JAVA CARD API. We then extended it based on the informal specification (API documentation) and we tried to make use of OCL's expressiveness wherever possible. Later on we tested parts of our specification by formally verifying (using the KeY system) part of the reference implementation of the JAVA CARD API with respect to our specification.

We start by giving an overall description of JML and the JML specification of the JAVA CARD API. Based on that we will describe the main problems to be tackled when writing OCL specification for the API.

3.1 JML vs. OCL

As in OCL, the specifications in JML are expressed as class invariants and method pre-/postconditions. Class invariants are assertions that should hold for all instances of the class at any time. Pre- and postconditions are contracts between the provider and the user of the method. The user has to fulfil the precondition when he or she calls the method. The provider guarantees that if the precondition holds at the beginning of the method call, then the corresponding postcondition will hold after the method call. In addition, JML allows one to express when a method throws an exception and which attributes of the class can be modified by the method. All the JML specifications are only valid in the context of their JAVA source code and are presented in the form of JAVA comments. Below is the general syntax of JML used to express the method's behaviour:

```
/**
 @public behavior
   @requires <precondition>;
   @assignable <list of attributes>;
   @ensures <postcondition>;
   @signals (Exception_1 e1) <ex1postcondition>;
   @signals (Exception_2 e2) <ex2postcondition>;
 */
public void aMethod() throws Exception { ... }
```

The `@requires` clause defines the method's precondition, the `@assignable` clause tells which attributes the method can modify. The meaning of the rest of the specification is the following: if the precondition is satisfied then either the method terminates normally (i.e., does not throw any exception) and the postcondition (`@ensures`) holds or one of the listed exceptions is thrown and then the corresponding postcondition holds.

JML also allows to use a simpler syntax in case the method is not supposed to throw any exceptions, as the example below shows. The example gives a general impression of what the JAVA CARD API specification in JML looks like. The following is a part of the `OwnerPIN` class:

```
public class OwnerPIN implements PIN {
 private byte[] pin;
 private byte maxTries;
 private byte triesRemaining;

 public boolean check(byte[] thePin, short offset, byte length)
   throws ArrayIndexOutOfBoundsException, NullPointerException {
   ...
 }
 ...
}
```

The `pin` array contains the PIN number, `maxTries` is the maximal number of attempts allowed to present the correct PIN before the card is locked, and

`triesRemaining` the number of attempts left to present the correct PIN. A JML invariant for this class is the following:

```
/**
 @invariant triesRemaining >= 0 && triesRemaining <= maxTries;
 */
```

A JML specification of the method `check` is given below. The `arrayCompare` method compares `length` elements of array `this.pin` starting at element indexed by 0 with `length` elements of array `thePin` starting at element indexed by `offset`:

```
/**
 @public normal_behavior
   @requires triesRemaining > 0 &&
   @   Util.arrayCompare(this.pin, (short)0,
   @                     thePin, offset, length) == 0;
   @ensures result == true && triesRemaining == maxTries;
*/
```

At this point we are ready to define the main differences between JML and OCL that caused us some problems when writing the Java Card API specification in OCL. The KeY system provides extensions to OCL to overcome most of those problems.

3.2 Exceptions

The current version of OCL in its standard form does not provide a straightforward way to specify that an exception is thrown by a method. A possible solution is to have an association link `thrownException` in our class, which represents a possible exception thrown by methods of that class. Then it is possible to specify that a method `aMethod` of class `MyClass` throws an exception of type `MyException` this way:

```
context MyClass::aMethod():
  pre:  true
  post: let e : self.thrownException in
          not e.oclIsUndefined() and e.oclIsKindOf(MyException)
  and e.oclIsNew()
```

The KeY system has a unified solution for this – one can use an `excThrown(MyException)` clause in the postcondition, which has a very similar meaning. Later on, when the OCL specification is transformed to a Java Dynamic Logic proof obligation for the prover, the `excThrown` clauses are properly translated to corresponding Java Dynamic Logic formulas.

Having that, we can now give the general representation of JML's `@behavior` clause in OCL:

```
context MyClass::aMethod()
pre:  <precondition>
```

```
post: (not excThrown(java::lang::Exception)
       and <postcondition>)
   or (excThrown(Exception_1) and <ex1postcondition>)
   or  ...
   or (excThrown(Exception_n) and <exnpostcondition>)
```

3.3 The null value

Another thing that is commonly used in Java, but which is not supported in the current version of OCL is the `null` value. This can be handled in OCL in two ways:

- When one wants to compare a class attribute to a `null` value, then it is possible to treat the attribute as an association end, which in OCL can be treated as a set. In that case one can simply say `attr->isEmpty()` to express the fact that `attr` has a `null` value.

- When comparing objects other than class attributes (for example, method arguments) to the `null` value things are a bit more difficult. If such an object is an array or a collection type, one can use the same technique as described above. Otherwise there is no way to specify that an object should (or should not) have the `null` value.

Fortunately, the KeY system provides a workaround for this problem as well. One can use the `null` value directly as if it were defined in OCL, and then during the translation to Java Dynamic Logic the `null` values are handled appropriately.

3.4 Integer Arithmetics

The main data types that Java Card programs deal with are Java `shorts`, `bytes` and arrays. Arrays do not cause much of a problem, in OCL they can be represented as the `Sequence` type. The Java arithmetic types `short` and `byte` however do not have a corresponding type in OCL. The only integer type in OCL is `Integer`. The most important aspect of Java `shorts` and `bytes` is that they can overflow (i.e., they are finite types), while the OCL `Integer` is an infinite type and never overflows. Since the overflow behaviour is a very important aspect of Java programs, we have to be able to distinguish between different integer types in OCL. For this purpose we used dummy "wrapper" classes `JByte` and `JShort` to represent corresponding Java types. They can be used like this:

```
context PIN::check(pin: Sequence(JByte), offset: JShort,
                   length: JByte): Boolean
...
```

This still does not solve the problem of proper interpretation of overflow behaviour in OCL. Luckily, the KeY system comes to the rescue again. When

the OCL specification is translated to a JAVA Dynamic Logic formula, the user can choose how the integer types are interpreted by the prover: either as finite JAVA types `short` and `byte`, or as infinite arithmetic types `arithShort` and `arithByte`. In both cases the issue of overflow is treated appropriately. More about handling arithmetics in the KeY system can be found in [BS04]. Also, [Cha03] gives insights into problems associated with integer arithmetics in JML.

3.5 JML assignable clause

As mentioned before, JML offers a possibility to express (with the `@assignable` clause) that a given method is allowed to change a limited set of attributes during its execution. OCL does not offer any mechanism or language construct to specify this in a nice way. One can of course state in the postcondition that the value of a given attribute is not changed by the method by saying:

```
post: self.attr = self.attr@pre
```

This is not a good solution, though. Suppose we have a class with 20 attributes and we want to express the fact that only one attribute is assignable. That means we have to write 19 expressions like the one above for all the remaining attributes. There is ongoing work that aims at solving this problem in the KeY system [BS03]. The work is about how to properly specify attribute modification behaviour and how such specification can be used in proofs. In the current version of our work we left out the parts of the specification corresponding to the `@assignable` clause in JML.

4 The Specification

The present work resulted in an OCL specification for all classes and interfaces of the JAVA CARD API 2.2. This specification expresses, with a few exceptions (some of the `signals` clauses and the `assignable` clauses were not possible to be fully expressed in OCL), as much as the JML specification for JAVA CARD API 2.1.1. In some cases the OCL specification expresses more than the JML specification. In the following we illustrate by example how our OCL specification was created and how it was improved (compared to JML).

First, let us look at the `PIN` interface (which `OwnerPIN` implements). The informal specification of method `check` in the `PIN` interface is the following:

> *`public boolean check(byte[] pin, short offset, byte length)`*
>
> *Compares `pin` against the `PIN` value. If they match and the `PIN` is not blocked, it sets the validated flag and resets the try counter to its maximum. If it does not match, it decrements the try counter and, if the counter has reached zero, blocks the `PIN`. Even if a transaction is in progress, the internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated.*

Notes:

- *If `NullPointerException` or `ArrayIndexOutOfBoundsException` is thrown, the validated flag must be set to `false`, the try counter must be decremented, and the `PIN` blocked if the counter reaches zero.*
- *If `offset` or `length` parameter is negative an `ArrayIndexOutOf-BoundsException` is thrown.*
- *If `offset+length` is greater than `pin.length`, the length of the `pin` array, an `ArrayIndexOutOfBoundsException` is thrown.*
- *If `pin` parameter is `null` a `NullPointerException` is thrown.*

Parameters:

`pin` *the `byte` array containing the `PIN` value being checked*

`offset` *the starting offset in the `pin` array*

`length` *the length of `pin`*

Returns:

`true` *if the `PIN` value matches; `false` otherwise*

Throws:

`ArrayIndexOutOfBoundsException` *if the `check` operation would cause access of data outside array bounds.*

`NullPointerException` *if `pin` is `null`.*

The JML specification for this method found in [Pol] is the following (the `\old` construct corresponds to OCL's `@pre`):

```
/**
 @ public normal_behavior
 @    requires triesRemaining == 0;
 @  assignable \nothing;
 @     ensures result == false;
 @ also
 @ public normal_behavior
 @    requires triesRemaining > 0 && pin != null && offset >= 0
 @             && length >= 0 && offset + length == pin.length &&
 @             Util.arrayCompare(this.pin, (short)0, pin,
 @                              offset, length) == 0;
 @  assignable isValidated, triesRemaining;
 @     ensures result == true && isValidated &&
 @             triesRemaining == maxTries;
 @ also
 @ public behavior
 @    requires triesRemaining > 0 && !(pin != null &&
 @             offset >= 0 && length >= 0 &&
 @             offset+length == pin.length &&
```

```
 @             Util.arrayCompare(this.pin, (short)0, pin,
 @                              offset, length) == 0);
 @  assignable isValidated, triesRemaining;
 @     ensures result == false &&
 @             !isValidated && triesRemaining ==
 @                 \old(triesRemaining) - 1;
 @     signals (NullPointerException)
 @             !isValidated &&
 @             triesRemaining == \old(triesRemaining) - 1;
 @     signals (ArrayIndexOutOfBoundsException)
 @             !isValidated &&
 @             triesRemaining == \old(triesRemaining) - 1;
 @
 */
public boolean check(byte[] pin, short offset, byte length)
   throws ArrayIndexOutOfBoundsException, NullPointerException;
```

It seems that the JML specification agrees with the informal specification in most part. One subject that is not touched upon in the JML specification is the following sentence from the informal specification: *Even if a transaction is in progress, the internal state such as the try counter, the validated flag and the blocking state must not be conditionally updated.* This is not possible to specify in either JML or OCL, as it has to do with the internal transaction mechanism of the Java Card Runtime Environment. The issue of specifying and verifying the programs involving Java Card's transaction mechanism has been investigated thoroughly in the KeY project [BM03]. For now, however, we decided to leave this issue out in our OCL specification. Another thing to notice is that the informal specification and the JML specification disagree on the subject of whether `offset + length` must be equal to `pin.length` or if `offset + length` might be less than or equal to `pin.length`. It seems that a mistake has been made in the JML specification, since it clearly disagrees with the informal specification and since there seems to be no good reason to demand that there must be no free elements in the `pin` array following the actual PIN value. Therefore our resulting OCL specification agrees with the informal specification in this case:

```
context PIN::check(pin : Sequence(JByte), offset : JShort,
                   length : JByte): Boolean
pre:  true
post: self.triesRemaining = 0 implies result = false
  and (self.triesRemaining > 0 and pin <> null
         and offset >= 0 and length >= 0 and
         offset+length <= pin->size()
         and self.pin->subSequence(1, length) =
             pin->subSequence(offset + 1, offset + length))
      implies (
         result = true and self.isValidated
         and self.triesRemaining = self.maxTries)
  and (self.triesRemaining > 0 and
```

```
        not(pin <> null and offset >= 0 and length >= 0
            and offset + length <= pin->size() and
            self.pin->subSequence(1, length) =
            pin->subSequence(offset + 1, offset + length)))
      implies (not self.isValidated and
        self.triesRemaining = self.triesRemaining@pre-1 and (
          (not excThrown(java::lang::Exception) and
             result = false)
           or excThrown(java::lang::NullPointerException)
           or excThrown(java::lang::ArrayIndexOutOfBoundsException)))
```

In the next example we show how the specification of method `setKey` in class `DESKey` has been enriched compared to JML specification. The method `setKey` copies the data (an array of bytes) that is passed as an argument and constitutes the actual key to the internal attribute `data`. Under certain circumstances, this data is not passed to the method in plain text but as a cipher and the method must then decrypt the data before it is copied into the internal representation. Here is the JML specification for this method:

```
/**
 @public behavior
 @    requires keyData != null && kOff >= 0 &&
 @             kOff < keyData.length;
 @  assignable CryptoException.systemInstance.reason;
 @     ensures isInitialized();
 @     signals (CryptoException e)
 @       e.getReason() == CryptoException.ILLEGAL_VALUE;
 */
void setKey(byte[] keyData, short kOff) throws CryptoException;
```

This specification does not give much information about what this method actually accomplishes. In the OCL specification though, we try to give an idea about this:

```
context DESKey::setKey(keyData : Sequence(JByte), kOff : JShort)
pre:  not (keyData = null) and kOff >= 0 and
      kOff < keyData->size()
post: (not excThrown(java::lang::Exception)
       and self.isInitialized() and (
          not self.oclIsKindOf(javacardx::crypto::KeyEncryption)
       or self.getKeyCipher() = null implies
            self.data->subSequence(1, self.getSize() / 8) =
            keyData->subSequence(kOff + 1, kOff + self.getSize() / 8))
      ) or (
       excThrown(javacard::security::CryptoException) and
       CryptoException.systemInstance.reason
          = CryptoException.ILLEGAL_VALUE
       and (
         not self.oclIsKindOf(javacardx::crypto::KeyEncryption)
         or self.getKeyCipher() = null implies
             kOff + self.getSize() / 8 > keyData->size()))
```

What we added in this specification is the following. If this particular instance of DESKey is not an instance of javacardx.crypto.KeyEncryption or if this instance is not associated with a Cipher object (the circumstances under which the input keyData have to be decrypted), then the input data is to be copied directly into the internal attribute data.

While studying the JML specification we found a small number of minor inconsistencies. In the class OwnerPIN for example, the invariant states that the internal class attribute pin should not be null at any point, which requires the constructor of that class to set pin (which is initially null) to a non null value. In that case the constructor should be able to modify the pin attribute, but a corresponding @assignable clause is missing in the specification of the OwnerPIN constructor. The informal specification of that constructor also says that two exceptions can be thrown – PINException and SystemException. The condition for throwing the PINException is clearly defined, but this information is not included in the constructor's specification.

We tried to fix all those small deficiencies in our OCL specification and express as much as possible, but, as we mentioned before, giving the full specification of the JAVA CARD API in OCL is not possible at the moment.

4.1 Formal Verification

To give our specification a test we looked into the source of the implementation of the JAVA CARD API distributed with SUN's JAVA CARD Development Kit version 2.1.1 [Sun]. We tried to verify this implementation with respect to the specification we have written. Due to current limitations of the KeY system this was not done to the extent one might wish for. One of the technical reasons for this is the fact that the KeY system does not handle arrays in the version we used. Since the arrays are present almost everywhere in the JAVA CARD API this was a major obstacle. We can however report that a number of simple getReason/setReason methods in the exception classes of javacard.framework package have been verified. A more complicated successful proof attempt was the verification of the reset method in the OwnerPIN class. The specification is the following:

```
context OwnerPIN inv:
  self.maxPINSize > 0 and self.maxTries > 0 and
  self.triesRemaining >= 0 and
  self.triesRemaining <= self.maxTries

context OwnerPIN::reset()
pre:  true
post: not excThrown(java::lang::Exception)
      and
      not self.isValidated
      and
      if self.isValidated@pre then
        self.triesRemaining = self.maxTries
```

```
else
  self.triesRemaining = self.triesRemaining@pre
endif
```

A proof obligation generated by the KeY system states the following: the execution of the `reset` method preserves the invariant and if the precondition is satisfied before `reset` is executed then the postcondition is satisfied after `reset` is executed. Explaining what this proof obligation looks like would require introducing the Java Dynamic Logic used in the KeY system in more detail. This would go beyond the scope of this paper. One thing we should say though, is that the proof to verify this specification is performed automatically by the KeY prover, reducing the user interaction to absolute minimum.

5 Short Evaluation of OCL

There are a few things that we found very useful about OCL. First of all, it is practically an industry standard and is (partially) supported by some CASE tools (for example, Borland Together Control Center that we use in the KeY project). Second, it seems that the OCL language is richer than JML in some respects, for example, the whole library of collection type operations makes expressing properties about `Sequence` (array) type much easier than in JML. Also, for the same reason, we find OCL much easier to read and understand.

When it comes to Java specific features, OCL turns out to be not as good as JML. Just to recapitulate the most important findings from Section 3.1: there is no standard way in OCL to express the fact that a method throws an exception, there is only one (infinite) integer type in OCL as compared to the whole set of Java integer types and there is no JML's `@assignable` counterpart in OCL. In this respect JML is a much stronger language than OCL. Of course, this is because JML was designed specifically for Java, while OCL was mainly designed for UML.

6 Conclusions

In this paper we presented our experience from the development of an OCL specification for the Java Card API 2.2. Despite the mentioned problems with OCL we managed to specify the whole Java Card API to a reasonable extent. The specification is available on-line at:

`http://www.key-project.org/doc/2003/exjob.html`

The two main purposes of this work were to aid and support formal verification of Java Card programs in the KeY system and to document the Java Card API in a formal way. We tested our specification by formally verifying the reference implementation of the Java Card API with the KeY system, however, due to technical limitations, this was not done to the desirable extent. Still, the proofs we attempted were successful and were performed automatically by the

KeY system. In the near future the KeY system will cover the full JAVA CARD standard. Then we plan to continue in this direction and also, based on our specification, perform formal verification of real life JAVA CARD case studies.

References

[ABB+04] 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.

[Bec01] 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.

[BM03] 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 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.

[BS03] 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.

[BS04] 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.

[Cha03] Patrice Chalin. Improving JML: For a safer and more effective language. In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli, editors, *International Symposium of Formal Methods Europe, Proceedings*, volume 2805 of *LNCS*, pages 440–461, Pisa, Italy, September 2003. Springer.

[Che00] Zhiqun Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, 2000.

[Lar03] Daniel Larsson. OCL specifications for the JAVA CARD API. Master's thesis, Chalmers University of Technology, Department of Computing Science, Göteborg, Sweden, 2003.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design.* Kluwer Academic Publishers, 1999.

[MP01] Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card API. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*, volume 2140 of *LNCS*, pages 165–178. Springer, September 2001.

[Obj03] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[Pol] Erik Poll. Formal interface Java specifications for the Java Card API 2.1.1. `http://www.cs.kun.nl/~erikpoll/publications/jc211_specs.html`.

[PvdBJ00] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the Java Card API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Academic Publishers, 2000.

[Sun] Sun Java Card developement kit 2.1.1. `http://java.sun.com/products/javacard/dev_kit.html#211`.

[Sun02] Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface*, 2002. `http://java.sun.com/products/javacard/specs.html`.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language, Second Edition: Getting Your Models Ready for MDA.* Object Technology Series. Addison-Wesley, Reading/MA, 2003.

Paper IV Verification of Safety Properties in the Presence of Transactions

Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop, Marseille, France, March 2004

Verification of Safety Properties in the Presence of Transactions

Reiner Hähnle* Wojciech Mostowski

Abstract

The JAVA CARD transaction mechanism can ensure that a sequence of statements either is executed to completion or is not executed at all. Transactions make verification of JAVA CARD programs considerably more difficult, because they cannot be formalised in a logic based on pre- and postconditions. The KeY system includes an interactive theorem prover for JAVA CARD source code that models the full JAVA CARD standard including transactions. Based on a case study of realistic size we show the practical difficulties encountered during verification of safety properties. We provide an assessment of current JAVA CARD source code verification, and we make concrete suggestions towards overcoming the difficulties by *design for verification*. The main conclusion is that largely automatic verification of realistic JAVA CARD software is possible provided that it is designed with verification in mind from the start.

1 Introduction

As JAVA CARD technology is picking up speed it becomes more and more interesting to employ formal analysis techniques in order to ensure that JAVA CARD applications work as intended. Formal approaches to JAVA CARD application development encompass a wide spectrum from byte code to source code, from fully automated to highly interactive, and from abstract to fully concrete semantics (see Section 5 for a brief overview).

Our work is aimed at JAVA CARD source code verification with full modelling of all semantic aspects. This includes the JAVA CARD transaction mechanism that ensures a sequence of statements either being executed to completion or not being executed at all. The underlying technology, described in Section 2.2, is theorem proving in an expressive logic, in which programs and their requirements are formalised. Fully automatic inference in this context is in general unachievable, but one goal of the presented work was to find out just how far automation reaches.

The experiments described in this paper were made with the KeY theorem prover, which is an interactive verification system for JAVA CARD featuring a

* Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, e-mail: reiner@cs.chalmers.se

complete formalisation of atomic transactions [BM03]. It is part of the KeY system [ABB+04], an integrated tool for informal and formal development of object-oriented software described in Section 2.1. This paper makes the following contributions:

- An experience report about the verification of parts of a Java Card electronic purse application (*Demoney*) of realistic complexity [MM02]. The code includes atomic transactions. To our best knowledge, this is the first report on verification of Java Card source programs with transactions without any simplification or abstraction. The case study and the experiments are described in Section 3.
- An assessment of current source code verification technology: what can be automatically proven in terms of LoC, complexity, etc.? Which desirable requirements can be expressed and which not? This is discussed in Section 4.1.
- An analysis of the limitations of current technology and how they can be overcome. We explain why the *Demoney* case study had to be partially refactored to make verification feasible. In particular, we make concrete suggestions towards overcoming the difficulties by *design for verification* in Section 4.2.

The main conclusion we draw in this paper is that largely automatic verification of realistic Java Card software is in the realm of the possible, but it is essential to move from *post hoc* verification to a more aggressive approach, where software is designed with verification in mind from the start.

2 Background

2.1 The KeY Project

The work presented in this paper is part of the KeY project[1] [ABB+04]. 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, the KeY Prover, has been developed. 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 including the full Java Card language specification. 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. The KeY system translates OCL specifications into Java Card DL formulae, whose validity can then be proved with the KeY Prover. All this is

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

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. As we see later, this is even relatively straightforward.

2.2 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 [Pra77, HKT00] 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). 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 [Bec01]. 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 [BGH+04], 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 [BS04].

To sum up the description of Java Card DL and to give the reader an impression of concrete Java Card DL formulae, we present a simple Java Card DL proof obligation:

$$\texttt{card.balance} \doteq b \;\vdash\; \langle \texttt{card.charge(amount);} \rangle\, \texttt{card.balance} \doteq b + \texttt{amount}$$

It says that if the `card` object's `balance` attribute is equal to b in the initial state, then the execution of method `charge` with argument `amount` terminates normally (no exception thrown) and afterwards the `card` object's initial `balance`

is increased by `amount`. The validity of this proof obligation under Java integer semantics depends on whether `charge()` accounts for overflow, the type of the + operator, etc.

2.3 Strong Invariants

While working on one of the Java Card case studies [Mos02] it became apparent that the specification semantics based on the initial and final states of a program is not enough to specify and verify some Java Card safety properties. It turned out that the Java Card applet in question was not "rip-out safe": it is possible to destroy the applet's functionality by removing (ripping out) the Java Card device from the card reader (terminal) while the applet on the card executes. As a result of this the applet's memory may become corrupted and left in an undefined state, causing malfunctioning of the applet.

To avoid such errors one has to be able to specify and verify the property that a certain invariant on the objects' data is maintained at any time during applet execution and, in particular, in case of abrupt termination. Usually, class invariants (in OCL and elsewhere) are interpreted with respect to pre/post state semantics, that is, if the invariant holds before a method is executed then it holds again after the execution of a method. This semantics does not suffice to ensure properties of data in intermediate states during method's execution. To solve this problem, we introduced *strong* invariants, which allow to specify properties about all intermediate states of a program.[2]

For example, the following strong invariant (expressed in pseudo OCL) says that we do not allow partially initialised `PersonalData` objects at any point in our program. In case the program is abruptly terminated we should end up with either a fully initialised object or an uninitialised (empty) one:

```
context PersonalData throughout:
  not self.empty implies
    self.firstName <> null and self.lastName <> null and self.age > 0
```

To introduce the notion of a strong invariant it was necessary to extend 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. And such properties (typically strong invariants and safety constraints) can be verified using the Java Card DL calculus extended with additional sequent rules for the "throughout" modality [BM03].

[2] In extended static checking a closely related concept called *object invariants* is used [LS97]. The semantics of OCL invariants is interpreted in the strong sense in [ZG03], where a temporal extension of OCL is introduced.

2.4 Java Card Atomic Transactions

There is one particular aspect of Java Card that makes the "throughout" extension considerably more complicated than expected, namely, the Java Card transaction mechanism. The transaction mechanism allows a programmer to enforce atomicity of sequences of Java Card statements. It is typically used to ensure consistency of related data that have to be updated simultaneously.

The memory model of Java Card differs somewhat from Java's memory model [Che00, Sun03]. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which is preserved between card sessions, and transient memory (RAM), whose contents disappears when power loss occurs, for example, when the card is removed from the reader. Hence, every memory location in Java Card (variable or object field) is either persistent or transient. The Java Card language specification gives the following rules (slightly simplified for this presentation): all objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Therefore, in Java Card assignments such as "`o.attr = 2;`", "`this.a = 3;`", and "`arr[i] = 4;`" all have a permanent character; that is, the assigned values will be kept after the card loses power. A programmer can create an array with transient elements, but currently there is no possibility to make objects (fields) other than array elements transient. All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by Java Card's transaction mechanism. The following are the Java Card system calls for transactions with their description:

`JCSystem.beginTransaction()` begins an atomic transaction. From this point onwards, until the transaction finishes, all assignments to fields of objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally (immediately).

`JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).

`JCSystem.abortTransaction()` aborts the transaction. All the conditional assignments are rolled back to the state in which the transaction started. Assignments to transient variables and array elements remain unchanged (as if there were no transaction in progress).

A "throughout" property (formula) has to be checked after every single field or variable assignment which, according to the Java Card runtime environment specification [Sun03], is atomic. Such checks have to be suspended, however, when a transaction is in progress, because the assignments inside a transaction are not atomic, only the whole transaction is atomic. Moreover, as already said, each transaction can either finish successfully, in which case it commits all the conditional assignments, or it can fail and in that case the transaction is aborted and all the conditional assignments have to be rolled back. The logic

has to account for the possibility of an abort and for the difference between persistent and transient data.

Observe that the possibility of an aborted transaction affects even the semantics of the standard modal operators $\langle\cdot\rangle$ and $[\cdot]$, because an abort affects the final state of the program. Details of how the extension of Java Card DL that deals with transactions is handled in the calculus can be found in [BM03]. We do not repeat the technical solution in this paper, but we stress that the details are rather involved and surprisingly complex. The KeY Prover implements the whole extension of Java Card DL with "throughout" and transaction mechanism. To our knowledge the KeY Prover is the only prover for Java Card programs that fully handles Java Card transactions.

When a strong invariant has been specified for a Java Card program, say, for a class C, each of C's methods can be a subject to verification with respect to the strong invariant. A typical proof obligation for a method `m()` involving a strong invariant looks as follows:

$$(\mathit{Inv} \wedge \mathit{Pre} \wedge \mathit{StrongInv}) \rightarrow [\![C::\texttt{m();}]\!]\ \mathit{StrongInv}$$

Inv stands for a standard (weak) invariant of class C and *Pre* stands for the method's precondition. Apart from those two premises one also has to assume that the strong invariant *StrongInv* holds before method `m()` is executed to establish that *StrongInv* holds throughout the execution of `m()`.

3 Case Study: Java Card Electronic Purse

The case study presented here is based on the Java Card electronic purse application *Demoney* [MM02]. While *Demoney* has not all 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.

Our target program is a somewhat refactored fragment of *Demoney* and concentrates on the important aspects of the application to highlight our verification results. The *Demoney* source code is at present not publicly available, and we do not show it. The program we verified is, however, very close to *Demoney* and follows the *Demoney* specification [MM02]. We deviate from *Demoney* mainly in that our program is designed to make verification simpler. We discuss these issues in detail in Section 4.2.

The safety properties that we discuss here were directly motivated by the ones described in [MM01]. In fact the property we prove (that the current balance of the purse is always in sync with the balance recorded in the most recent log entry) for the `processSale` method presented in Section 3.4 is exactly the one described in [MM01, Section 3.5]. The example mentioned there is also based on the *Demoney* application.

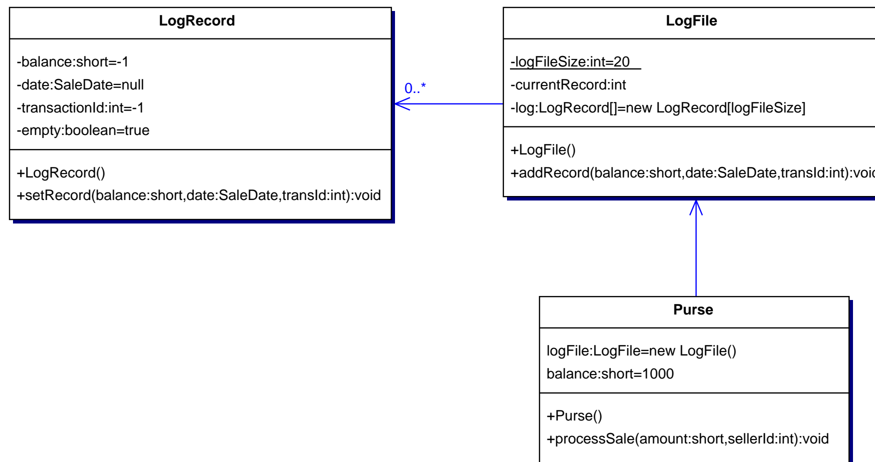


Figure 1. `Purse` application class diagram

3.1 The LogRecord Class

The UML class diagram of our program is shown in Figure 1. The basic class is `LogRecord` which is used to store data about a single purse transaction. The data consists of the new balance after the transaction (`balance:short`), transaction identifier (`transactionId:int`) and transaction date (`date:SaleDate`). Additionally, the attribute `empty` states if a particular instance of `LogRecord` is in use.

Such an attribute is characteristic for the Java Card platform, which is a memory constrained device and in general does not possess a garbage collector. To avoid memory overflow during execution all objects are allocated during the initialisation phase of Java Card applets and the programmer keeps track of which objects are already in use, for example by introducing attributes like `empty`.[3] The `LogRecord` class contains only one method, which is responsible for assigning values to its attributes:

```
public void setRecord(short balance, SaleDate date, int transId) {
  this.balance = balance;
  this.date = date;
  this.transactionId = transId;
  this.empty = false;
}
```

[3] Some design and implementation choices in our example may seem artificial (for example, the value of `empty` never changes from `false` to `true`), but the point was to illustrate certain critical issues.

3.2 Specification and Verification of setRecord

Regarding data consistency, the main property one needs to establish about the class `LogRecord` is to assure that at any point all the instances of this class that are in use are properly initialised. Expressed in (pseudo) OCL this property reads:

```
context LogRecord throughout:
  not self.empty implies
    self.balance >= 0 and self.transactionId > 0 and self.date <> null
```

This states that all attributes of `LogRecord` objects that are in use have proper values at any point in time. We want to prove that the method `setRecord` preserves this strong invariant. In order to do this, one needs a precondition saying that the parameters that are passed to `setRecord` have proper values. The resulting Java Card DL proof obligation in the actual notation used by the KeY Prover is:

```
  !self = null
& balance >= 0 & !date = null & transId > 0
& (self.empty = FALSE ->
    (self.balance >= 0 & !self.date = null & self.transactionId > 0))
-> [[{ self.setRecord(balance, date, transId); }]]
  (self.empty = FALSE ->
    (self.balance >= 0 & !self.date = null & self.transactionId > 0))
```

This is proved automatically with 230 rule applications in 2 seconds.[4] If we change the strong invariant into a weak invariant, that is, replace the throughout modality in the formula above with a diamond modality, the resulting proof obligation is (as expected) also provable (125 rules, less than 2 seconds).

Observe that the order of attribute assignments in `setRecord`'s body is crucial for the strong invariant to hold. If we change `setRecord`'s implementation to

```
public void setRecord(short balance, SaleDate date, int transId) {
  this.empty = false;
  this.balance = balance;
  this.date = date;
  this.transactionId = transId;
}
```

then it does not preserve the strong invariant anymore, while it still preserves the weak invariant. When trying to prove the strong invariant for this implementation the prover stops after 248 rule applications with 6 open proof goals. The proof for the weak invariant proceeds in the same fashion as for the previous implementation.

[4] All the benchmarks presented in this paper were run on a Pentium IV 2.6GHz Linux system with 512MB of memory. The version of the KeY Prover used (0.1200) is available on request. The prover was run with Java 1.4.2.

3.3 The Purse Class

The `Purse` class is the top level class in our design. The `Purse` stores a cyclic file of log records (each new entry allocates an unused entry object or overwrites the oldest one), which is represented in a class `LogFile`. `LogFile` allocates an array of `LogRecord` objects, keeps track of the most recent entry to the log and provides a method to add new records – `addRecord`.

The `Purse` class provides only one method – `processSale`. It is responsible for processing a single sale performed with the purse – debiting the purchase amount from the balance of the purse and recording the sale in the log file. To ensure consistency of all modified data, Java Card transaction statements are used in `processSale`'s body. Figure 2 shows the UML sequence diagram of `processSale`. The total amount of code invoked by `processSale` amounts to less than 30 lines, however, it consists of nested method calls to 5 different classes.

3.4 Specification and Verification of processSale

As stipulated in [MM01], we need to ensure consistency of related data. In our case, this means to express that the state of the log file is always consistent with the current state of the purse. More precisely, we state that the current balance of the purse is always equal to the balance stored in the most recent entry in the log file. The corresponding strong invariant expressed in pseudo OCL is:

context Purse **throughout**:
  self.logFile.log.get(self.logFile.currentRecord).balance = **self**.balance

Since `processSale` is the method that modifies both the log file and the state of the purse, we have to show that it preserves this strong invariant. The most important part of the resulting proof obligation expressed in Java Card DL is the following:

```
  JCSystem.transactionDepth = 0
& !self = null
& !self.logFile = null
& !self.logFile.log = null
& self.logFile.currentRecord >= 0
& self.logFile.currentRecord < self.logFile.log.length
& self.logFile.log[self.logFile.currentRecord].balance = self.balance
-> [[{ self.processSale(amount, sellerId); }]]
  self.logFile.log[self.logFile.currentRecord].balance = self.balance
```

This proof obligation is proved automatically by the KeY Prover modelling the full Java Card standard (see Section 3.6) in less than 2 minutes (7264 proof steps).

3.5 *Post Hoc* Verification of Unaltered Code

We just reported on successful verification attempts of a refactored and partial version of the *Demoney* purse application. When it comes to capabilities

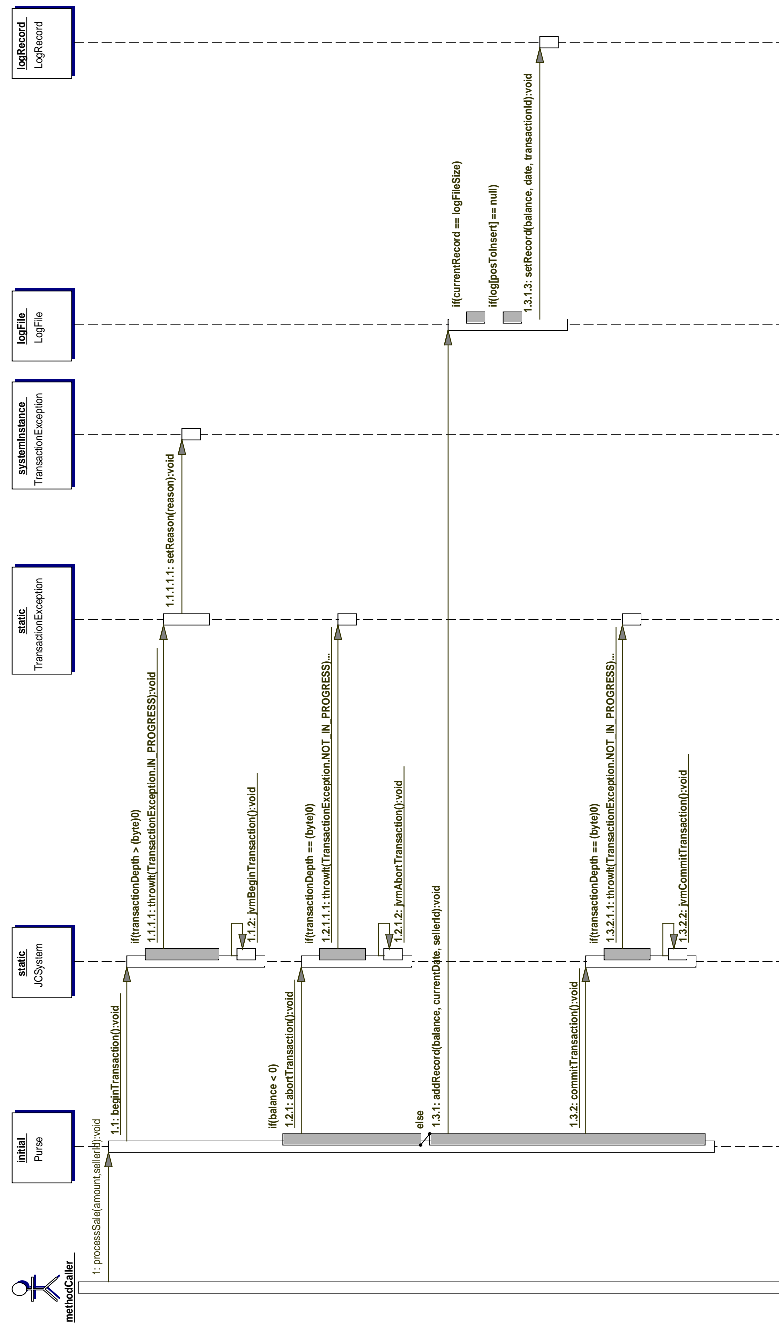


Figure 2. Sequence diagram of the `processSale` method

and theoretical features of the KeY Prover there is nothing that prevents us in principle from proving properties about the real *Demoney* application. There are, however, some design features in *Demoney* that make the verification task difficult. We discuss them in detail in Section 4.2.

We also proved total correctness proof obligations for two simple, but *completely unaltered*, methods of *Demoney* called `keyNum2tag` and `keyNum2keySet`. This was possible, because the problems discussed in Section 4.2 below stayed manageable in these relatively small examples. It was crucial that the KeY Prover allows to prove properties of *unaltered* Java code. This implies that, in principle, Java code does not have to be prepared, translated, or simplified in any way before it can be processed by the prover. Unaltered Java source programs are first-class citizens in Dynamic Logic. Java Card DL formulae simply contain references to source code locations such as this:

```
fr.trustedlogic.demo.demoney.Demoney self;
byte keyNum;
byte result;
...
result = self.keyNum2tag(keyNum);
```

As the source code we proved properties about was given beforehand, what we did can be called *post hoc* verification.

3.6 Performance

We emphasise that all mentioned proofs were achieved fully automatically. What it means for the user is that there is no interaction required during the proof and, as a consequence, the user does not have to understand the workings of the Java Card DL calculus.

Table 1 summarises proof statistics relating to the examples discussed previously. Some explanations about the three different versions of the proof for `processSale` are due: the KeY Prover allows to use different settings for the rules used during a proof. One of those settings concerns the kind of arithmetics (see Section 2.2). When ideal arithmetic is used, then all integer types are considered to be infinite and, therefore, without overflow. When Java arithmetic is used, the peculiarities of integer types as implemented in Java are taken into account: different range (`byte`, `short`, etc.), finiteness, and cyclic overflow.

Another prover setting is the `null` value check. When switched off, many variables with object references are assumed to be non `null` without bothering to prove this fact. When switched on, the prover establishes the proper value of every object reference. Obviously, proofs involving `null` checks are more expensive. The checks for index out of bounds in arrays are *always* performed by the prover. The benchmark for the third version of `processSale` represents the prover's behaviour with support for the full Java Card standard.

Figure 3 shows a screenshot of the KeY Prover with a successful proof for the third version of `processSale`.

Proof Obligation	Time (sec.)	Steps	Branches
⟦setRecord⟧	2.0	230	20
⟨setRecord⟩	1.5	125	6
⟦setRecord⟧[F]	2.1	248	6 open
⟨keyNum2tag⟩[D]	3.3	392	18
⟨keyNum2keySet⟩[D]	5.5	640	33
⟦processSale⟧[1]	41.4	3453	79
⟦processSale⟧[2]	51.3	4763	248
⟦processSale⟧[3]	111.1	7264	338

[F] Failed proof attempt
[D] Methods from *Demoney* (full pre/post behavioural specification)
[1] Ideal arithmetic, no null pointer checks
[2] Ideal arithmetic, with null pointer checks
[3] JAVA arithmetic, with null pointer checks

Table 1. Performance of KeY Prover for examples discussed in the text

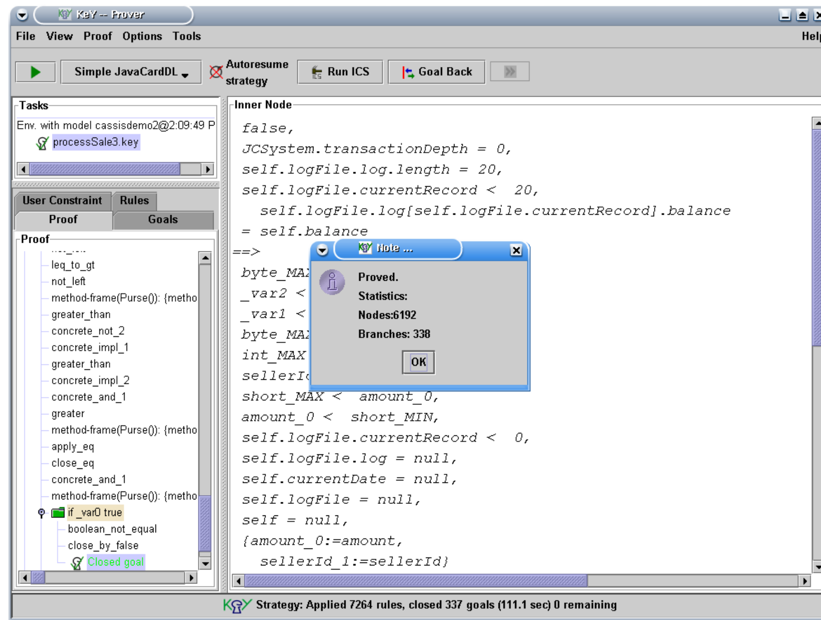


Figure 3. KeY Prover window with successful proof

4 Results

4.1 Verification Technology

Although we so far managed to verify only a small and partly refactored part of *Demoney*, we are encouraged by what we could achieve. The verified programs contain many complex features: nearly every statement can throw an exception, many Java arithmetic and array types occur, there are several nested method calls and, above all, Java Card transactions that may cause subtle errors.

The largest example involves about 30 lines of source code. This may not seem much, but it clearly indicates that methods and classes of non-trivial size can be handled. In addition, the next version of the KeY prover will support composition of proofs including a treatment of representation exposure by computation of modifier sets [BS03]. Consequently, we expect that formal verification of Java Card programs comparable to *Demoney* is achievable before long.

On the other hand, there are also serious limitations. To start with, we observed that verification of the more complex methods of the unaltered *Demoney* program results in specifications and proof obligations that simply become too long and complex. In our opinion, this problem must be attacked by moving from *post hoc* verification to *design for verification*, see the following section.

It would be desirable to have a more formal statement here relating types of programs and proof complexity. The problem is that even loop-free Java Card programs contain control structures like exceptions and transactions that have a global effect on control flow. Taking away all critical features yields an uninteresting programming language, while leaving them in renders general statements on proof complexity (at least the ones we could think of) simply untrue.

A principal obstacle against automating program verification is the necessity to perform induction in order to handle loops (and recursion). In most cases, the induction hypothesis needs to be generalised, which requires considerable user skill. There is extensive work on automating induction proofs, however, mostly for simple functional programming languages as the target. Only recently, preliminary work for imperative target languages [RCK03, HW03] appeared. If, however, *Demoney* is a typical Java Card application, then loops might be much less of a problem than thought: of 10 loops in *Demoney* (9 `for`, 1 `while`) most are used to initialise or traverse arrays of known bounds. Such loops do not require induction at all. The next version of the KeY Prover contains a special automated rule for handling them. Our analysis showed that at most one loop in *Demoney perhaps* needs induction. There is no recursion.

Speed and automated theorem proving support, for example, for arithmetic properties, need to be improved in order to achieve an interactive working mode with the prover, which is not possible with proofs that in some cases take minutes. There is no principal obstacle here; for example, the speed increased by an order of magnitude since we began the case study.

An important question is whether we are able to express all relevant require-

ments. There is no agreement on standard requirements for Java Card, but the report [MM01] can serve as a guideline. Many of the security properties related there can be expressed in Java Card DL including strong invariants. In the present paper we concentrated on data consistency in connection with atomic transactions. The examples included also overflow control. In [DHS04] it was shown that also information flow properties are expressible. We have strong evidence that also memory allocation control, error control and even the well-formedness of transactions can be formulated. For example, the following two properties, taken from [MM01] can be formulated in Java Card DL: (i) no `TransactionException` related to well-formedness is thrown, (ii) only `ISOExceptions` are thrown at the top level of an applet.

The main limitation of the currently used version of Java Card DL is the impossibility to express complex temporal relationships between the execution of different code fragments to establish advanced control flow properties such as a certain temporal order on method calls. This requires more complex temporal operators than "throughout" or some kind of event mechanism, and is a topic for future research. On the specification side, some work was done in [TH02], while [BCW$^+$02] looked at abstracted byte code in a model checking framework.

4.2 Design for Specification and Verification

The way *Demoney* is designed and coded causes certain technical complications both when specifying and proving safety properties of programs with transactions. We demonstrate two issues and discuss their impact on the process of specification and verification of Java Card programs. Thereby, we give guidelines for the design of Java Card applications to avoid such problems.

Byte Arrays. Following the specification in [MM02, p. 17] *Demoney* implements a cyclic log file in a very similar fashion to our `Purse` class. *Demoney* stores more information than our program in a single log record, but that's not an issue when it comes to formal verification. The major difference is that each single log record is implemented as a `byte` array instead of an object (of class `LogRecord` in our case). We suspect that the main reason for implementing a log record as a `byte` array is to ease the transportation of log data to the card terminal. Another reason, explicitly mentioned in the specification, is to follow the schema of recording data in the form of TLVs (Tag-Length-Value). Finally, because of memory costs in smart cards, `byte` arrays are still much used to save some small memory overhead one has to pay for object instances and booleans.[5]

The use of a `byte` array instead of an object type has consequences for the verification process. To start with, Java Card allows only one dimensional arrays, which means that one cannot explicitly declare in a Java Card program that a log file is a two-dimensional array. So, instead of saying

```
byte[][] logFile;
```

[5] The last point was confirmed by Renaud Marlet, Trusted Logic S.A., in personal communication.

one has to say

```
Object[] logFile;
```

and then allocate this data structure by saying:

```
logFile = new Object[logFileSize];
for(short i=0; i<logFile.length; i++)
  logFile[i] = new byte[LOG_RECORD_SIZE];
```

Since this is a dynamic allocation, there is no static information on the type of elements in the `logFile` array. Statically, one can only deduce that those elements are of type `Object`. In the verification process however, such information has to be made more precise. Since it cannot be deduced statically, it has to be included in the assumptions (that is, preconditions) of a proof obligation explicitly. In Java Card DL this requires use of existential quantifiers and lengthy Dynamic Logic expressions. In many cases, existential quantification makes it harder to find a proof automatically. If, instead, one declares a `logFile` as

```
LogRecord[] logFile;
```

the situation is much clearer from the prover's point of view. The only assumption needed in this case is that the elements of the `logFile` array are not `null`. In general it would also require a quantifier (universal), but in the special case of our program we are only interested in two elements of this array, so that the following assumption is sufficient:

```
!logFile[currentRecord] = null &
  !logFile[(currentRecord + 1) % logFileSize] = null
```

This, together with the declaration of `logFile`, is enough for the prover to establish type information about all relevant elements of `logFile`. Moreover, if the `logFile` is statically allocated right after it is declared,

```
LogRecord[] logFile = new LogRecord[20];
```

then no assumptions about the elements of `logFile` are necessary at all. The `logFile` example is not an isolated case, as one can find several occurrences of declarations of `Object` arrays in *Demoney*.

The second issue with the use of `byte` arrays for storing log records is related to arithmetics. The strong invariant for our `Purse` class states:

```
self.logFile.log[self.logFile.currentRecord].balance = self.balance
```

The type of attribute `balance` both in `LogRecord` and in `Purse` is `short`. When the `byte` array is used for storing log record data, then the value of balance is stored in two `byte` elements of this array. Comparing such a two `byte` value stored in an array to a `short` value becomes a bit complicated:

```
self.logFile.log[self.logFile.currentRecord][OFF_BALANCE] =
  castToByte((self.balance - castToByte(self.balance % 256)) / 256) &
self.logFile.log[self.logFile.currentRecord][OFF_BALANCE + 1] =
  castToByte(self.balance % 256)
```

This specification expression is based on an educated guess of how the Java Card API method `Util.setShort` [Sun03] is implemented (`setShort` is a native method and its implementation is not disclosed). Expanding the Dynamic Logic function symbol `castToByte` results in another modulo operation. Also note that all arithmetic function symbols have Java types and must be checked against overflow. Proving with expressions such as the one shown above is difficult, if not practically unfeasible.

We sum up the problems associated to `byte` arrays: (1) typing information is difficult to establish, causing very complicated preconditions, and (2) comparison of `short` values unwrapped into two `byte` values requires the use of complex expressions involving modulo arithmetics. Both problems have serious impact on the size of proofs and automation.

The use of `byte` arrays is partially steered by the TLV standard. We do not argue with the purpose or usability of this standard in smart card technology, and we accept its motivations, such as the performance and space optimisation of Java Card applets. It seems obvious, however, that some things have to be traded off to ease formal specification and verification of Java Card programs.

One general guideline would be to use object types to store any kind of non-primitive data, at least if they are persistent (for transient data there is no choice but an array in Java Card). Furthermore, serialise objects only if necessary (in case of Java Card for communication). As part of a bigger picture one should consider to decouple application functionality from the communication model. Such a decoupled design is likely to allow decomposable, and thus easier, verification. It is more robust, too. We point to the fact that the examples presented in [MM01] follow for the most part the guideline of using object types instead of `byte` arrays for storing data.

Cyclic Indexing of Arrays. Another problematic issue for specification and verification is the way information on the most recent record in the log file is kept and updated in *Demoney*. This is rather a problem of coding conventions and not a design issue. *Demoney*'s cyclic file class has an attribute that stores the index of the next record to be used – `nextRecordIndex`. In order to access the most recent entry in the log, one writes an expression like:

```
logFile[(nextRecordIndex - 1) % logFileSize] ...
```

Modulo arithmetics is used to calculate the actual index. If we add the way the `nextRecordIndex` is updated, that is

```
nextRecordIndex = (nextRecordIndex + 1) % logFileSize;
```

then the prover has to establish the validity of equations such as:

```
index = (((index - 1) % logFileSize) + 1) % logFileSize
```

where all arithmetic function symbols have Java types and must be checked against overflow. This is certainly not impossible, but it adds substantially to the complexity of the resulting first-order proof obligations and, in connection

with other phenomena, can make the problems too difficult to prove automatically.

To avoid these complications, we suggest two simple guidelines. The first is to keep track of those indices that are relevant for specification and verification, instead of those for implementation (or simply keep both kinds of indices). The second is to avoid modulo operations, if possible. The update of `nextRecordIndex` can be easily rewritten as:

```
nextRecordIndex++;
if (nextRecordIndex == logFileSize)
   nextRecordIndex = 0;
```

This program fragment might not be as simple and fast as the one before, but it considerably eases verification.

We believe that if the problems mentioned in this section were not present we would be able to verify automatically that *Demoney*'s `performTransaction` method preserves the kind of strong invariant that we had in our `Purse` class.

Discussion. Asking a programmer to rewrite the code to ease verification may seem unrealistic. It may look as if we put the burden of making verification feasible on the programmer instead of enabling the prover handle arbitrarily complex programs. This is not the case. Our aim is to make the KeY prover powerful enough to deal with complex Java Card code, however, one cannot expect a prover to deal with baroque programs optimised for performance. A trade-off has to be found. The guidelines we proposed are simple to follow and, in addition, make sense from a software engineering point of view. In particular, we do not assume that the programmer has any knowledge of the theorem prover.

Another counter argument against rewriting the code is that abstraction and interface specification should be used to simplify the verification process and get around some of the problems we described above. We fully agree with this, where this possibility is applicable, but in the context of Java Card applet verification it is not so. For example, when one proves a rip-out related property, one cannot abstract away from the implementation of the API methods, because the actual implementation of an API method affects the intermediate states of the program being verified.

5 Related Work

A version of Dynamic Logic that extends pure Dynamic Logic with trace modalities “throughout” and “at least once” was first presented in [BS01]. The axiomatisation of transactions was provided in [BM03]. Paper [HP04] proposes another approach to reasoning about rip-out properties (called card tears there). It presents a theoretical framework for dealing with card tears and transactions based on global program (method) transformation (as opposed to the KeY approach of local transformations). This paper does not report on any practical

verification attempts. In [TH02] temporal constructs are introduced to the JAVA Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Paper [JMR04] is closely related to our work in the sense that it reports on successful verification attempts of a commercial JAVA CARD applet with different verification tools (ESC/JAVA2, JIVE, KRAKATOA, LOOP). The security property under consideration, also mentioned in Section 4.1, is that only `ISOExceptions` are thrown at the top level. Transaction related properties are not investigated. Like in the present study, it is stressed that two-dimensional `byte` arrays and the use of `byte` arrays in general are problematic in JAVA CARD verification, and have serious impact on the size and complexity of proofs. One of the main results is that subtle bugs were found in the applet.

Gemplus provides a JAVA CARD case study similar to *Demoney* [BMGL01], also a purse application and publicly available.[6] We do not use it at the moment, because it contains a large number of features that detract from the basic issues and make it less suitable as a starting point for JAVA CARD verification. In addition, it was not developed further in the last three years.

Related work in JAVA CARD verification can be classified according to several criteria. Working on byte code avoids the problems of source code availability and compiler trustworthiness, but makes full verification more difficult due to information loss during compilation. An overview of work done on the byte code level is provided in [Boy03] – we concentrate on efforts targeted at source code: here, one can distinguish between methods that attempt complete modelling of the JAVA CARD semantics and those that do not. The latter include model checking and extended static checking.

Model checking is based on a suitable abstraction of the execution model, which in the Bandera project [CDHR00] is JAVA, and of the requirements. The advantages are full automation of the model checking phase, trace generation for counter models, and treatment of concurrent JAVA programs. The drawback is the need for abstraction which poses difficulties for programs containing JAVA arithmetic and other inductive data structures. Bandera handles JAVA, not JAVA CARD, and hence no transactions. In design-by-contract [Mey92] and extended static checking (ESC) [FLL+02] JAVA source code is decorated with annotations from a restricted language. Annotated programs (via an intermediate representation) undergo a dynamic analysis that produces first-order verification conditions for a theorem prover. The analysis does not attempt to be complete, but it is fully automatic and produces warnings, when annotations are potentially violated. ESC is related to our strong invariants, because arbitrary code locations can be annotated with object invariants [LS97]. An approximation of strong invariants within ESC can be obtained by annotating every program point with the desired invariant.[7] Again, atomic transactions are not supported, as the target language is JAVA.

[6] `http://www.gemplus.com/smart/enews/st1/pacap.html`

[7] We thank Rustan Leino for pointing this out.

Closest to our approach are source code verifiers for JAVA based on various program calculi. The LOOP tool [JP03] translates JAVA source code with JML specifications into theories for the PVS theorem prover. JAVA semantics is described with co-algebras and uses higher-order logic as an internal representation. Higher-order logic is also used to formalise syntax and semantics of a JAVA fragment in Isabelle [vO01] and in the KRAKATOA tool [MPMU04]. In the latter JAVA programs and their JML specifications are translated into an intermediate, mostly functional, language, then proof obligations are generated, which in turn are proved with the COQ proof assistant. The JIVE system [MMPH00] is based on an extended Hoare style calculus, Jack [BRL03] on weakest precondition calculus, and KIV [Ste01] on Dynamic Logic. The last three systems are closely related to the KeY Prover in that they all axiomatise JAVA with logical rules that can be seen as a small step operational semantics and proofs can be interpreted as symbolic execution with induction. The differences lie in the details and scope of the axiomatisation as well as support for automation. As far as we know, KeY is the only system that supports strong (object) invariants and, in particular, the semantics of JAVA CARD transactions.

6 Conclusions

In this paper, we presented and analysed a case study concerned with formal specification and verification of JAVA CARD programs. Our results show that largely automated formal verification of realistic JAVA CARD applications without abstraction is possible in the near future. It is possible already now provided that applications are designed with verification in mind from the start. We gave a number of simple design guidelines that drastically simplify proofs while creating only a moderate performance overhead. We believe this to be acceptable, because even in the smart card world, performance restrictions become less of an issue. Besides, a small memory overhead seems an acceptable price for provably correct programs.

We concentrated in this case study on safety (data consistency) properties in the presence of transactions and possible arithmetic overflow. Information flow, memory allocation, well-formedness of transactions, and error analysis would be possible to formulate, but we cannot say anything about feasibility at this time. Temporal relationships between the execution of different code fragments as needed to enforce an order on method calls are a topic for future research.

Acknowledgements

We would like to thank Renaud Marlet of Trusted Logic S.A. for providing the *Demoney* case study. We also thank the organisers of CASSIS'04 for the opportunity to present this work. We thank the following people for reading drafts of this paper and providing valuable feedback: Renaud Marlet, Steffen Schlager, and Martin Giese. The anonymous reviewers helped with their constructive criticism and pointers to relevant literature to improve the paper.

References

[ABB+04] 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.

[BCW+02] Pierre Bieber, Jacques Cazin, Virginie Wiels, Guy Zanon, Pierre Girard, and Jean-Louis Lanet. Checking secure interactions of Smart Card applets. *Journal of Computer Security*, 10(4):369–398, 2002.

[Bec01] 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.

[BGH+04] 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.

[BM03] 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 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.

[BMGL01] Eric Bretagne, Abdellah El Marouani, Pierre Girard, and Jean-Louis Lanet. PACAP purse and loyalty specification v0.4. Technical report, Gemplus, January 2001.

[Boy03] 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.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In *Proceedings, Formal Methods Europe 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[BS01] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference*

on Automated Reasoning, Siena, Italy, LNCS 2083, pages 626–641. Springer, 2001.

[BS03] 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.

[BS04] 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.

[CDHR00] 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.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Java Series. Addison-Wesley, 2000.

[DHS04] 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.

[FLL+02] 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.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic.* MIT Press, 2000.

[HP04] Engelbert Hubbers and Erik Poll. Reasoning about card tears and transactions in Java Card. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.

[HW03] Reiner Hähnle and Angela Wallenburg. Using a software testing technique to improve theorem proving. In Alex Petrenko and Andreas Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES), Montréal, Canada*, volume 2931 of *LNCS*, pages 30–41. Springer, 2003.

[JMR04] 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.

[JP03] 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.

[LS97] K. Rustan M. Leino and Raymie Stata. Checking object invariants. Technical Note #1997-007, Digital Systems Research Center, Palo Alto, USA, January 1997. Available from `ftp://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-007.ps.gz`.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.

[MM01] 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.

[MM02] Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.

[MMPH00] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The JIVE system – Implementation description. Available from `http://softech.informatik.uni-kl.de/downloads/publications/jive.pdf`, 2000.

[Mos02] 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`.

[MPMU04] 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`.

[Pra77] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.

[RCK03] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants for imperative programs. Available from `http://www.lsi.upc.es/~erodri/ijcar04ex.ps`, November 2003.

[Ste01] Kurt Stenzel. Verification of Java Card programs. Technical Report 2001–5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available from `http://www.Informatik.Uni-Augsburg.DE/swt/fmg/papers/`.

[Sun03] Sun Microsystems, Inc., Santa Clara/CA, USA. *Java Card 2.2.1 Platform Specification*, October 2003.

[TH02] Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, volume 2422 of *LNCS*, pages 334–348. Springer-Verlag, 2002.

[vO01] David von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, January 2001.

[ZG03] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen, Fachbereich für Mathematik und Informatik, 2003.

Paper V

Formalisation and Verification of JAVA CARD Security Properties in Dynamic Logic

Fundamental Approaches to Software Engineering Conference 2005, Edinburgh, Scotland, April 2005

Formalisation and Verification of Java Card Security Properties in Dynamic Logic

Wojciech Mostowski

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 [Che00] 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 [MM01]. 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 [HM05] 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 [MM01], 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] [ABB+04]. 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.

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

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 [Obj03]. 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 [Che00] 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 protocol data units (APDUs, ISO 7816–4 standard). The communication is done in master-slave mode – it is 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 oth-

ers. 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 [Pra77, HKT00] 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) [BS01]. 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 [Bec01]. 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 [BGH+04], 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 [BS04].

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 (for example, 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 [Che00] 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 [BM03] and the practical experiences in [HM05]. 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 [Boy03] and model checking [CDHR00]. 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. [JMR04] 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 (successor of ESC/JAVA [FLL+02]), KRAKATOA [MPMU04], JIVE [MMPH00], and LOOP [JP03]. 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] [MM02]. 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 [HM05], 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 [Mos02].

4 Security Properties

The security properties that we discuss here are directly based on the ones described in [MM01], 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.

[2] We thank Renaud Marlet of Trusted Logic S.A. for providing the *Demoney* code.
[3] `http://www.ibutton.com`

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 (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 $\langle p \rangle true$. 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;
}
```

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

```
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
   & !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

[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.

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 {
  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).

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 [Mül01, BS03], 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, for example, 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, for example, 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 [HM05], for this work however we were able to use the real *Demoney* applet instead of the simplified one used in [HM05]. 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 (for example, 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 (for example, 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 do not 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 [BS04].

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 (for example, 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 [DHS04], 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 [JMR04] 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 [JMR04] 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 [Pla04]. 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 [BGH+04], 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 [Mül01, BS03], 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, for example, 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 [Ged04]. 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

[ABB+04] 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.

[Bec01] 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.

[BGH+04] 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.

[BM03] 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 2003, Warsaw, Poland*, volume 2621 of *LNCS*, pages 246–260. Springer, April 2003.

[Boy03] 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.

[BS01] 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.

[BS03] 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.

[BS04] 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.

[CDHR00] 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.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Java Series. Addison-Wesley, 2000.

[DHS04] 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.

[FLL+02] 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.

[Ged04] Tobias Gedell. Integrating static analysis into theorem proving, 2004. Available from `http://www.cs.chalmers.se/~gedell/publications/satp.ps`.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[HM05] Reiner Hähnle and Wojciech Mostowski. Verification of safety properties in the presence of transactions. In Gilles Barthe and Marieke Huisman, editors, *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *LNCS*, pages 151–171. Springer, 2005.

[JMR04] 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.

[JP03] 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.

[MM01] 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.

[MM02] Renaud Marlet and Cédric Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.

[MMPH00] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The JIVE system – Implementation description. Available from `http://softech.informatik.uni-kl.de/downloads/publications/jive.pdf`, 2000.

[Mos02] 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`.

[MPMU04] 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`.

[Mül01] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[Obj03] Object Modeling Group. *Unified Modelling Language Specification, version 1.5*, March 2003.

[Pla04] André Platzer. Using a program verification calculus for constructing specifications from implementations. Minor thesis, Karlsruhe University, Computer Science Department, Karlsruhe, Germany, February 2004.

[Pra77] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.