Thesis for the Degree of Licentiate of Philosophy

Towards Development of Safe and Secure JAVA CARD Applets

Wojciech Mostowski

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, December 2002

Towards Development of Safe and Secure Java Card Applets
Wojciech Mostowski

© Wojciech Mostowski, 2002

Technical Report no. 16L
ISSN 1651–4963
School of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, 2002

Abstract

This thesis is concerned with different aspects of Java Card application development and use of formal methods in the Java Card world. Java Card is a technology that provides means to program smart (chip) cards with (a subset of) the Java language. The use of formal methods in the Java Card context is highly justified due to the criticality of Java Card applications. First of all, Java Card applications are usually *security critical* (e.g., authentication, electronic cash), second, they are *cost critical* (i.e. they are distributed in large amounts making updates quite difficult) and finally, they can also be *legally critical* (e.g., when the digital signature law is considered). Thus the robustness and correctness of Java Card applications should be enforced by the best means possible, i.e. by the use of formal verification techniques. At the same time Java Card seems to be a good target for formal verification—due to the relative simplicity of Java Card applications (as compared to full Java), formal verification becomes a feasible and manageable task. In this thesis, we touch upon different levels of Java Card application development and the use of formal methods. We start by defining a UML/OCL supported development process specifically tailored to Java Card applications, then we go on to define an extension to the logic used in formal verification of Java Card programs to handle so called "rip out" properties (properties that should be maintained in case of an unexpected termination of a Java Card program), which are specific to Java Card. Finally, we end up with a simple tool support for Java Card program compilation and testing inside a CASE tool. The thesis contains three papers focusing on these aspects. The main purpose of this work is to ensure the robustness of Java Card applications by providing a well defined development process and means to formally verify properties specific to Java Card applications to be able to develop Java Card applets which are robust "by design". At the same time we want to make rigorous Java Card development relatively easy, tool supported (automated wherever possible) and usable by people that do not have years of training in formal methods.

Contents

Introduction

1 Overview

This thesis is concerned with different aspects of Java Card application development and use of formal methods in the Java Card world. Java Card is a technology that provides means to program smart (chip) cards with (a subset of) the Java language. The use of formal methods in the Java Card context is highly justified due to the criticality of Java Card applications. First of all, Java Card applications are usually *security critical* (e.g. authentication, electronic cash), second, they are *cost critical* (i.e. they are distributed in large amounts making updates quite difficult) and finally, they can also be *legally critical* (e.g. when the digital signature law is considered). Thus the robustness and correctness of Java Card applications should be enforced by the best means possible, i.e. by the use of formal verification techniques. At the same time Java Card seems to be a good target for formal verification—due to the relative simplicity of Java Card applications (as compared to full Java), formal verification becomes a feasible and manageable task. In this thesis, we touch upon different levels of Java Card application development and the use of formal methods. We start by defining a UML/OCL supported development process specifically tailored to Java Card applications, then we go on to define an extension to the logic used in formal verification of Java Card programs to handle so called "rip out" properties (properties that should be maintained in case of an unexpected termination of a Java Card program), which are specific to Java Card. Finally, we end up with a simple tool support for Java Card program compilation and testing inside a CASE (Computer Aided Software Engineering) tool. The thesis contains three papers focusing on these aspects. The main purpose of this work is to ensure the robustness of Java Card applications by providing a well defined development process and means to formally verify properties specific to Java Card applications to be able to develop Java Card applets which are robust "by design". At the same time we want to make rigorous Java Card development relatively easy, tool supported (automated wherever possible) and usable by people that do not have years of training in formal methods.

All the work presented in this thesis was carried out in the context of the KeY project [1, 12]. In general the KeY project is concerned with integrating object oriented design with formal methods and, in particular, with formal verification of real world object oriented programs (in this case JAVA/JAVA CARD). The integration of the object oriented design with formal methods is realised as the KeY system (tool), which is built on top of a commercial CASE tool.

In Section 2 we describe the KeY system, its aims and current state in more detail. Section 3 gives a brief introduction to JAVA CARD technology and describes the JAVA CARD language in some detail. Section 4 describes the papers presented in this thesis in more detail and how they fit into this thesis. The contributions those papers provide to the KeY project are described in Section 5. Section 6 gives pointers to work related to formal verification of JAVA and JAVA CARD programs, and finally Section 7 gives some directions for future work.

2 The KeY System

KeY [1, 12] 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)*. To achieve this, the system is realised as an extension of a commercial UML-based 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 proper combination at all. 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.

The target language of KeY-driven software development is JAVA. More specifically, the verification facilities of KeY are restricted to code written in JAVA CARD [21, 6]. JAVA CARD is a proper subset of the JAVA programming language, excluding certain features (like threads, cloning or dynamic class loading) and with a much reduced API. The JAVA CARD language [21] and platform [22] are provided by Sun Microsystems to enable JAVA technology to run on smart cards and other devices with limited memory, such as embedded systems. Due to the relative simplicity of the JAVA CARD language (compared to full JAVA), verification of the full JAVA CARD language becomes a *feasible* task. At the same time, as already outlined above, the quality of JAVA CARD applications is highly critical. Thus, choosing JAVA CARD language as the target language for verification in the KeY system is highly *justified*. It is important to note here that the KeY system is not really restricted to be used for the development of smart card applications, because many Java applications do not use features excluded by Java Card. In the next section we present a short description of Java Card technology and language.

UML based software development puts 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 [17] includes the *Object Constraint Language (OCL)*. 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 the 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.

Most of the described features of the KeY system are already implemented, however the system is still under development. The current publicly available version can be downloaded from KeY project's webpage [12].

3 Java Card

Here we give a short introduction to Java Card technology and language [6, 21, 22] that will shed more light on the papers included in this thesis.

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 (e.g. electronic cash) and the RAM is used for temporary calculations. 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's 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" (e.g. 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 Java, 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. 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 Java Card run-time environment).

Finally, we present a small example of a Java Card applet to give a feeling for how the 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's 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. Notice 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 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 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 commited, 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 goes 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.

In reality the JAVA CARD transaction mechanism is a bit more complicated than this simple example shows. See the second paper of this thesis for the full account.

4 Description of the Papers

In general this thesis contributes to the world of Java Card application development and formal verification in three ways. First of all, it defines a development process for Java Card. The development process is based on UML/OCL and is designed to enable the use of formal verification. Basically two kinds of OCL specifications are involved in the design process—generated from a UML sequence diagram and generated from specification templates (KeY specification idioms). Then the actual Java Card code can be proven correct with respect to the specification using the KeY interactive prover. Second, we extend the Java Card Dynamic Logic used in the KeY interactive prover to handle "rip-out" properties ("strong" invariants)—properties that should be maintained in case of an unexpected termination of a Java Card program. Finally, we present a tool set to support handling of the actual Java Card devices inside the CASE tool—compiling, downloading and testing Java Card applets.

In the following we elaborate more on the three papers presented in this thesis, describe their goals and the relations between them.

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's based on a real-life case study, which is a system for authenticating users in the Linux system with Java Powered iButtons[1] 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 with the KeY system. 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

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

point the property is maintained. The problem of handling (and in particular proving) "rip-out" (or throughout) properties is the subject of the next paper.

This paper was presented at the Rigorous Object Oriented Methods Workshop in London, March 2002. The paper was later invited for submission to the Software and Systems Modelling Journal by the workshop organisers and is currently undergoing the review process.

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 steps). The main challenge in this work was to handle JAVA CARD's transaction mechanism 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 "box" (partial correctness) and "diamond" (total correctness) modal operators (specifically programmatic abortion of a transaction)—the necessary rules to handle transactions for box and diamond operators are also presented in the paper. To our knowledge none of the published formalisations of JAVA CARD actually handle transactions and object persistency. The paper also contains examples of simple proofs using the new rules.

The paper was submitted to the Fundamental Approaches to Software Engineering Conference 2003 and is currently undergoing the review process.

JAVA CARD Tools for Together Control Center

This paper is a brief description of a tool set for the Together Control Center CASE tool that supports development of JAVA CARD applets. In particular, the tool set provides a uniform environment for compiling, installing and testing JAVA CARD applets inside the CASE tool, independently of the actual JAVA CARD platform (device) used. This way all vendor specific solutions and user interfaces for creating and testing JAVA CARD applets are overcome providing a uniform environment for JAVA CARD applet development with powerful UML support from the CASE tool.

5 Contribution to the KeY Project

This work contributes to the KeY project in many ways. The first paper provides a useful, real-life case study—the KeY system was tested in practice on this case-study, both the modelling capabilities of the CASE tool as well as the verification capabilities of the KeY extensions were used to reengineer the case study, thus giving a good test of the whole system. Additionally, the JAVA CARD tool set mentioned in the third paper was used for comfortable testing of the applet within the KeY system. The first paper also provides the development process, which can be used in constructing new case studies for the KeY system and also can serve as a basis for defining a general "KeY development process" ("KeY method").

The second paper extends the Java Card Dynamic Logic used in KeY's interactive theorem prover. We already argued about the importance of treating the "rip-out" problem for Java Card. The extension of the logic with the "throughout" operator and transaction handling enhances the functionality of the KeY system to treat the "rip-out" properties. Additionally the transaction handling "completes" the logic to handle full Java Card. Also, introducing the "throughout" operator can be treated as a first step to introduce other temporal operators to the logic. Such temporal operators can be used to specify and prove correctness of non-terminating programs (e.g. control software).

Finally, the Java Card tool set described in the third paper completes the KeY system with the possibility to access and operate Java Card devices from within the KeY system. This makes the KeY system an integrated platform for the development of Java Card applications: modelling, coding, verifying, installing and testing Java Card applets.

6 Related Work

Here we present some work that is closely as well as generally related to the material presented in this thesis. We start with projects related to Java Card as such. The largest project dedicated to Java Card is the VerifiCard project [25, 9, 3] together with the LOOP project [14]. The VerifiCard project aims at providing the basic technology for verification of the Java Card platform (virtual machine) and Java Card applications with the emphasis on the byte-code level verification, while the LOOP project concentrates on the source code verification of Java Card programs annotated with Java Modelling Language (JML) [11] (see e.g. [24, 16]). GemPlus (a major producer of Java smart cards) carries out some work towards automated testing of Java Card applications [15].

Next we would like to mention some approaches that provide formalisations of Java semantics and proof systems targeted at Java. To start with, [8] contains an overview of the existing literature on Java and Java Card safety with emphasis on formal approaches. The mentioned approaches include the following:

- [18] presents a Hoare style programming logic for sequential Java. This work is related to the VerifiCard project.

- [10, 23] present an approach to formal verification of sequential Java programs using a Hoare like logic formalised in Type Theory. The machine assisted proving is done using PVS. This work is related to the LOOP project.

- [19] gives a formalisation of JAVA using Abstract State Machines.
- Finally, Compaq's ESC/JAVA project [13, 7] develops a tool for automatic, static checking of JAVA programs annotated with specifications written in (a subset of) JML.

Last, but not least, we should mention the KeY system's formalisation of JAVA/JAVA CARD—JAVA CARD Dynamic Logic [4, 5], which is heavily explored in the second paper of this thesis. It is an extension of Dynamic Logic to handle sequential JAVA and, in particular, JAVA CARD.

Dynamic Logic was also used in the KIV system to prove correctness of imperative programs [2]. More recently, the KIV system supports also a fragment of the JAVA language [20].

7 Future Work

The work presented in this thesis can be continued in a number of ways:

- The JAVA CARD Dynamic Logic rules for the "throughout" operator and transaction mechanism presented in the second paper should be implemented in the KeY system's interactive prover and then tried with real (extensive and meaningful) examples. Another possible direction here is to formally prove the soundness and completeness of these rules. Such work (using Isabelle) for the unextended JAVA CARD Dynamic Logic has been already tried and could possibly be adapted to include the "throughout" and transactions extensions.
- Security properties and security related design patterns could be studied in the context of the first paper, i.e. the development process could be extended to focus on the security of JAVA CARD applications via the use of security related idioms and design patterns adapted to JAVA CARD environment.
- The work from the first paper could be generalised to the broader aspect of using JAVA with the small devices, i.e. a similar, more general development process could be considered for JAVA2 Micro Edition applications. This would involve developing a meaningful (real life) J2ME application for a case study.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering (FASE), Part of Joint European Conferences on*

Theory and Practice of Software, ETAPS, Grenoble, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.

[2] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*. Springer-Verlag, 2000.

[3] Gilles Barthe, Simao Sousa, Guillaume Dufay, and Marieke Huisman. Jakarta: A toolset for reasoning about JAVA CARD. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[4] 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*, LNCS 2041, pages 6–24. Springer, 2001.

[5] Bernhard Beckert and Bettina Sasse. Handling JAVA's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.

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

[7] ESC/JAVA homepage. `http://www.research.compaq.com/SRC/esc/`.

[8] 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.

[9] B. Jacobs, H. Meijer, and E. Poll. VerifiCard: A european project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001.

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

[11] JAVA Modelling Language homepage. `http://www.cs.iastate.edu/~leavens/JML.html`.

[12] KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.

[13] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking JAVA programs via guarded commands. Technical report, Compaq Systems Research Center, Palo Alto, California, 1999.

[14] The LOOP project homepage. `http://www.cs.kun.nl/~bart/LOOP`.

[15] H. Martin and L. du Bousquet. Tools for automated conformance testing of Java Card applets. Technical report, Gemplus, September 2000.

[16] Hans Meijer and Erik Poll. Towards a full formal specification of the Java Card API. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[17] Object Modeling Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

[18] 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 *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

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

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

[21] Sun Microsystems, Inc., Palo Alto/CA. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997.

[22] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Platform Specification*, September 2002.

[23] 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.

[24] 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, *Java on Smart Cards: Programming and Security*, volume 2041 of *LNCS*, pages 137–150. Springer-Verlag, 2001.

[25] VerifiCard project homepage. `http://verificard.org/`.

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 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 modern CASE tool support.

1 Introduction

In this paper 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 [1]. First we analyse the current situation of JAVA CARD application development, then we present a real life JAVA CARD case study (`pam_iButton` [7]) 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 [15] specifications, software idioms and design patterns, formal specifications and a modern CASE tool support 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 [8]. JAVA CARD provides 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's always the master/terminal application that initialises the communication by sending the command APDU to the card and then the card replies by sending a response APDU (possibly with empty contents). In 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 incoming command APDUs and sending the response APDUs back to 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 (the active one is the one most recently selected by Java Card run-time environment).

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 Java Card language is based on full Java the nature of the Java Card environment (e.g. constrained memory, no garbage collection) makes Java Card programming very 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 (`http://www.ibutton.com/`), which we use in our research, and the development environment (iB-IDE—`http://www.ibutton.com/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 and Open Card

Framework [16] 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. One more thing which should be mentioned here is the fact that 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 which means that those solutions are not (easily) portable to other Java Card platforms. In contrast to this, SUN's Java Card reference development kit (`http://java.sun.com/products/javacard/`) 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 thing available are command line tools for compilation and running the applets in a simulated/emulated environment.

The next issue we want to discuss 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 numbers is not possible, thus correctness of the card application should be ensured 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 make 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 the Nokia OK process) will benefit from adding the formal methods support to it.

Taking all this into account it becomes clear that the development of Java Card applications needs to be done in a controlled, systematic, well defined, rigorous way giving the possibility to formally verify the application's properties.

1.3 Related Work

We briefly discuss some other work that is done in the areas around Java and Java Card program development and verification. The largest project concerned with the use of formal methods for Java Card is the VerifiCard project [20, 10, 3]. Its goal is to provide the basic technology for verification of both the Java Card platform and Java Card applications. The project does not concentrate so much on the actual design process of Java Card applications. Trusted Logic [19] provides tools for the verification of Java Card byte-code (on-card byte-code verifier). GemPlus (a major producer of Java smart cards) carries out some work towards automated testing of Java Card applications

[13]. The Open Card organisation [16] 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). Compaq's ESC/Java project [9] develops a tool for automatic, static checking of Java programs annotated with specifications written in (a subset of) Java Modelling Language. Finally [11] shows how UML can be used to express security requirements during system development.

1.4 Our Approach

In our approach to 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 [1, 6]. 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 [2]. 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.

Having all this support we can make Java Card applications robust and secure by design and easier for verification. For successful verification one needs a suitable formal verification system, and this is where we turn to the KeY project [1, 12] which we use as our framework. The KeY project is concerned with integrating formal methods and object oriented software design. It incorporates both into one framework by extending a commercial UML CASE tool with formal verification modules in a seamless way. The KeY project focuses on Java source code verification (as opposed to byte code verification) and the usage of the formal verification extensions is fully integrated into the tool. The KeY project also proposes the usage of a set of standard specifications that can be applied to common problems. We will use some of those specification patterns in our approach. In order to achieve our goals we also need support from a modern, fully customisable UML CASE tool. The one we use is Together Control Center from TogetherSoft (`http://www.togethersoft.com/`). It provides very good support for UML and a Java open API, by which most of the tool features can be accessed and extended, which makes it extremely suitable for our purposes. Since it is the same tool the KeY project uses it is easy to integrate our solutions into the KeY system. Another reason to use an independent CASE tool is that we want to make our solutions to Java Card design issues independent of the actual Java Card platform and vendor specific development environment obtaining generic, powerful UML support at the same time. It also should be mentioned that at this point of our work we limit ourselves to the card applications (Java Card applets). At this stage we don't consider the problems of developing the host application, however, we will address this problem in future.

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 present our framework to solve some of the problems: first we show how UML state chart diagram can be used to define a Java Card applet behaviour and command invocation protocol, then we show how the actual implementation is derived from the diagrams in a rigorous way. Next we show how formal specifications are used to assure extra reliability and enable formal verification of a Java Card application. We also give an example of how the KeY system is used to construct specifications. Finally, Section 5 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 [7]. 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 substantially from the current Java Card API 2.1.1 specification and the upcoming 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 user ID number and a pair of private and public RSA keys is generated on the iButton and stored together with the user 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 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 is 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.

We now give some more details about `Safe_Applet`. Here is the list of the most important and interesting command APDUs that the original applet accepts:

- Store data—stores temporary data for a subsequent command.
- Authenticate user—given the user 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 ID generates an RSA key pair (the generation is done on the card) and stores it together with user ID in the applet's memory. In response sends back the public part of the key.
- Get public key—given the user ID sends back the public part of the key.
- Delete key pair (PIN code protected)—given 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.

A 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 was 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's 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, e.g. if a single applet can be used on many different systems then it certainly should be a user owning the applet's PIN code and it should be a 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 an 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 (e.g. intended for 'authenticate user'), which may cause corruption of the applet data. We only mentioned the 'authenticate user' command that relies on 'store temporary data', but there are other commands doing this too. 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. There are also no integrity checks on the data being sent along, which connects to the previous as well as the next problem. Some of the commands may require input that does not fit into a single APDU, so there are multiple APDUs being sent, but 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 established solution of presenting the PIN code once per command exchange (card) session.

Another thing which we find 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—it may decline proper functioning at any point of execution where memory allocation occurs when there is no free memory on the device left (and this is very likely to happen on such a memory constrained device). Such an approach to Java Card programming also makes the applications not portable to other Java Card devices that don't support garbage collection.

While testing `Safe_Applet` we came to another interesting issue. If the user rips out the iButton from the reader during authentication the applet is not functioning properly any more during subsequent authentication sessions. At first it seemed to be a simple programming mistake, but it turned out not to be. We strongly suspect the Java Card run-time environment to be the source of the problem. However the point here to make is that the design of a Java Card application should take similar possibilities into account and try to make the applets as robust and rip-out proof as possible, assuming of course that the underlying Java Card run-time environment is implemented properly.

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 it should be forbidden by the applet to make double entries of this kind, instead of making the user responsible for controlling the state of the key pair entries in the applet.

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 applet's 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 [8].

To end this section we want to make an important note. In our work we don't want to impose certain ways of solving design problems for Java Card applications (e.g. what actual deployment steps the applet should have or whether a certain command should be PIN code protected or not), 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, this time doing things in a more defined way. We will also point to places where CASE and verification tools play an important role. We will not develop the whole `Safe_Applet` here, only the things important to exemplify our approach.

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 went wrong during normal applet usage, e.g. 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.

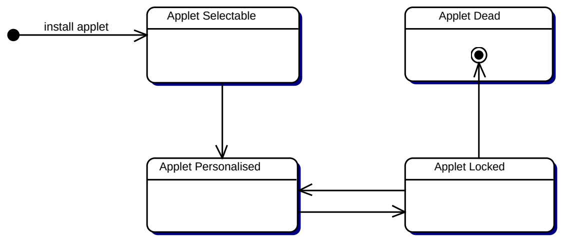


Figure 1: `Safe_Applet` life states

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 personalised and locked states many times during its life time. We will show later what are the exact conditions that cause an applet's life state change. 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.

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

4.2 Applet Commands

We are now at the point where we can start defining the command APDUs that the applet should support. The commands presented here are slightly different from the original ones described in Section 2. This is because the new set of commands is intended to avoid some of the problems described earlier (e.g. unnecessary PIN code sending, see verifyUserPIN below). For each of the commands we give its 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 done in the issuer's trusted area it is not necessary to require PIN authentication for updating the user 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 invocation it should move the applet from state selectable to personalised.

verifyUserPIN This command performs the verification of the user 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 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 one) for a given user 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 ID from applet's memory.

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

disableUser, enableUser Those commands are used to disable and enable the authentication of a given user specified by a user 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 (e.g. in order 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).

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 triggered 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

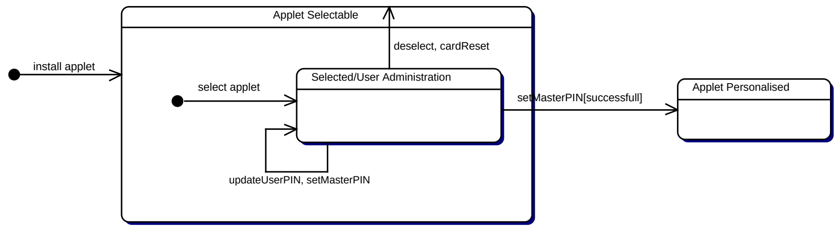


Figure 2: Command states in the selectable life state

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 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.

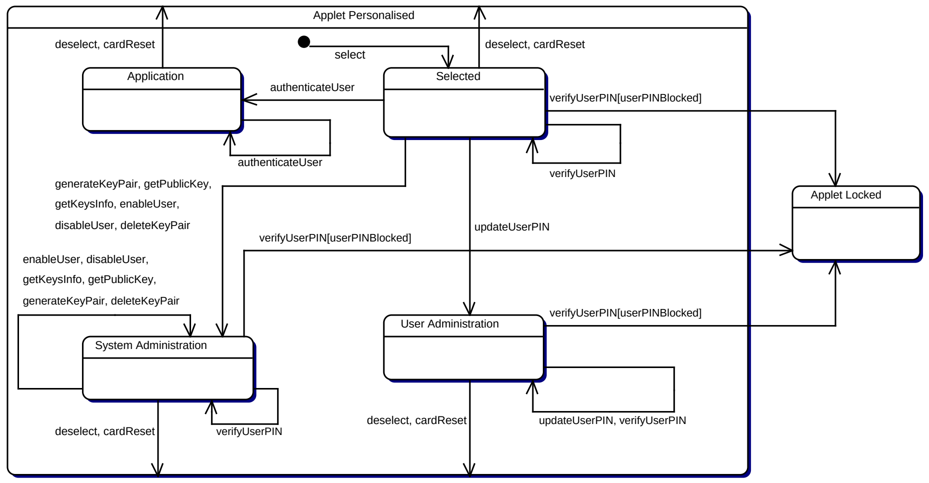

Applet Personalised
deselect, cardReset
select
deselect, cardReset
Application
authenticateUser
Selected
verifyUserPIN[userPINBlocked]
authenticateUser
verifyUserPIN
generateKeyPair, getPublicKey, getKeysInfo, enableUser, disableUser, deleteKeyPair
Applet Locked
updateUserPIN
verifyUserPIN[userPINBlocked]
enableUser, disableUser, getKeysInfo, getPublicKey, generateKeyPair, deleteKeyPair
verifyUserPIN[userPINBlocked]
User Administration
System Administration
deselect, cardReset
updateUserPIN, verifyUserPIN
deselect, cardReset
verifyUserPIN

Figure 3: Command states in the personalised life state

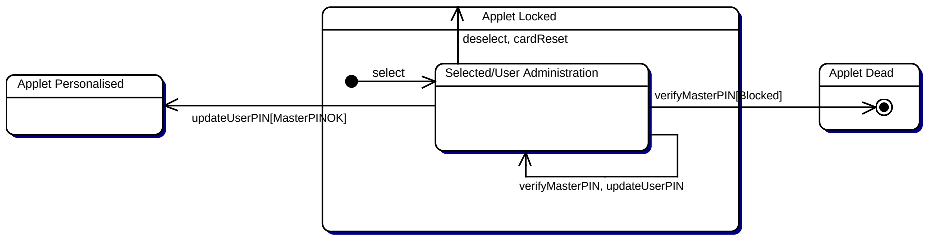


Figure 4: Command states in the locked life state

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 failed 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 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 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.

Before continuing we would like to make a comment: notice 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 so far.

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

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

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 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's 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 required nor input data integrity verification. 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);
  }
}
```

The `readBigInput` method requires a bit more attention. Both `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);
  }
}
```

More interesting methods are `setAppletState`, `updateCommandState` and `readBigInput`. 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 (e.g. 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
  }
}
```

As we said before we want to use the support of a CASE tool to automate the development process. Parts of the applet were created automatically by Together Control Center extension modules we developed, e.g. the skeletons for command dispatching methods were created automatically out of the given command tables. Most of the other code was engineered "by hand". However we still see possibilities to automate the process further 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 (see the following subsection). This can be easily implemented in Together Control Center.

- Generating (possibly with a little help from the developer) 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 the tool of our choice (e.g. in Together Control Center there are ready 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.

4.5 Formal Specification

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 Object Constraint Language (OCL, part of UML) [21] specifications like the following. Those specifications don't 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, e.g.:

```
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 don't 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 pattern that produces our 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. 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 another situation 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 pretty 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, e.g. by giving the master PIN. To verify such a property one needs formalisation of Java integer arithmetics that handles properly the "modulo" behaviour of Java integer types. The KeY system both supports the specification idioms [2] and contains formalisation of Java integer arithmetics as part of the KeY specification library [5].

The specifications we have shown can be subjected to formal verification. Parts of the specifications that describe the behaviour of the state machines controlling the life cycle and the protocol were successfully verified with the KeY system. Most of the other specifications refer to methods that use the Java Card API and to verify them one needs to have access to formal specifications of the API. Such specifications [14, 18] for JML and ESC/Java are publicly available on the web [17], but they need to be ported to OCL to be usable with the KeY system.

One thing that we did not specify is that the applet data stays consistent in case when an applet's execution terminates abnormally by ripping out the card from the reader. This would require 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 OCL, neither is it possible to express in the KeY system's Dynamic Logic for Java, however, this problem has been solved for pure Dynamic Logic [4]. We are currently working on extending Dynamic Logic for Java with an "always" operator, which will allow to specify and prove such "rip-out proof" properties with the KeY system. Here we should also mention that the "rip-out" problem of

Safe_Applet described in Section 3 does not exist any more in the reengineered version of the applet.

4.6 Employing the KeY System

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. Let's first 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 shouldn't 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:

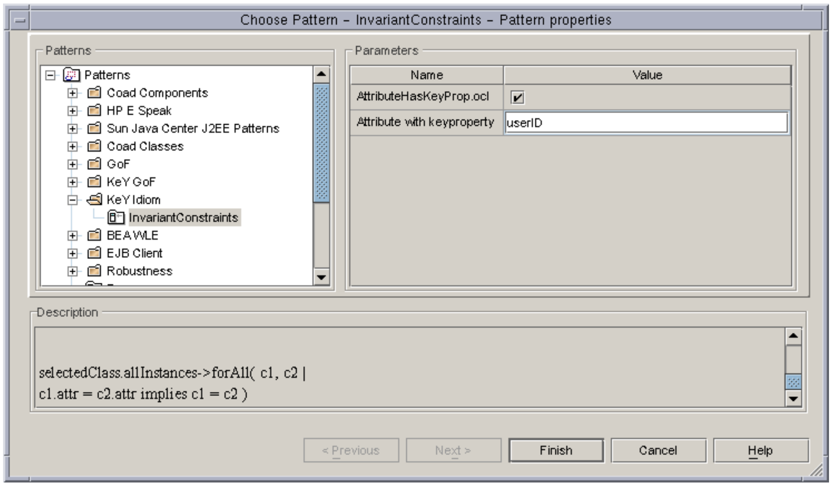
```
context User:
inv: User.allInstances->forAll(c1, c2 |
       c1.userID = c2.userID implies c1 = c2)
```

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)
```

5 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 (we have developed a Together Control Center plug-in supporting this—http://www.cs.chalmers.se/~woj/javacard/). Most of the code we have shown was developed by hand, but we were precisely following the UML diagrams we constructed, the coding was



Name	Value
AttributeHasKeyProp.ocl	✓
Attribute with keyproperty	userID

```
selectedClass.allInstances->forAll( c1, c2 |
c1.attr = c2.attr implies c1 = c2 )
```

Figure 5: Applying specification patterns in the KeY system

quite straightforward and almost a one pass process—we made the applet work in the expected way in a very short time. However, as we already mentioned, designing the application in UML requires some expertise in a given domain and is a bit more lengthy process.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. 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 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[2] 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.

[3] Gilles Barthe, Simao Sousa, Guillaume Dufay, and Marieke Huisman. Jakarta: A toolset for reasoning about Java Card. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France*. Springer-Verlag, September 2001.

[4] 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.

[5] Bernhard Beckert and Steffen Schlager. Integer arithmetic in the specification and verification of JAVA programs. In *Proceedings, Workshop on Tools for System Design and Verification (FM-TOOLS), Reisensburg, Germany*, 2002. To appear.

[6] D. Bolignano, D. Le Métayer, and C. 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.

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

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

[9] ESC/JAVA homepage. `http://www.research.compaq.com/SRC/esc/`.

[10] B. Jacobs, H. Meijer, and E. Poll. VerifiCard: A european project for smart card verification. *Newsletter 5 of the Dutch Association for Theoretical Computer Science (NVTI)*, 2001.

[11] Jan Jürjens. Developing secure systems with UMLsec — from business processes to implementation. In Andreas Pfitzmann Dirk Fox, Marit Köhntopp, editor, *Proc. Verlässliche IT-Systeme 2001 — Sicherheit in komplexen IT-Infrastrukturen, Kiel, Germany.* Vieweg Verlag, 2001.

[12] KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.

[13] H. Martin and L. du Bousquet. Tools for automated conformance testing of JAVA CARD applets. Technical report, Gemplus, September 2000.

[14] Hans Meijer and Erik Poll. Towards a full formal specification of the JAVA CARD API. In *Smart Card Programming and Security, International Conference on Research in Smart Cards, e-Smart 2001, Cannes, France.* Springer-Verlag, September 2001.

[15] Object Modelling Group. *Unified Modelling Language Specification, version 1.4*, September 2001.

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

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

[18] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the Java Card API in JML. CSI Report CSI-R0005, Computing Science Department, Nijmegen, March 2000.

[19] Trusted Logic homepage. `http://www.trusted-logic.fr/`.

[20] VerifiCard project homepage. `http://verificard.org/`.

[21] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

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 an 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, 9]. 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 [6] (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 exists in JAVA and that 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 out the smart card from 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.

To ensure that a JAVA CARD program is "rip out" safe we need to be able to specify "strong" invariants—invariants that hold throughout the whole execution of a JAVA CARD program (but not when a transaction is in progress). The

*Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe, Germany, e-mail: beckert@ira.uka.de

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) [2, 3]. An extension to pure Dynamic Logic to include trace modalities "throughout" and "at least once" is presented in [4]. Here, we extend that work and introduce the "throughout" operator to Java Card DL (we don't 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 that is not a difficult task.

Related work. As already mentioned the work presented here is based on [4], 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 [5] for one of them and an overview of some others. In [16] 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 gives some insights into the Java Card transaction mechanism. Section 3 contains a brief introduction to Java Card Dynamic Logic and Section 4 introduces the "throughout" operator in detail, gives its semantics and sequent calculus rules to handle the new operator and transaction mechanism. Section 5 presents 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 [1, 9] 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 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 [11]. 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" proof. That is, it is possible to destroy the applet's functionality by taking out (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 execution of a method. Normally it is not required for an invariant to hold in the intermediate states of a method's execution (although it is not very clear what the precise semantics of OCL invariant is). To solve this problem, we introduce "strong" invariants, which allow us 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 objects of type `PersonalData` 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 for this paper. A full description of the transaction mechanism can be found in [6, 13, 14, 15].

The memory model of Java Card differs slightly from Java's model. In smart cards there are two kinds of writable memory: EEPROM—persistent memory, which holds its contents between card sessions—and RAM—non-persistent (transient) memory, 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 contents by calling a certain method from the Java Card API (`JCSystem.makeTransient...`). Currently there is no possibility to make objects other than arrays transient.
- The contents of 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 the 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 contents remain unchanged (as if there was no 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 this program is executed 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, i.e. 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 was 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 the strong invariants will not (and should not) be checked within a transaction—in case our program is terminated abruptly while a transaction is in progress, 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 [7, 8, 10, 12] 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 [2]. 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 that is not available in plain Java Card. The purpose of this extension is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To handle 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 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_call(`x`){`*prog*`}` to occur in DL programs. Note, that 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, $\langle$`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

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

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

To simplify notation, 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 an intuition of how the JAVA CARD DL sequent calculus works.

Notation. 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_call(...){`" 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.[1] 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 a first simple example, we present the rule for the `if` statement:

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

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,

[1] 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 $\langle \pi pq\omega \rangle \phi$.

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.

The Assignment Rule and Handling State Updates. The assignment rule

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

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 $\langle$`x = ++i;`$\rangle\phi$ with $\langle$`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, which must be valid throughout program execution.

Following [4], 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 Java Card runtime environment specification [14] each single field or variable assignment is atomic. This matches exactly Java Card DL's notion of a single update. That means that a "throughout" property has to hold after every single Java

Card DL update. However, the 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 can be applied. 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 block (like the `method_call` blocks used for method bodies) 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.

Usually only the formulas of the form $\phi \rightarrow [\![p]\!]\phi$ will be considered in practice, where ϕ can be seen as a "throughout" (strong) invariant. Also one can view a normal invariant as a special case of a "throughout" property where method execution is defined to be atomic.

4.1 Additional Sequent Calculus Rules for the $[\![\cdot]\!]$ 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 $[\![\cdot]\!]$. Assignments `loc = expr;` are atomic programs. By definition, their 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` or throwing an exception) resulting in a termination of the whole loop.

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

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

where

$$\begin{aligned}\alpha &\equiv \texttt{if(}a\texttt{) \{}l_{break}: \texttt{\{try \{}l_{cont}: \texttt{\{}p'\texttt{\} abort;\} catch(Exception e)\{\}\}\}}\\ \beta &\equiv \texttt{if(}a\texttt{) }l_{cont}:\ l_{break}: \texttt{\{}p'\texttt{\}}\end{aligned}$$

In the above rule, λ is a (possibly empty) sequence "$l_1 : \dots l_n :$" 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 $\langle\alpha\rangle\mathit{true}$ evaluates to *false* and $[\beta]\mathit{Inv}$ 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 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).

Those 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 commit (`cT`),
- TRabort: —a transaction is in progress and is expected to abort (`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:

$$\frac{\Gamma \vdash \mathcal{U}\phi \quad \Gamma \vdash \mathcal{U}[\![\mathsf{TRcommit}\colon \pi\omega]\!]\phi \quad \Gamma \vdash \mathcal{U}[\![\mathsf{TRabort}\colon \pi\omega]\!]\phi}{\Gamma \vdash \mathcal{U}[\![\pi\ \mathtt{bT;}\ \omega]\!]\phi} \tag{R5}$$

$$\frac{\Gamma \vdash \mathcal{U}\langle\mathsf{TRabort}\colon \pi\omega\rangle\phi \quad \Gamma \vdash \mathcal{U}\langle\mathsf{TRcommit}\colon \pi\omega\rangle\phi}{\Gamma \vdash \mathcal{U}\langle\pi\ \mathtt{bT;}\ \omega\rangle\phi} \tag{R6}$$

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:

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

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

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 a corresponding 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:

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

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

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 $[\![\mathsf{TRcommit}\colon \ldots]\!]$ is the same as the rule for an assignment within $[\cdot]$ (the same holds for all other programming constructs). Note that, even using the $[\![\mathsf{TRcommit}\colon \ldots]\!]$ modality, ϕ only has to hold at the end of the transaction, which is considered to be atomic.

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

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 these fields and array elements by references to their copies holding the new value. Note that, if an object field is referenced to which no new value has been assigned (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 change 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:

$$\frac{\mathit{\Gamma} \;\vdash\; \mathcal{U}\{\mathit{loc}' := \mathit{expr}'\}[\![\mathsf{TRabort}\!: \pi\omega]\!]\phi}{\mathit{\Gamma} \;\vdash\; \mathcal{U}[\![\mathsf{TRabort}\!: \pi\ \texttt{loc = expr;}\ \omega]\!]\phi} \tag{R12}$$

As usual *expr* has to be a logical term. The contents of all objects and arrays are persistent, so all the subexpressions such as $\mathit{obj}.a_1.\mathit{arr}[i].a_2\ldots$ in *expr* are replaced by $\mathit{obj}.a_1'.\mathit{arr}'[i'].a_2'\ldots$ in expr' (the prime denotes the copy of a reference). The first reference *obj* or *arr* (as in $\mathit{arr}[i].a$) 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 is changed into loc' 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 $\mathit{obj}.a'$ or $\mathit{arr}[i']$ is referred to in an update, but it's not known (i.e. there was no such value assigned in the preceding updates) then it is considered to be equal to $\mathit{obj}.a$ or $\mathit{arr}[i]$, 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 contents, 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:

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

The remaining rules for $[\![\mathsf{TRabort}\colon \cdot]\!]$ (i.e. for other programming constructs) are the same as for $[\cdot]$, and the remaining rules for $[\mathsf{TRabort}\colon \cdot]$ and $\langle\mathsf{TRabort}\colon \cdot\rangle$ are the same as if there was 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 transactions 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 a tedious work, but it should not be hard for the user—most of the steps can be done automatically, in fact the only place where user interaction is required is giving 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 $x \geq 2$ holds, i.e. we prove the formula $x \geq 2 \to [\![p]\!]x \geq 2$.

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 $x \geq 2$ (ϕ) 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.

$$x \geq 2 \ \vdash \ \{x := 3\} x \geq 3 \tag{4}$$
$$x \geq 3 \ \vdash \ [\![\texttt{if(x<10)}\lambda\texttt{\{}\beta\texttt{\}}]\!] x \geq 2 \tag{5}$$
$$x \geq 3 \ \vdash \ \langle\alpha\rangle \mathit{true},\ [\texttt{if(x<10)}\lambda\texttt{\{}\beta\texttt{\}}] x \geq 3 \tag{6}$$

where:

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

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 descendent. 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:

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

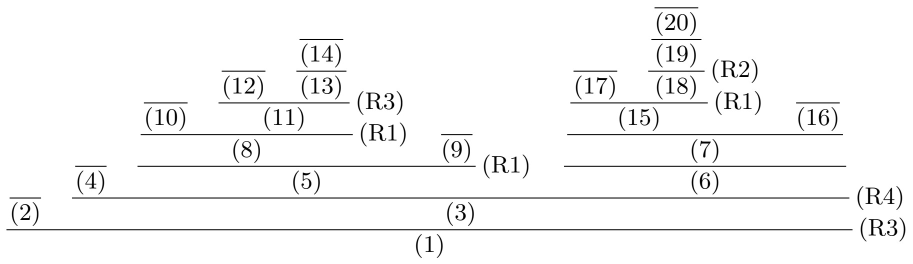


Figure 1: The proof from Example 1

Sequent (16) is valid by contradiction in the descendent. 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 descendent. Applying the assignment rule to (18) gives:

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

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 the following formula:

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

Note that this formula is not provable.

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

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

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

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

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

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

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

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

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:

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

Applying the exit transaction rule to (8) gives:

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

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

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

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

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:

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

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

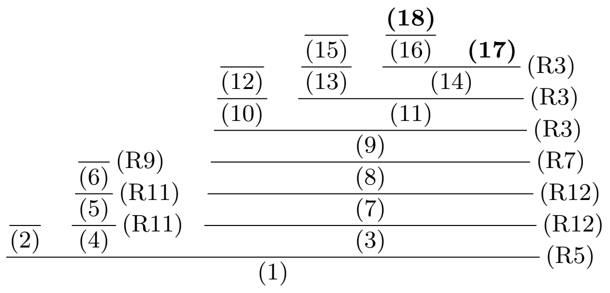


Figure 2: The proof from Example 2

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:

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

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

Sequent (16) is reduced to:

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

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 two open proof goals.

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

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The

KeY system: Integrating object-oriented design and formal methods. 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 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

[2] Bernhard Beckert. A dynamic logic for the formal verification of JAVA CARD programs. In I. Attali and T. Jensen, editors, *JAVA on Smart Cards: Programming and Security. Revised Papers, JAVA CARD 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[3] 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.

[4] 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.

[5] 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.

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

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

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

[9] KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.

[10] 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.

[11] Wojciech Mostowski. Rigorous development of Java Card applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002. `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.

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

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

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

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

[16] K. Trentelman and M. 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.

Java Card Tools for Together Control Center

Wojciech Mostowski

Abstract

This is a description of the Java Card Tools package for Together Control Center. The package supports the development of Java Card applets (writing, compiling, installing, testing, etc.) inside the Together Control Center CASE tool.

1 Introduction

Java Card technology [2] provides means to program Smart Cards using a subset of the Java language. Nowadays there is a great variety of different Java Card devices/platforms available on the market. Although they all have to conform to the Java Card specification, they usually differ slightly from each other. It's very common that each Java Card device producer provides its own Java Card development environment (it can be a complete Integrated Development Environment or just a set of tools/scripts/compilers). None of those tools provide a uniform, high level, UML supported environment for creating Java Card applications. Some of those tools are not very user friendly either. The Java Card Tools package tries to solve those inconveniences. It is an extension of a commercial CASE tool to provide support for different Java Card devices/platforms. The main goal is to provide a push-button technology inside the CASE tool that makes Java Card development easy and uniform for different kinds of Java Card devices and overcome vendor specific Graphical User Interfaces at the same time. The CASE tool in question is Together Control Center (later referred to as TogetherCC) from TogetherSoft [8]. There are two reasons for this choice. First of all TogetherCC is a state-of-the-art CASE tool with excellent UML support and open architecture, which makes writing extensions (in Java) an easy task. Second TogetherCC is the CASE tool used (as a base) in the KeY system. The KeY project [1, 5] aims at integrating object-oriented design with formal methods. The target language for the KeY project is Java Card due to its relative simplicity. So Java Card Tools can be seen as an extra support for the KeY system.

In Section 2 we describe the two Java Card platforms that are currently supported by Java Card Tools in detail. Section 3 presents the capabilities of Java Card Tools and describes an example usage. Section 4 gives some of the implementation details, Section 5 gives some directions for future extensions and finally Section 6 gives some pointers to the Java Card Tools documentation and download web site.

2 Java Card Platforms Supported

Currently there are two Java Card platforms/kits supported by Java Card Tools, however, extending it to support other platforms is possible and the Java Card Tools architecture makes writing extensions a relatively easy task. In the following subsections we describe the supported platforms in more detail.

2.1 Sun Java Card Development Kit

Sun Java Card Development Kit (jcdk in short) is a reference implementation of the Java Card development kit and tools. On its own the kit does not operate on any real Java Card devices, but such a device can be simulated or emulated by the kit tools. The package includes:

- Class file verifier and converter (`converter` tool), which makes sure that a given Java Card applet complies to Java Card language restrictions and creates a `.cap` file suitable for downloading to a Java Card device.

- `jcwde` tool (Java Card Workstation Development Environment), which is a simple Java Card simulator (e.g. it does not allow saving the state of a smart card between subsequent runs).

- `cref` tool (C reference implementation of a Java Card environment). This tool serves as a fully operational Java Card emulator. It operates on Java Card EEPROM images and allows saving a smart card's state after each session in form of such an EEPROM image.

- `apdutool`, which is used to send Application Protocol Data Units to a simulated/emulated Java Card. APDUs are the only means of communication between a smart card and the host application/system.

2.2 Dallas Semiconductor Java Powered iButtons

Dallas Semiconductor Java Powered iButton [4] is a Java Card device embedded in a small button (buttons are more durable than normal smart cards). iButtons implement Java Card API version 2.0. A development environment for iButtons (iB-IDE) is provided free of charge and can be downloaded from the web [3]. iB-IDE provides the following tools and functionality:

- an integrated development environment for creating both Java Card applets and host applications. Skeleton code can be generated for both with the help of a project wizard.

- an APDU sender, which provides low level access to iButtons: downloading and removing applets from iButtons, changing iButton settings (e.g. iButton password), sending arbitrary APDU packets to iButton, etc.
- iButton emulator, which can be used for testing applets before they are downloaded to the real iButton.

iB-IDE is built on top of low level libraries with well documented API, which makes it possible and relatively easy to reuse the libraries and build a new set of tools to operate on iButtons.

3 Support for Java Card Development

Java Card Tools is a uniform front-end to the tools and libraries described above. Since the tool set is embedded in TogetherCC it is possible to use the full support of the CASE tool and operate with Java Card devices and emulators at the same time.

The support for Java Card can be divided into two main parts:

- Java Card patterns (code skeletons),
- user friendly, easily accessible commands and tools to test applets by means of an APDU sender, which is used to "talk to" Java Card devices (either emulated/simulated or real). Those commands also include helper commands to prepare and download applets to Java Card devices.

3.1 Java Card Patterns

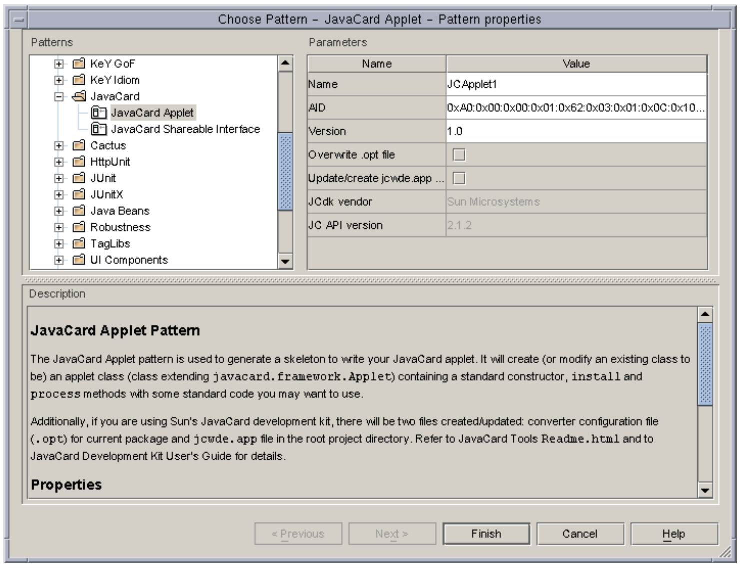
There are two patterns available at the moment, namely *Java Card Applet* and *Java Card Shareable Interface*. The latter is only available when Java Card API version 2.1 or higher is used (i.e. it's not applicable when iButtons are used).

As an example, Figure 1 shows the dialog of the *Java Card Applet* pattern. After applying this pattern to the project the skeleton code for a Java Card applet is created as well as some files necessary for the development kit used (in this case Sun jcdk). The skeleton applet code generated by the pattern looks as follows:

```
/* Generated by Together */

package testapplet;

import javacard.framework.*;
public class JCApplet1 extends Applet {
    protected JCApplet1(){
        // Write init code here
        register();
    }
```



Name	Value
Name	JCApplet1
AID	0xA0:0x00:0x00:0x01:0x62:0x03:0x01:0x0C:0x10...
Version	1.0
Overwrite .opt file	☐
Update/create jcwde.app ...	☐
JCdk vendor	Sun Microsystems
JC API version	2.1.2

JavaCard Applet Pattern

The JavaCard Applet pattern is used to generate a skeleton to write your JavaCard applet. It will create (or modify an existing class to be) an applet class (class extending `javacard.framework.Applet`) containing a standard constructor, `install` and `process` methods with some standard code you may want to use.

Additionally, if you are using Sun's JavaCard development kit, there will be two files created/updated: converter configuration file (`.opt`) for current package and `jcwde.app` file in the root project directory. Refer to JavaCard Tools `Readme.html` and to JavaCard Development Kit User's Guide for details.

Properties

Figure 1: *Java Card Applet* pattern

```
    /**
     * @param bArray array with initialisation data
     * @param bOffset offset into this array
     * @param bLength length of the data
     */
    public static void install(byte[] bArray,
                               short bOffset, byte bLength){
        new JCApplet1();
    }

    /**
     * @param apdu the incoming apdu packet to process
     */
    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
        }
    }
}
```

The *Java Card Shareable Interface* pattern simply creates an empty Java Card shareable interface like the following:

Sun JAVA CARD Development Kit:

- Display JAVA CARD environment info
- Rediscover JAVA CARD environment info
- Converter tools…
    - Create/update `.opt` file for this package/applet
    - Run converter for this package/applet
- Jcwde tools…
    - Create/update `jcwde.app` file
    - Start APDU sender
- Cref tools…
    - Create virtual card EEPROM image
    - Start APDU sender

Dallas Semiconductor iButtons:

- Display JAVA CARD environment info
- Rediscover JAVA CARD environment info
- JiBlet…
    - Build JiBlet for this applet
- Applet administration…
    - Get iButton information
    - Load this applet to iButton
    - List applets installed on iButton
    - Remove this applet from iButton
    - Remove applet from iButton by name
    - Master erase iButton
    - Set new password for iButton
- APDU sender…
    - Start APDU sender for this applet
    - Start APDU sender by name

Figure 2: JAVA CARD Tools pop-up menu structure

```
/* Generated by Together */

package testapplet;

import javacard.framework.*;
public interface MyInterface extends Shareable {
}
```

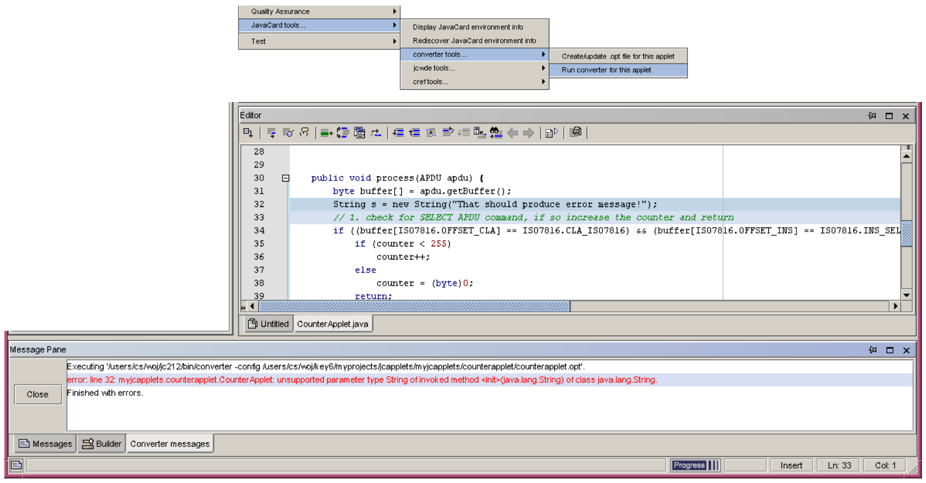
3.2 Installing and Testing Applets

The remaining JAVA CARD Tools functionality is accessed through TogetherCC's class diagram context (pop-up) menu (*JAVA CARD tools…* menu group). Figure 2 shows the full structure of this menu for both Sun JAVA CARD Development Kit and Dallas Semiconductor iButtons. Two menu commands are always present regardless of the JAVA CARD kit used, namely *Display JAVA CARD environment info* and *Rediscover JAVA CARD environment info*. The first one displays all the necessary information about the JAVA CARD kit that is currently used and the second one reconstructs this information in case the user changed the kit. In the following we demonstrate how some of the other commands listed in Figure 2 are used for JAVA CARD applet installation, testing and management.

Preparing the Applet

Suppose a JAVA CARD applet is ready for testing at some point. The first thing that has to be done is applet compilation. The standard TogetherCC tools are used for that. After that, the compiled `.class` file(s) need to be converted to a proper format. In case of Sun's jcdk that is a `.cap` file. To create a `.cap` file one needs to invoke *Run converter for this applet* command from the JAVA CARD Tools pop-up menu. The messages from the converter are displayed in TogetherCC's message windows with 'jump to error location' feature. In the example shown in Figure 3 a JAVA CARD applet uses a forbidden class `String`. By clicking on a message the source of the error is displayed in the editor window.

When JAVA Powered iButton is used then the JiBlet file (`.jib`) needs to be built by invoking *Build JiBlet for this applet* command from JAVA CARD Tools menu. It works almost in the same way as the `converter` tool.



```
    public void process(APDU apdu) {
        byte buffer[] = apdu.getBuffer();
        String s = new String("That should produce error message!");
        // 1. check for SELECT APDU command, if so increase the counter and return
        if ((buffer[ISO7816.OFFSET_CLA] == ISO7816.CLA_ISO7816) && (buffer[ISO7816.OFFSET_INS] == ISO7816.INS_SEL
            if (counter < 255)
                counter++;
            else
                counter = (byte)0;
            return;
```

```
Executing '/users/cs/woj/jc212/bin/converter -config /users/cs/woj/key6/myprojects/jcapplets/myjcapplets/counterapplet/counterapplet.opt'.
error: line 32: myjcapplets.counterapplet.CounterApplet: unsupported parameter type String of invoked method <init>(java.lang.String) of class java.lang.String.
Finished with errors.
```

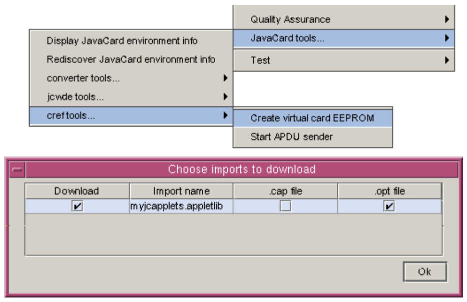
Figure 3: The `converter` tool

Testing Applets with Sun Jcdk Tools

After a `.cap` file for an applet is created, we can create a card's EEPROM image with that applet and run JAVA CARD emulator on the EEPROM image. To create an EEPROM image the *Create virtual card EEPROM image* command has to be invoked from the *Cref tools* submenu. If the applet uses some additional libraries, which are not part of the current package the user is asked whether those libraries should be included in the card's EEPROM image (that's usually the case), see Figure 4. Then an appropriate script is invoked and the EEPROM image is created with all the necessary messages displayed in TogetherCC's message window.

After the EEPROM image is created we can start an APDU sender for an emulated card. This is done by invoking *Cref tools/Start APDU sender* command. A window like the one in Figure 5 is displayed.

By pressing the 'Power up' button the `cref` emulator is started up. Then an applet can be selected on a virtual card by pressing the 'Select applet' button. Then any APDU packet can be constructed and sent to the emulator. The results will be displayed in the response panel. After pressing 'Power down' the current EEPROM image of the emulated card is saved, so that it can be used again for further testing. Pressing 'Exit' closes the APDU sender window (and also makes sure that the EEPROM image is saved).


Quality Assurance
JavaCard tools...
Test
Display JavaCard environment info
Rediscover JavaCard environment info
converter tools...
jcwde tools...
cref tools...
Create virtual card EEPROM
Start APDU sender
Choose imports to download
Ok

Download	Import name	.cap file	.opt file
✓	myjcapplets.appletlib	☐	✓

Figure 4: Creating a card EEPROM image

Another option is to test the applet using the `jcwde` card simulator. It works in a similar way to `cref` except that the EEPROM image is not used and therefore does not have to be created.

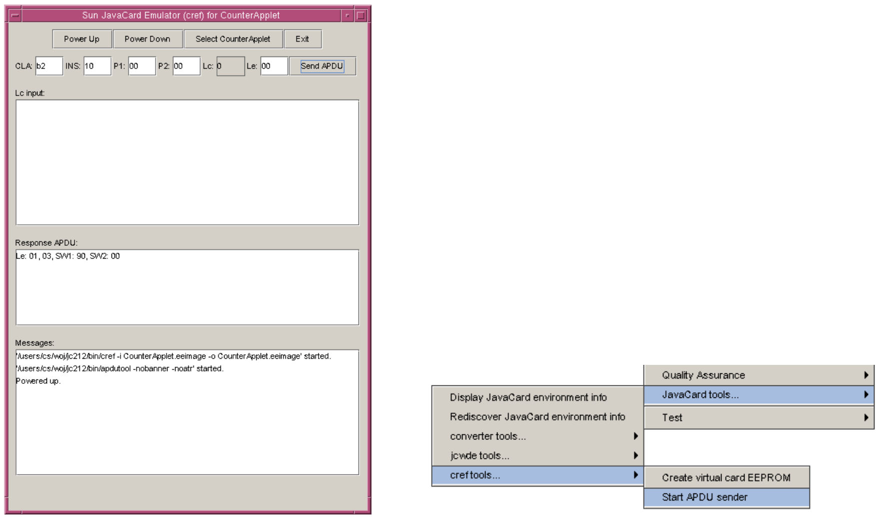
Testing Applets on iButtons

Once a JiBlet file is created for an applet, it can be downloaded to the iButton. This is done by invoking *Load this applet to iButton* command from the *Applet administration* submenu. If there is more than one iButton attached to the computer the user will be asked to choose one for downloading the applet. After successful download an APDU sender can be started to communicate with the iButton. This is done by invoking *Start APDU sender for this applet* command. exactly the same window as described earlier pops up (see Figure 6). Again 'Power up' button should be pressed to initialise the iButton, then the given applet can be selected by pressing 'Select applet' and then the user can start sending arbitrary APDUs to the applet and watch the responses.

In the current version of Java Card Tools there is no possibility to operate on an emulated iButton, only real iButtons. This is due to the lack of API documentation for the iButton emulator library.

Other iButton commands. There are a number of other, very useful commands in the *Applet administration* submenu for iButtons. They allow the following things:

- listing all the applets installed on an iButton,
- removing applets from iButtons,
- getting full iButton device information (ID string, firmware version, free memory, etc.),
- master erasing an iButton,
- changing the access password for an iButton.



CLA: b2 INS: 10 P1: 00 P2: 00 Lc: 0 Le: 00

Lc input:

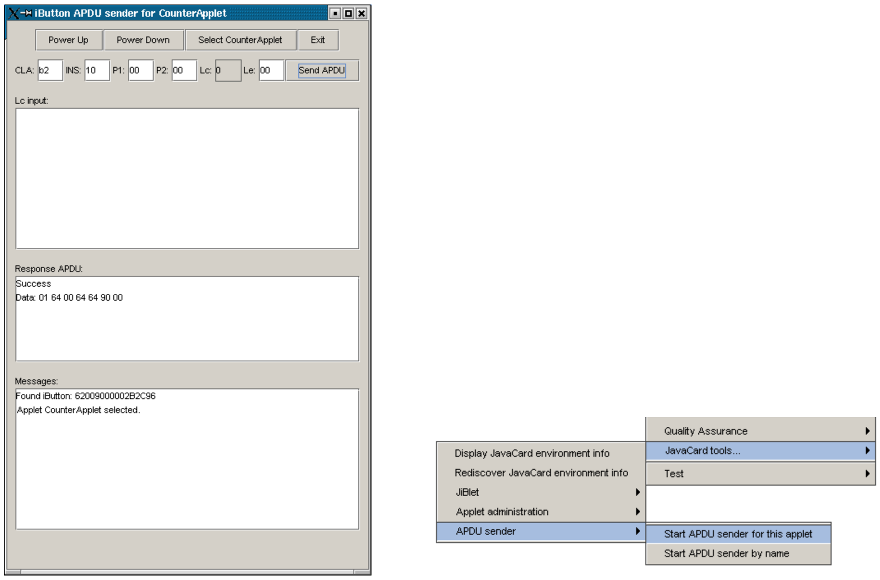
Response APDU:

```
Le: 01, 03, SW1: 90, SW2: 00
```

Messages:

```
'/users/cs/woj/jc212/bin/cref -i CounterApplet.eeimage -o CounterApplet.eeimage' started.
'/users/cs/woj/jc212/bin/apdutool -nobanner -noatr' started.
Powered up.
```

Figure 5: APDU sender window running `cref` emulator


iButton APDU sender for CounterApplet
Power Up
Power Down
Select CounterApplet
Exit
Send APDU
Quality Assurance
JavaCard tools...
Test
Display JavaCard environment info
Rediscover JavaCard environment info
JiBlet
Applet administration
APDU sender
Start APDU sender for this applet
Start APDU sender by name

CLA: b2 INS: 10 P1: 00 P2: 00 Lc: 0 Le: 00

Lc input:

Response APDU:

```
Success
Data: 01 64 00 64 64 90 00
```

Messages:

```
Found iButton: 62009000002B2C96
 Applet CounterApplet selected.
```

Figure 6: APDU sender window for an iButton

4 A Few Words About the Implementation

As already mentioned JAVA CARD Tools is just a front-end to a set of tools and libraries. As TogetherCC provides an open JAVA API, writing plug-ins and extensions is a relatively simple task—most of the internals of TogetherCC are available to the programmer. JAVA CARD Tools provides their support by extending TogetherCC's class diagram pop-up menu and adding new patterns to the TogetherCC pattern collection. To interface the tool set with Sun JAVA CARD Development Kit, processes running required tools (`apdutool`, `cref`, . . .) are started inside TogetherCC and the messages are passed to and from appropriate windows (TogetherCC message window, APDU sender window, etc.). The iButton support and communication is provided by the JiBlet JAVA library (`JiB.jar`) and iButton 1-Wire JAVA library (`OneWireAPI.jar`). Operating and communicating with iButtons is implemented by a few simple calls to those libraries. Both of those libraries are provided with iButton Integrated Development Environment free of charge.

JAVA CARD Tools package relies on TogetherCC's API, however there are no obstacles to make the package work with another CASE tool as long as the tool provides the necessary functionality and API for JAVA CARD Tools to function properly (e.g. access to a project's class path, access to the tool's editor, etc.). Also, as mentioned earlier, the JAVA CARD Tools package is easily extensible to support other JAVA CARD development kits and devices. For example, since most of the APDU sender functionality is independent of the device used, to enable sending APDUs to a JAVA CARD device with APDU sender it is basically enough to write a JAVA class that will communicate with the device and then just plug it in to the APDU sender class through a very simple interface.

5 Possible Extensions

One of the things that should be considered as a future extension is a generic support for any JAVA CARD platform that implements Open Card API [7]. The menu structure could be improved by abstracting the common commands for different development kits (e.g. *Prepare the applet for download*). Such menu abstraction would make the menus more uniform, independent of the actual JAVA CARD platform and easier to use for unexperienced users. On the other hand, however, experienced users familiar with a given development kit may actually prefer the concrete version of the menu to have more control over the actual tools they invoke with menu commands.

Furthermore some small extensions are possible, like having a set of an appropriately formed APDUs for all commands predefined in an applet to choose from in APDU sender window, etc. Also some more useful JAVA CARD patterns and code skeletons can be added to the pattern collection.

6 Further Information

A detailed User Guide and Installation Guide in HTML format can be found in Java Card Tools package as well as on the package web site [6]. The package itself (currently version 2.0—`jctools-2.0.zip`) can be downloaded from this page too.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. 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 2002, Proceedings*, volume 2306 of *LNCS*, pages 327–330. Springer, 2002.

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

[3] iB-IDE homepage. `http://www.ibutton.com/iB-IDE/`.

[4] iButton homepage. `http://www.ibutton.com/`.

[5] KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.

[6] Wojciech Mostowski. Java Card Tools for Together Control Center webpage. `http://www.cs.chalmers.se/~woj/javacard/`.

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

[8] TogetherSoft homepage. `http://www.togethersoft.com/`.