A programming and a modelling perspective on the evaluation of Java Card implementations[1]

Pieter H. Hartel and Eduard de Jong
phh@ecs.soton.ac.uk and Eduard.deJong@Sun.COM

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2000-8
August 16 2000

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

[1]This work was supported by Sun Microsystems Inc.

A programming and a modelling perspective on the evaluation of Java Card implementations[†]

Pieter H. Hartel[‡] and Eduard de Jong[§]

August 17, 2000

Abstract

Java Card Technology has provided a huge step forward in programming smart cards: from assembler to using a high level Object Oriented language. However, the authors have found some differences between the current Java Card version (2.1) and main stream Java that may restrict the benefits of using Java achievable in smartcard programming. In particular, efforts towards evaluating Java Card implementations at a high level of assurance may be hampered by the presence of these differences as well as by the complexity of the Java Card VM and API. The goal of the present paper is to detail the differences from a programming and a modelling point of view.

1 Introduction

With Java [1] Card Technology smart cards can be programmed in Java enjoying the benefits of object orientation. Both the card operating system as well as more application specific programming can be done in Java. Allowing a clearer distinction between the service and application layers in card software than previously possible. The second major advance of the Java Card VM is therefore the support for card applets. These applets take care of all application specific processing in a structured, efficient and secure manner. Moreover, card applets are downloadable and provide the opportunity to dynamically manage the services provided by a card. Thirdly, the Java Card API offers a model for controlled object sharing between applets. Finally, there is a special Java Card runtime library API, designed specifically for smart cards. It includes support for basic cryptographic routines (DES and RSA). There is no need to support say a windowing system on a smart card.

To reflect the limited computing resources inherent to smart cards, the Java Card VM and API impose restrictions. For example, there is no support for threads, garbage collection, or real numbers. While the standard word size for Java is 32 bits, for the Java Card VM this is 16 bits. Instead of relying on middle ware products, the Java Card API includes a simple transaction facility built in the VM. It also includes an interface to the ISO 7816-4 standard for the format of communication between smart card and terminal. Using this rather low-level protocol in Java is somewhat cumbersome, but Java Card applications are fully compatible with legacy terminals. Without this compatibility, an evolutionary approach for introducing Java Card technology into the market place would not work. Finally, Java Card implementations do not provide generic auditing facilities, which makes it difficult to evaluate the effectiveness of the Java card security mechanisms. Instead, it is left to the Java Card application programmers to ensure that appropriate logging information is maintained.

The already wide adoption of Java Card technology has shown that it has much to offer to the smart card community. Using Java makes it possible to deploy up-to-date software engineering techniques, ranging from object oriented design to formal methods. Gaining experience while deploying Java Cards has revealed the inevitable flaws in a first generation technology; and enhancements and additional features are being proposed by the user community. This paper evaluates the current specification of the Java Card technology to support its further development as the programming environment of choice for smart cards.

Offering programming facilities for smart cards, and adding new ones, places responsibilities on the implementors and users of Java Card systems, to maintain appropriate levels of trust. Responsibilities not common to the world of programmers at large. The goal of this paper is to explore possible avenues of bringing these two worlds closer, with an emphasis on programming patterns and formal modelling.

[†]This work was supported by Sun Microsystems Inc.

[‡]Dept. of Electronics and Computer Science, Univ. of Southampton, UK, Email: phh@ecs.soton.ac.uk

[§]Sun Microsystems, Palo Alto, CA 94043 USA, Email: Eduard.deJong@Sun.COM

2 Related work

A number of reports in the open literature provide evaluations of the Java Card specification.

Oestreicher [14] discusses the Java Card Transaction mechanism and proposes a number of improvements. Montgomery and Krishna [11] expose a potential security problem in the Java Card model for object sharing, and give guidance on how to avoid the problem. Oestreicher and Krishna [15] suggest how the Java Card persistence model may be improved. Rose and Rose [19] comment on the lack of on-card byte code verification and propose a solution. We list and evaluate these and other issues.

Formal modelling of Java Card aspects has been considered by: Denny and Jensen [3], Lanet and Requet [10], Motré [12], Posegga and Vogt [17], and Reid and Looi [18]. For a comprehensive discussion of these papers please refer to our earlier paper [8].

3 Methodology

The Common Criteria for IT Security Evaluation [16] require the presentation of formal models of IT systems for evaluation at the highest assurance level. It is possible to develop such formal models after the fact. However this is not ideal since the modelling activity is sure to uncover hitherto unknown problems in the design and implementation of the actual IT system. A good example is provided by Bertelsen's work on the specification of the JVM [1], and the resulting list of errata to the official Sun documentation.

A more profitable approach to evaluation at the highest assurance levels is to consider formal modelling as an integrated part of the software development process. This would ensure that the system under development can actually be formalised effectively. By effective we mean that the models are sufficiently clear and concise to make them useful for reasoning, whilst assuring that the models are a sufficiently accurate abstraction of the IT system. In an ideal world one would use formal methods throughout the design and implementation process. In practice this may not achievable for reasons of costs, increased production times, or simply lack of skills on the part of the engineering team. However, if the stakes are sufficiently high, such as in the safety critical software industry, the use of formal methods is the norm.

As a compromise we favour a structured interaction between the engineering team and a team of formal methods specialists. Both teams would be in a continued dialogue, with proposals made by one team being reviewed by the other. This would ensure that implementation considerations and modelling issues are both addressed right from the start.

For those critical of formal methods, we should like to point out that the issues most likely to cause problems to modelling and reasoning are precisely the same issues that would be troublesome to the engineering team. Examples include issues that make a system difficult to understand, that cause complex interactions between supposedly independent components, or that make testing a nightmare. Using formal methods is a good way to identify the important issues early in the software life cycle.

To build a formal model of a system is the same as to express the system in a different, more abstract and mathematical way. A particular concept in a system may be represented incorrectly in the model, or may not be found easy to understand. Either case may mean that the concept is complex and difficult to explain, and therefore likely also difficult to implement correctly. Programming, even in a high level language, is still a relatively low-level activity, requiring the programmer to keep an eye on a considerable amount of, often dispersed, details. By contrast, modelling is a high level activity, working with reasonable abstractions within a limited scope. For example in most models it is reasonable to assume that an unlimited amount of memory is available. The programmer might also make this assumption but would then additionally have to build a garbage collector to support it.

To make matters more concrete we focus on a number of aspects of Java Card implementations. A smart card is not a PC; resource constraints and security considerations require special attention. In the current version of the Java Card 2.1 specification this has lead to a significant number of changes and/or additions to the Java language and the API. We reflect on these changes and additions in Section 6, with a view to reduce their number. A specific problem we have encountered is that often a design/implementation seems to require a particular feature to achieve one objective and a different feature to achieve another. However, on further study it may appear that both objectives may be achieved via the addition of a single, somewhat different, feature, thus reducing the complexity of the system. Our recommendations for the process of additions and changes are:

- stimulate reflection on a proposal (additions, changes etc);
- encourage generalisation of the proposal, perhaps at the expense of some efficiency;
- require capitalisation on the "investment" as much as possible, i.e. consider how a feature might be used to achieve other objectives as well;
- assess all the potential interactions between the newly proposed feature and existing features;
- ask for a second opinion, i.e. find ways of viewing

the proposal from alternate angles, for which formal modelling is appropriate.

In the context of Java language and API changes or extensions we can make these recommendations more practical. A useful course of action is to investigate whether a desired feature can be expressed in some way in terms of features already provided. For example one could try to write a transaction facility in Java, to discover what is the essence of what is missing in order to make it work. This would focus on the missing essential feature. In a parallel modelling activity one might try to capture the semantics of the missing feature and add it to the semantic model of the existing system. We will give an example of this approach in our modelling case study of Section 5.

Java Card programmers might be less concerned with the principles of language and API addition or extension mechanisms rather than with the practice. Therefore, we also present a case study illustrating some of our observations on a simple example. This is the subject of the next section.

4 Programming case study

To illustrate the issues in programming a Java Card applet consider as an example the code fragment of Figure 1. This is taken verbatim from the Java Card 2.1 Application Programming Interface [20, Page 40]. The line numbers have been added for ease of reference.

4.1 APDU based communication

Communication between the card and the terminal must be expressed in terms of byte oriented APDU commands. This problem arises merely because the Java Card framework addresses the specific problem of using Java to write smart card applications. Card applications are fundamentally small server programs that rely on communication with an inherently limited bandwidth and a severely restricted packet size. The usual programming abstraction of communication as an unlimited stream is hard to maintain in the card API in its full generality. The Open Card framework [2] addresses the complementary problem for terminals. The two frameworks use the standardized APDU communication format as an interface. The code fragment shows typical in card processing for this format.

The `process` method receives an `APDU` object to discover which command to process (line 4). However, the actual command information is not available in the `APDU` object, but needs to be acquired by the `apdu.getBuffer()` method call (line 6). This method returns a reference to a global buffer with the actual command data. Regardless of the requirements of the actual command, the data is offered as a raw array of bytes. This leaves the application programmer with the unenviable task of manually unmarshalling application data. For example the `class` byte is obtained by an array access (line 7). In keeping with the object oriented philosophy one would have preferred to write `apdu.cla`, or `apdu.cla()`, or in accordance with the Java style recommendations `apdu.getClassByte()`.

A second example of the difficulty in manually unmarshalling data is found at line 12. Here the byte at offset `ISO7816.OFFSET_LC` is an *unsigned* value. The masking operation, and the fact that in Java intermediate results of a computation are integers ensures that bytes in the range -128 .. -1 are mapped onto shorts in the range 128 .. 255. This level of detail is not something that one would like to burden the programmer with.

A third example of how tricky low level programming is can be found at line 34, where obviously by a cut and paste error, an assignment is made to `buffer[3]` instead of `buffer[2]`.

Not obvious from the coding example is the fact that sending and receiving APDU commands has some degree of protocol dependency: T=0 and T=1 do not always have the same view on the number of bytes sent or received.

A solution would be to create an all embracing framework that abstracts away from APDU commands, perhaps using a lightweight RMI style interface. Full RMI would be too expensive to implement on a smart card, it is too powerful and general purpose for smart cards, and RMI does not offer the security that is required. Work is in progress on a number of other solutions, for example the GemPlus Direct Method Invocation (DMI).

4.2 Type casts

Java is based on 32-bit words; the Java Card VM uses a mixture of 8, 16 and 32-bit semantics. The APDU communication is based on byte arrays. The stack contains 16 bit items. There is optional support for 32 bit integers. Since intermediate results from 8 or 16-bit calculations may require 32-bits, the Java language requires the programmer to state explicitly (by inserting appropriate type casts) where data may be lost.

Type casts are notoriously difficult to get right. Consider as an example the code at lines 13 and 16. One of the comparisons uses a type cast, and the other does not. The type cast is redundant here, because the intermediate result of the comparisons is of type `int`. By contrast the type cast in line 12 is required by the Java semantics, as the result of the expression on the right hand side is automatically an integer. Programmer action is required to explicitly cast an integer result into a short. This is a standard feature of Java. However, the problem arises because Java Card support for the `int` type is optional. Some Java Card implementations therefore would not be

```
 1   // The purpose of this example is to show most of the methods
 2   // in use and not to depict any particular APDU processing
 3
 4  public void process(APDU apdu){
 5    // ...
 6    byte[] buffer = apdu.getBuffer();
 7    byte cla = buffer[ISO7816.OFFSET_CLA];
 8    byte ins = buffer[ISO7816.OFFSET_INS];
 9    ...
10    // assume this command has incoming data
11    // Lc tells us the incoming apdu command length
12    short bytesLeft = (short) (buffer[ISO7816.OFFSET_LC] & 0x00FF);
13    if (bytesLeft < (short)55) ISOException.throwIt( ISO7816.SW_WRONG_LENGTH );
14
15    short readCount = apdu.setIncomingAndReceive();
16    while ( bytesLeft > 0){
17        // process bytes in buffer[5] to buffer[readCount+4];
18        bytesLeft -= readCount;
19        readCount = apdu.receiveBytes ( ISO7816.OFFSET_CDATA );
20        }
21    //
22    //...
23    //
24    // Note that for a short response as in the case illustrated here
25    // the three APDU method calls shown : setOutgoing(),setOutgoingLength() & sendBytes()
26    // could be replaced by one APDU method call : setOutgoingAndSend().
27
28    // construct the reply APDU
29    short le = apdu.setOutgoing();
30    if (le < (short)2) ISOException.throwIt( ISO7816.SW_WRONG_LENGTH );
31    apdu.setOutgoingLength( (short)3 );
32
33    // build response data in apdu.buffer[ 0.. outCount-1 ];
34    buffer[0] = (byte)1; buffer[1] = (byte)2; buffer[3] = (byte)3;
35    apdu.sendBytes ( (short)0 , (short)3 );
36    // return good complete status 90 00
37    }
```

Figure 1: The main method of a prototypical Java Card applet.

able to cope with the example if the `short` had been replaced by `int`.

It is probably more a matter of months than years before smart cards are sufficiently powerful to sustain full 32-bit applications. This would solve this problem.

4.3 Memory management

Java Card programmers cannot rely on the services of a garbage collector. Instead they are required to pre-allocate all storage, both in RAM and EEPROM. Reusing space is fully under control of the programmer. This creates similar memory allocation problems to what one would find in C programs, such as premature re-use of space, or space leaks.

The advantage of pre-allocation is that applets always have the required heap space available. However, they might still run out of stack frames.

Preallocation has two disadvantages. Firstly, it does not cope with transient data. Therefore the Java Card API provides a feature to make sure that pre-allocated RAM data is cleared appropriately.

Secondly, since all applets pre-allocate their store, it is not possible that one applet temporarily uses more than its share. Such scenarios might arise in particular when applets call upon other applets for some service. Over commitment has been a standard technique used in operating systems for many years, and it could have been used here with success.

With a garbage-collected heap, neither pre-allocation nor explicitly transient objects would be necessary. One argument has been raised claiming that transient objects in RAM are useful to maintain cryptographic session keys. However, from a security perspective, it is probably easier to spy out data in RAM than it is to spy out EEPROM. The relative strength of the memory technologies seems irrelevant as the more precious master keys are stored in EEPROM, from which the session keys are derived.

Potential solutions to the pre-allocation problem include using transacted memory [9], or using a moving garbage collector that migrates long-lived data from RAM to EEPROM.

The Java Card VM does not support finalizers, because it does not support garbage collection. However, not having finalizers is generally considered an advantage, because the presence of finalizers makes the meaning of Java programs dependent on the, asynchronous, behaviour of the garbage collector. Without finalizers there is no such dependency.

While a space conscious system, the current Java Card specification seems to have paid less attention to stack space requirements. First it has 16-bit words; secondly there is no limit on stack growth, since Java Card applets can be recursive. With recursion banned, which is certainly feasible for programs with the scope of a Java card applet, a tool could work out the maximum number of stack frames needed by an applet, which coupled with a maximum heap size as required by the programmer could yield a true deadlock free applet at least in terms of space requirements.

Exceptions in Java are objects, and in the JCRE (the Java Card Runtime Environment) an object has been pre-allocated for every exception that could be raised by a Java Card applet. Such exceptions are accessed via an index in a table, and they are raised using the `throwIt()` method, as illustrated at line 13 and 30 of the code fragment of Figure 1.

This mechanism is considered redundant, at least in its exposure to the card application programmer. Similar savings could have been achieved by allowing the VM storage allocator to cache the objects created for exceptions. Assuming that most applets throw far fewer exceptions than there are defined by the Java Card specification, this represents a significant saving. An implementation based on caching could be entirely transparent to the programmer, thus obviating the need to redefine that part of the API that deals with exceptions.

4.4 Concurrency

The innocent looking method call on line 35 represents an interesting problem because the `sendBytes` call may be asynchronous. This means that the data stored in the shared buffer must not be altered until the send operation has completed. There is no way for an applet of finding out whether the operation has actually completed. This is the only aspect of the Java Card specification that permits concurrency, as threads are not supported. Programming concurrent systems is harder than programming sequential systems, and not surprisingly modelling work on concurrent systems is harder than modelling sequential systems.

We believe that the small optimisation that may be present in some Java Card implementations by allowing an asynchronous send does not outweigh the disadvantage of having to cope with concurrency, and the possibility of differing semantics of an applet on different implementations of the VM.

5 Modelling case study

In this section we model an environmental constraint pertinent to smart cards known as “card tear”, which is the sudden removal of power from the processor. We study how it might interact with the normal operation and persistence in a Java Card implementation. Our model makes some simplifying assumptions, making it possible to learn

about card tear in an abstract setting that is not cluttered by detail. One would have to check of course that the lessons learned also apply to a real system. This is future work.

Nielson and Nielson [13] provide an excellent introduction to the methods and notation used. LETOS has been used to typecheck and execute our specifications [7].

Every Java thread has a method `uncaughtException`, which is called when the thread raises an exception that is not caught [4, Section 20.21.31]. In contrast, the Java Card VM takes the view than an uncaught exception is handled in an implementation defined way [22, Section 2.3.3.1]. This difference may be relevant for review, but is not considered here.

Most Java exceptions are synchronous, which means that they are raised as a result of the current computation. Exceptions are also precise [4, Section 11.3.1] in the sense that they are raised immediately a semantic constraint is violated. Java also has two asynchronous exceptions: `ThreadDeath` and `InternalError`, which may be raised at any time. As the Java Card specification does not allow threads, it does not allow `ThreadDeath`. The Java Card specification does permit raising `InternalError`, but leaves it up to the implementation to decide how to handle it [22, Section 2.3.3.1]. The intention is that Java Errors can only be caught if the situation is deemed recoverable. For now we will assume that an `InternalError` cannot be caught, and that it renders the card muted to avoid security problems. This effectively removes all asynchronous exceptions from the Java Card specification.

Standard Java does not provide persistence. Therefore a Java applet does not expect objects to be preserved between to runs of the applet. In contrast, the Java Card API does support persistence. Java Card applets retain some of their objects (those stored in EEPROM), but loose others (stored in RAM). To acknowledge this, the life time of the Java Card VM is deemed to be the same as the life time of the smart card on which it runs [21, Chapter 2]. The Java Card VM detects whether an applet has been interrupted by a power failure, and ensures that the persistent objects of the applet are in a consistent state upon restart of the applet.

This raises the question of how to model power failure and recovery. The most appropriate way of doing so seems to be to introduce an asynchronous exception `PowerFailure`. In the model, the Java object store is represented by a RAM, and an EEPROM which shadows the RAM. The programmer is responsible for choosing the moment at which to save RAM contents into EEPROM. Java programs are modelled by a small subset covering the essence of an applet, saving and storing RAM contents and the handling of exceptions. This is the subject of the following sections.

5.1 Syntax

Consider the smallest fragment of Java as shown below, which permits throwing and handling exceptions. The fragment offers just four statements: `throw` to raise an exception, `try...catch` to bind a statement to its exception handler, the increment statement ($\mathsf{v}++$, where v represents a program variable), and the method calls `save`(), and `restore`(). The (first) semicolon represents statement composition, ϵ represents an empty statement sequence.

$$\begin{array}{l}\mathsf{s} \equiv \texttt{throw}\ \mathsf{x} \mid \texttt{try}\{\mathsf{s}\}\texttt{catch}(\mathsf{x})\{\mathsf{s}\} \mid \\ \quad \mathsf{v}\,{+}{+} \mid \texttt{save}() \mid \texttt{restore}() \mid \\ \quad \mathsf{s}\ ;\ \mathsf{s} \mid \epsilon;\end{array}$$

Also define two exceptions (`ArithmeticException`, and `PowerFailure`), and an out-of-band value (normal) that indicates normal processing.

$$\mathsf{x} \equiv \texttt{ArithmeticException} \mid \texttt{PowerFailure} \mid \text{normal};$$

5.2 Transition relation

A machine to execute the statements (s) would also need the state of the RAM (r), and the state of the EEPROM (e). The RAM is modelled as a mapping from variables to numbers (data). The EEPROM is modelled as a copy of saved RAM contents.

$$\begin{array}{l}\mathsf{r} \equiv \{\langle \mathsf{v} \mapsto \mathbb{N}\rangle\};\\ \mathsf{e} \equiv \mathsf{r};\end{array}$$

The transition relation ($\rightarrow$) giving the natural semantics of the Java fragment has the type:

$$\rightarrow :: \langle \mathsf{s},\ \mathsf{r},\ \mathsf{e}\rangle \leftrightarrow \langle \mathsf{r},\ \mathsf{e},\ \mathsf{x}\rangle;$$

An increment statement stores a new value in RAM. The `save` operation stores the contents of the RAM in the EEPROM, and the `restore` operation recovers the RAM contents.

$$\begin{array}{ll}[{+}{+}] & \langle \mathsf{v}\ {+}{+},\ \mathsf{r},\ \mathsf{e}\rangle \rightarrow \\ & \langle \mathsf{r} \oplus \{\mathsf{v} \mapsto \mathsf{r}(\mathsf{v}) + 1\},\ \mathsf{e},\ \text{normal}\rangle;\\ [\texttt{save}] & \langle \texttt{save}(),\ \mathsf{r},\ \_\rangle \rightarrow \langle \mathsf{r},\ \mathsf{r},\ \text{normal}\rangle;\\ [\texttt{restore}] & \langle \texttt{restore}(),\ \_,\ \mathsf{e}\rangle \rightarrow \langle \mathsf{e},\ \mathsf{e},\ \text{normal}\rangle;\end{array}$$

Raising an exception is modelled by recording the exception in the third component of the result state.

$$\begin{array}{ll}[\texttt{throw}^1] & \langle \texttt{throw}\ \mathsf{x},\ \mathsf{r},\ \mathsf{e}\rangle \rightarrow \langle \mathsf{r},\ \mathsf{e},\ \mathsf{x}\rangle,\\ & \textbf{if}\ \mathsf{x} \neq \texttt{PowerFailure};\end{array}$$

A power failure additionally wipes out the RAM. This models the fact that the RAM contents is actually lost

when the power fails, and not when the power is restored. We could have decided to leave the RAM in an undefined state, which probably models real hardware more accurately, but this would represent a security risk. Memory remanence [6] might make it possible with some memory technology for some of the old contents to reappear when power is restored.

$$[\texttt{throw}^2]\ \langle \texttt{throw}\ \mathsf{x},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \{\langle \mathsf{v} \mapsto 0\rangle \mid \mathsf{v} \in \mathsf{domain}(\mathsf{r})\},\ \mathsf{e},\ \mathsf{x}\rangle, \quad \textbf{if } \mathsf{x} = \texttt{PowerFailure};$$

The `try...catch` statement can be executed in different ways, depending on whether an exception is raised, or whether the `try` clause has completely normally. The first possibility below applies when the `try` clause completes without raising an exception.

$$[\texttt{try}^1]\ \frac{\langle \mathsf{s_t},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_t}\rangle}{\langle \texttt{try}\{\mathsf{s_t}\}\texttt{catch}(\mathsf{x})\{\mathsf{s_c}\},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \text{normal}\rangle}, \quad \textbf{if } \mathsf{x_t} = \text{normal};$$

If the `try` clause has caused an exception and the current `catch` clause can handle it, the statements of the `catch` clause are executed.

$$[\texttt{try}^2]\ \frac{\langle \mathsf{s_t},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_t}\rangle,\ \langle \mathsf{s_c},\ \mathsf{r}',\ \mathsf{e}'\rangle \to \langle \mathsf{r}'',\ \mathsf{e}'',\ \mathsf{x_c}\rangle}{\langle \texttt{try}\{\mathsf{s_t}\}\texttt{catch}(\mathsf{x})\{\mathsf{s_c}\},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}'',\ \mathsf{e}'',\ \mathsf{x_c}\rangle}, \quad \textbf{if } \mathsf{x_t} \neq \text{normal} \wedge \mathsf{x_t} = \mathsf{x};$$

If the `try` clause has caused an exception and the current `catch` clause cannot handle it, the exception will be propagated to another, embracing handler.

$$[\texttt{try}^3]\ \frac{\langle \mathsf{s_t},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_t}\rangle}{\langle \texttt{try}\{\mathsf{s_t}\}\texttt{catch}(\mathsf{x})\{\mathsf{s_c}\},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_t}\rangle}, \quad \textbf{if } \mathsf{x_t} \neq \text{normal} \wedge \mathsf{x_t} \neq \mathsf{x};$$

Statement composition is handled in a similar way as described above. The execution proceeds differently, depending on whether the first statement causes an exception to be raised, or whether it completes normally.

$$[;^1]\ \frac{\langle \mathsf{s_1},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_1}\rangle,\ \langle \mathsf{s_2},\ \mathsf{r}',\ \mathsf{e}'\rangle \to \langle \mathsf{r}'',\ \mathsf{e}'',\ \mathsf{x_2}\rangle}{\langle \mathsf{s_1}\ ;\ \mathsf{s_2},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}'',\ \mathsf{e}'',\ \mathsf{x_2}\rangle}, \quad \textbf{if } \mathsf{x_1} = \text{normal};$$

$$[;^2]\ \frac{\langle \mathsf{s_1},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_1}\rangle}{\langle \mathsf{s_1}\ ;\ \mathsf{s_2},\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \mathsf{r}',\ \mathsf{e}',\ \mathsf{x_1}\rangle}, \quad \textbf{if } \mathsf{x_1} \neq \text{normal};$$

Finally, a statement may not be able to complete due to power failure. This is modelled by the rule below.

$$[\text{power}]\ \langle \_,\ \mathsf{r},\ \mathsf{e}\rangle \to \langle \{\langle \mathsf{v} \mapsto 0\rangle \mid \mathsf{v} \in \mathsf{domain}(\mathsf{r})\},\ \mathsf{e},\ \texttt{PowerFailure}\rangle;$$

The power axiom duplicates the object of the `throw`[2] axiom. The rule is applicable whenever any of the other 10 rules are applicable. The semantics has thus become non-deterministic. From the programmers point of view this means that any statement can be replaced by `throw PowerFailure`. This can be done any number of times.

The current draft of the SCSUG Smart Card Protection Profile [5, Section 3.2] specifies the same assumption as we have made here: "Power and Clock come from the terminal. These are not considered reliable sources". Interpreting unreliable as 'can happen at any time' translates directly into the use of non-determinism in our semantics.

5.3 Operational semantics

We are now able to put all the pieces together in a function S (below), which gives the semantics of an applet j.

An applet can never handle a power failure. Therefore, we require that there are no occurrences of `try{...}catch(PowerFailure){...}` in j. All other exceptions are required to be handled by the applet j itself, as is the case in standard Java.

Almost paradoxically, to allow the applet j to complete normally, we will try to execute it repeatedly. Each time a power failure occurs, execution is interrupted and then restarted, until finally j completes normally. This coincides with the view that the Java Card VM lives as long as the carrier smart card is operational. The repeated execution is modelled by wrapping applet j in a `try` statement as shown by the local definition of w in the semantic function S below.

The recursive definition of the wrapper w ensures that when the applet j is aborted by a power failure, the `catch` clause causes the whole process to be started again. If the applet j runs to completion, the `catch` clause is ignored and the wrapper w terminates.

Each run of j begins by restoring the RAM contents from the EEPROM, and ends either with a `PowerFailure`, or normal termination. The initial EEPROM and RAM map all addresses to zero.

$$\begin{aligned}\mathsf{S}[\![\mathsf{j}]\!] = \langle \mathsf{w},\ \mathsf{r},\ \mathsf{r}\rangle \to\ & \textbf{where}\\ & \mathsf{w} = \texttt{try}\{\texttt{restore}();\ \mathsf{j}\}\\ & \qquad \texttt{catch}(\texttt{PowerFailure})\{\mathsf{w}\};\\ & ;\ \mathsf{r} = \{\langle \mathsf{a} \mapsto 0\rangle \mid \mathsf{a} \in [0..]\};\\ & ;\end{aligned}$$

5.4 Example applet

We can now use the semantic function to trace the execution of a applet, for example the following applet j_1:

$$\mathsf{j}_1 = \mathsf{a}\ {+}{+}\ ;\ \texttt{save}();\ \mathsf{b}\ {+}{+};$$

The applet j_1 is not intended to do something useful, but it happens to count the power failures witnessed when trying to execute the $\mathsf{b}{+}{+}$ statement. The number will be reflected in the final state of the RAM and the EEPROM, which therefore may assume any final state as shown below:

$$\begin{array}{l}\mathsf{S}[\![\mathsf{j}_1]\!]\in\{\langle\{\langle\mathsf{a}\mapsto\mathsf{n}\rangle,\ \langle\mathsf{b}\mapsto 1\rangle\},\\ \{\langle\mathsf{a}\mapsto\mathsf{n}\rangle,\ \langle\mathsf{b}\mapsto 0\rangle\},\ \mathrm{normal}\rangle\mid \mathsf{n}\in[1..]\};\end{array}$$

Here is another, simpler example:

$$\mathsf{j}_2 = \mathsf{a}\ {+}{+}\ ;\ \mathsf{b}\ {+}{+};$$

As compared to j_1, this applet lacks the `save`. Therefore its behaviour is characterised by one of two possible outcomes:

$$\begin{array}{l}\mathsf{S}[\![\mathsf{j}_2]\!]\in\{\langle\{\langle\mathsf{a}\mapsto 1\rangle,\ \langle\mathsf{b}\mapsto 0\rangle\},\\ \{\langle\mathsf{a}\mapsto 1\rangle,\ \langle\mathsf{b}\mapsto 0\rangle\},\ \mathrm{normal}\rangle,\\ \langle\{\langle\mathsf{a}\mapsto 1\rangle,\ \langle\mathsf{b}\mapsto 1\rangle\},\\ \{\langle\mathsf{a}\mapsto 1\rangle,\ \langle\mathsf{b}\mapsto 0\rangle\},\ \mathrm{normal}\rangle\};\end{array}$$

Both assertions are can be proved by induction on the number of times `PowerFailure` is raised.

5.5 Properties

There are several useful properties that one might study for the given semantics. Of particular interest is the influence of the wrapper on the semantics of the applets j.

5.5.1 Preservation of termination

Firstly it would be desirable to prove that the wrapper preserves termination of applets.

The set of rules is compositional, so any derivation tree is finite, and thus all applets j terminate. Assume that there are $n >= 0$ power failures during the life time of the applet j. We now sketch a proof by induction on n that the wrapper preserves termination.

In the base case ($n = 0$) and by our requirement that all exceptions (except `PowerFailure` of course) are handled by j itself we have by rules [`restore`], [;^{1}] and [`try`^{1}]:

$$\frac{\langle\texttt{restore}(),\ \mathsf{r},\ \mathsf{e}\rangle\to\langle\mathsf{r}',\ \mathsf{e}',\ \mathrm{normal}\rangle,\quad \langle\mathsf{j},\ \mathsf{r}',\ \mathsf{e}'\rangle\to\langle\mathsf{r}'',\ \mathsf{e}'',\ \mathrm{normal}\rangle}{[]\ \langle\texttt{try}\{\texttt{restore}();\mathsf{j}\}\texttt{catch}(\texttt{PowerFailure})\{\mathsf{w}\},\ \mathsf{r},\ \mathsf{e}\rangle\to\langle\mathsf{r}'',\ \mathsf{e}'',\ \mathrm{normal}\rangle};$$

In the general case assume that there are $n > 0$ power failures during the life time of the applet. Then unfolding the recursive definition of the wrapper w by n times would give us the inductive case.

Here we have glossed over one issue: the power rule is always applicable. Therefore even in the base case the following derivation is valid:

$$[]\ \langle\texttt{try}\{\texttt{restore}();\mathsf{j}\}\texttt{catch}(\texttt{PowerFailure})\{\mathsf{w}\},\ \mathsf{r},\ \mathsf{e}\rangle\to\langle\{\langle\mathsf{v}\mapsto 0\rangle\mid\mathsf{v}\in\mathsf{domain}(\mathsf{r})\},\ \mathsf{e},\ \texttt{PowerFailure}\rangle;$$

To remedy this shortcoming of our model we make a fairness assumption, which states that a derivation may not begin with an application of the power rule, and which also rules out an application of the power rule immediately after the previous.

5.5.2 EEPROM preservation

A second property would establish that whatever is written to the EEPROM can eventually be read back. Ideally one would like to prove this preservation of information property in the deterministic setting, i.e without the rule power present. However, the proof would not carry over to the extended semantics, because the extension is not operationally conservative. Therefore one would have to re-prove the preservation property in the extended setting. This represents the cost of adding a feature (i.e. modelling power failure).

5.6 Modelling in Java

A natural question to ask is: What could have been achieved by attempting to model power failure in Java itself? To answer this we implemented the formal model by way of a Java 'simulator', a fragment of which is shown below. The fragment corresponds to a single unfolding of the wrapper w, with the sample applet j_1 from Section 5.4.

The `tear()` method simulates the non-deterministic choice of whether card tear should trigger the `PowerFailure` exception.

```
class PowerFailure extends Exception {
   PowerFailure() { super( ); }
}
```

The `power()` method actually raises the exception, after clearing the RAM. The result is that any statement of our sample applet j_1 is either executed or aborted, as required by the formal model.

```
try {
   if( tear() ) power() else restore() ;
   if( tear() ) power() else a++ ;
   if( tear() ) power() else save() ;
```

```
   if( tear() ) power() else b++ ;
} catch( PowerFailure e1 ) {
   try {
     if( tear() ) power() else restore() ;
     if( tear() ) power() else a++ ;
     if( tear() ) power() else save() ;
     if( tear() ) power() else b++ ;
   } catch( PowerFailure e2 ) {
     ...
   }
}
```

The simulator has been validated by using it to execute a number of rather trivial example applets, and by comparing the resulting states to those obtained by a proof from the formal model.

To return to the question raised at the beginning of this section, we believe to have shown that a lot can be achieved by modelling in Java. However some of the concepts required are not particularly obvious from a programmer's perspective, such as the recursive wrapper, or the non-deterministic choice. Some mathematical training is essential to apply these ideas. On the other hand, using an exception to model power failure, and clearing the RAM to enhance security could be ideas natural to the programmer.

The main difference between formal modelling and simulation in Java is that the latter activity does not support proofs. We believe that engineering and modelling skills should be present in an engineering team to achieve best results.

6 Comparison of the Java and Java Card specifications

Having presented two detailed case studies, we now give a comprehensive overview of the differences between the Java and Java Card specifications. This section is best read with the relevant Java Card documentation [20, 22, 21] available.

The Java Card specification is based on a subset of Java and its APIs. The subset was chosen primarily to cope with resource constraints. However, it is also an extension of the subset, with the extensions to provide additional smart card specific functionality. Table 1 lists the exclusions by the subset and the additions by the extension. The table is provided by way of summary, we will not discuss it entry by entry. Instead we will discuss the issues in an appropriate context.

6.1 Software Engineering Aspects

Some aspects of using the programming environment we believe need to be improved in future specifications:

- Some of the limitations and exclusions imposed in the Java Card specification (i.e. memory size) cannot be enforced statically. This makes it more difficult for the programmer to test and debug Java Card applets, because they may not hit the restriction or limitation on any development environment.
- The Java Card specification encourages a low level programming style that that does not sit comfortably with mainstream object oriented analysis and design.
- Java and Java Card documentation sometimes express high-level concepts in low-level terms. For example the Java 2 security model talks about stack inspection, and the Java Card security model discusses which byte codes access objects. Java programmers should be able to understand security in Java terms.
- Java Card vendors have considerable freedom in extending/revising their Java Card versions. This may hamper portability at the level of CAP and Export files (not at the level of class files).

Also, the consequences of the language specification as a superset of a subset of Java we have recognized as:

- Java API's or applets cannot be ported easily to Java Card implementations, and, vice versa, Java Card applets cannot be developed easily using a generic Java IDE.
- As a naturally evolving programming environment new features will be added and old features will disappear; addressing the inherent legacy problem will be harder.
- Main-stream Java programmers, in addition to learning the card application framework API, will have to be specially trained to use the card specific extensions.

Summarising, there is less portability between Java and Java Card implementations, in terms of software engineering techniques, than the authors consider desirable.

6.2 Object life times

The Java Card specification takes the useful view that the life time of applets spans terminal sessions. This means that the objects held on to by the applets may also live forever. Therefore, objects are by default persistent, as is indeed required in smart cards. An applet itself is an object, which is created by the static `install` method of the applets defining class. The JCRE implementation arranges for a context switch when calling the `install` method to satisfy the ownership relation.

subset	[20]	superset of the subset
no class files	1.2	cap + export files
no dynamic class loading	2.2.1.1	applet installer
no security manager	2.2.1.1, 3.4	contexts
no garbage collection	2.2.1.1	
no finalization	2.2.1.1	
no threads	2.2.1.1	asynchronous writes using APDU buffer
no cloning	2.2.1.1	
no native code in applets	2.2.1.2	
subset of Java visibility rules	2.2.1.1, 5.4.1	two kinds packages: library and applet, with different visibility rules
no multidimensional arrays	2.2.1.3	
no char, double, float, or long types	2.2.1.3	
only a small part of the Java API	2.2.1.4	Java Card API
no reflection, no class Class	2.2.1.4	
type int is optionally supported	2.2.3.1	
limited number of numbers of classes, interfaces, methods, fields, array elements, and cases.	2.2.4.1	
only initialisation of static fields that are of primitive type or array of primitive type	2.2.4.5	
no long, float, double, and monitor byte codes, 73 in total	2.3.2.1	All different byte codes
no checked exceptions, only some runtime exceptions and errors	2.3.3	SystemException with reason codes
	4.2	Application Identifier (AID)
no name based linkage	4.3.6	token based linkage
no class based linking and loading	4.4	loading is package based
no primitive final fields in the constant pool	4.4	primitive final fields are inlined
binary compatibility not fully supported by the off-card byte code verifier, extra restrictions	4.4,4.5	major and minor version numbers
	5.4.1	sharable interfaces, global arrays
no 32 bit stack	7.4	16 bit stack

Table 1: A summary of the differences between the Java and Java Card specifications.

An applet may communicate with the terminal only when it is selected. Only one applet is selected at any one time. An applet may call on another applet for some service. This causes a context switch, but not a deselect of the caller. There are thus two notions of an applet being 'current': "current context" and "currently selected".

An applet is automatically deselected when the terminal selects another applet. An applet is not deselected upon power failure. This arrangement makes it hard for the applet to perform a proper cleanup [21, Section 3.5].

There is a notion of a default applet, which is implicitly selected after a card reset. It is unclear how such an applet may know whether it has been selected by default, or explicitly [21, Section 4.1], or, if indeed this is relevant.

6.3 Linking and loading

Linking and loading of Java Card code has a number of aspects that Java does not have:

- Linking and loading is package based, rather than class based.
- The linking and loading process creates the additional in-card generic object attribute of ownership. This attribute is essential to the Java Card security model. However, it does not exist in Java and its relationship to the package as unit of linking and loading seems arbitrary.
- Visibility rules differ between library packages and packages that contain applets.
- The information from a class file is represented in two different files (the cap file and the export file), thus making it possible for the files to get out of synchronisation, while it is harder for a class file to become internally inconsistent.
- Version control is only supported with a major and minor version number, and not with byte code verification (on the card).
- Binary compatibility in the Java Card specification is a based on a subset of Java's binary compatibility rules, e.g. changing final static fields of primitive types is a binary compatible change according to Java, but not so according to the Java Card specification.
- Cap files contain considerable redundant information to optimize the loading, linking and applet installation, as well as the efficient lookup of methods etc, making this code representation less robust. Particular examples include the ordering of virtual methods in the appropriate table, the inclusion of the entire class hierarchy in the export file, and the separation of name spaces for public and private virtual methods [20, Section 4.3.7.6].

A further investigation into the interaction of these issues would be worth while.

6.4 Security

The Java Card API offers a protocol which applets have to go through to obtain an object shared with another applet. This has two problems: First, the protocol is quite involved, and secondly the mechanism is not fully object based. Let us consider some of the important aspects of the sharing model.

A context is a trusted domain, which acts as a principal. All applets defined in the same package share a context. A context is called a group context if it contains more than one applet. Only applets and the run-time environment create objects. The run-time environment is represented by a 'pseudo' context. The context of the applet that creates an object is the owning context of that object. An applet is an object, it is owned by its context; however, a context is not an object.

Objects created by applets in the same context may be shared freely. Objects from different contexts can only be shared if a special protocol is followed [20, 6.2.4.2]. Ownership is thus a relation between contexts and objects. In an object oriented world it would be more natural and flexible to define this relation between objects.

There are two relevant relationships: ownership and access. An applet may grant another applet access, subject to following the 'shared interface protocol'. An owner may invoke methods, read and write fields etc. An applet with only access may only invoke methods on the shared object.

A context is a static concept. There is a current context, maintained by the run-time environment. The current context is switched by calls to (instance) methods and returns from those calls, in LIFO order. Contexts are also switched by exceptions. Invoking a method and throwing an exception may cause a security exception, when the context switch is not permitted. For symmetry reasons, returning from a method should be able to also generate a security exception. Static methods and fields are transparent for context switches. The run-time environment knows which context and object belongs to from the object header.

The ownership scheme has two problems: Firstly, it does not allow for server applets to create objects on behalf of other applets, because ownership is not transferable [20, Section 6.1.3]. Secondly, the ownership scheme does not allow an applet to manage a group of other applets, because applets are owned by a context, not by an object.

The JCRE owns a number of global arrays, and a number of entry point objects. All fields, methods and components of these are accessible to all applets. References to temporary entry point objects and global arrays can only be manipulated in certain ways, subject to fairly involved rules [20, Section 6.2].

Summarizing, there are three notions of sharing: entry point objects, global arrays, and sharable interface objects. Ideally there should be just one. Unfortunately, none of the sharing mechanisms address cryptographic security. Perhaps a more logical notion of sharing would use the same mechanisms that are now used to share information between the terminal and the card. In this case, APDUs (or the high level equivalent) could be shared between either the terminal and the cards or between different applets on the card.

7 Conclusion

Figure 2 summarises our findings graphically. The innermost area consist of Java's imperative core and the object oriented features. The next layer adds exceptions, concurrency, garbage collection and finalizers. Each of these added features interacts with the object orientation and the imperative core, potentially requiring the programmer to understand, and the modeller to study many separate interactions. Java also adds a security manager and dynamic class loading requiring further interactions to be considered for regular Java. The Java Card specification does not offer the grayed features, but has a variety of features of its own. In addition the Java Card specification leaves some issues open to the implementation (e.g. whether to support `int`, which exceptions are recoverable). We believe that this gives rise to rather too many interactions to be considered easily in formal analysis and recommend simplification.

We have presented two case studies. The first discusses the programmer's view on the somewhat low level feel to Java Card programming. The second case study presents a formal model of card tear, that is shown to be consistent with the Draft SCSUG Smart Card Protection Profile. We have translated the formal model back into a simulator written in Java to show that modelling provides practical information to the Java design level.

From both case studies we conclude that Java Card specifications can be improved by simplification. In fact we give a number of concrete suggestions for such simplifications.

References

[1] P. Bertelsen. Semantics of Java byte code. Technical report, Technical Univ. of Denmark, Mar 1997. `www.dina.kvl.dk/ ~pmb/`.

[2] OpenCard Consortium. *OpenCard Framework – General Information Web Document*. IBM Deutschland Entwicklung GmbH, Böblingen, Germany, second edition, Oct 1998. `www.opencard.org`.

[3] E. Denney and Th. Jensen. Correctness of Java card method lookup via logical relations. In E. Smolka, editor, *9th European Symp. on programming (ESOP), LNCS 1782*, pages 104–118, Berlin, West Germany, Mar 2000. Springer-Verlag, Berlin.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 1996.

[5] Smart Card Security User Group. *Smart Card Protection Profile*. U. S. Dept. of Commerce, National Bureau of Standards and Technology, May 2000. `http:// csrc.nist.gov/ cc/`.

[6] P. Gutmann. Secure deletion of data from magnetic and Solid-State memory. In *6th Int. USENIX Security Symp. Focusing on Applications of Cryptography*, pages 77–89, San Jose, California, Jul 1996. Usenix Association, Berkely, California.

[7] P. H. Hartel. LETOS – a lightweight execution tool for operational semantics. *Software—practice and experience*, 29(15):1379–1416, Sep 1999. `www.ecs.soton.ac.uk/ ~phh/ letos.html`.

[8] P. H. Hartel. Formalising Java safety – an overview. In J. Domingo-Ferrer and A. Watson, editors, *4th Int. IFIP wg 8.8 Conf. Smart card research and advanced application (CARDIS)*, page to appear, Bristol, UK, Sep 2000. Kluwer Academic Publishers, Boston.

[9] P. H. Hartel, M. J. Butler, E. de Jong, and M. Longley. Transacted memory for smart cards. Declarative Systems & Software Engineering Technical Reports DSSE-TR-2000-9, Univ. of Southampton, Aug 2000. `www.dsse.ecs.soton.ac.uk/ techreports/ 2000-9.html`.

[10] J.-L. Lanet and A. Requet. Formal proof of smart card applets correctness. In J.-J. Quisquater and B. Schneier, editors, *3rd Int. Conf. Smart card research and advanced application (CARDIS 1998 preproceedings)*, Louvain la Neuve, Belgium, Sep 1998. Univ. Catholique de Louvain la Neuve.

[11] M. Montgomery and K. Krishna. Secure object sharing in Java card. In *USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 119–127, Chicago, Illinois, 1999. USENIX Assoc, Berkeley, California.

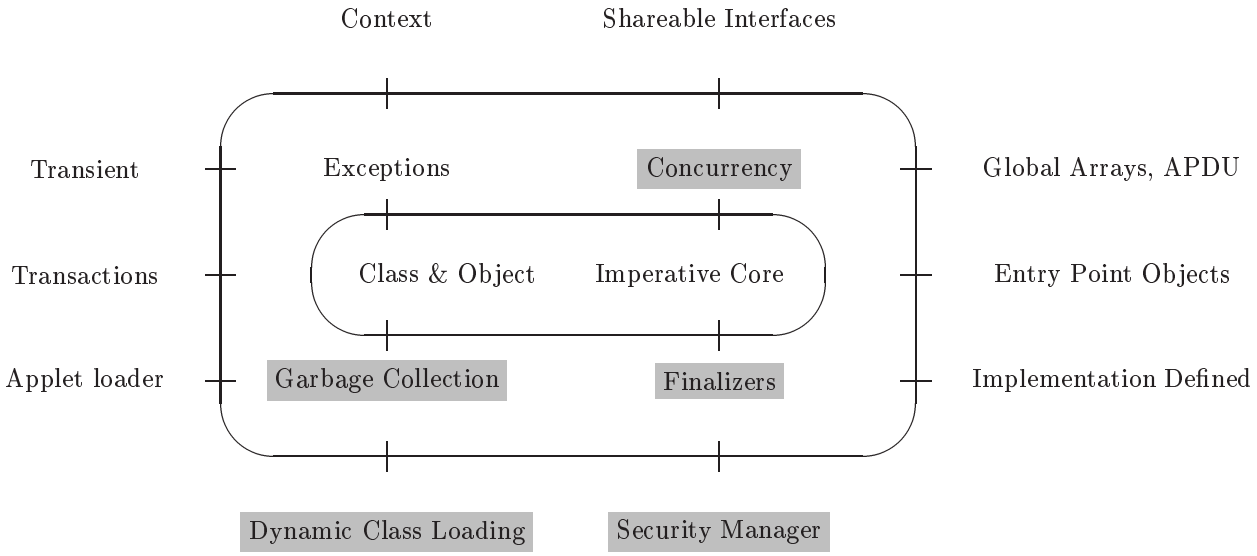

Figure 2: Interactions between features of the Java and Java Card specifications.

[12] S. Motré. Formal model and implementation of the Java card dynamic security policy. In *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'2000*, Grenoble, France, Jan 2000. `http:// www-lsr.imag.fr/ afadl`.

[13] H. R. Nielson and F. Nielson. *Semantics with applications: A formal introduction.* John Wiley & Sons, Chichester, UK, 1991.

[14] M. Oestreicher. Transactions in Java card. In *15th Annual Computer Security Applications Conference (ACSAC)*, pages 291–298, Phoenix, Arizona, Dec 1999. IEEE Comput. Soc, Los Alamitos, California. `www.acsac.org/ 1999/ abstracts/ thu-b-1500-marcus.html`.

[15] M. Oestreicher and K. Krishna. Object lifetimes in Java card. In *USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 129–37, Chicago, Illinois, 1999. USENIX Assoc, Berkeley, California.

[16] National Institute of Standards and Technology. *Common Criteria for Information Technology Security Evaluation.* U. S. Dept. of Commerce, National Bureau of Standards and Technology, Aug 1999. `http:// csrc.nist.gov/ cc/`.

[17] J. Posegga and H. Vogt. Byte code verification for Java smart cards based on model checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *European Symposium on Research in Computer Security (ESORICS), LNCS 1485*, pages 175–190, Louvain-la-Neuve, Belgium, Sep 1998. Springer-Verlag, Berlin.

[18] J. Reid and M. Looi. Making sense of smart card security certifications. In J. Domingo-Ferrer and A. Watson, editors, *4th Int. IFIP wg 8.8 Conf. Smart card research and advanced application (CARDIS)*, page to appear, Bristol, UK, Sep 2000. Kluwer Academic Publishers, Boston.

[19] E. Rose and K. H. Rose. Lightweight bytecode verification. In *OOPSLA'98 Workshop on Formal Underpinnings of Java (FUJ)*, Vancouver, Canada, Nov 1998. `www-dse.doc.ic.ac.uk/ ~sue/ oopsla/ cfp.html`.

[20] Sun. *Java Card 2.1 Applications Programming Interface.* Sun Micro systems Inc, Palo Alto, California, Jun 1999. `http:// java.sun.com/ products/ javacard/`.

[21] Sun. *Java Card 2.1 Runtime Environment (JCRE) Specification.* Sun Micro systems Inc, Palo Alto, California, Jun 1999. `http:// java.sun.com/ products/ javacard/`.

[22] Sun. *Java Card 2.1 Virtual Machine Specification.* Sun Micro systems Inc, Palo Alto, California, Mar 1999. `http:// java.sun.com/ products/ javacard/`.