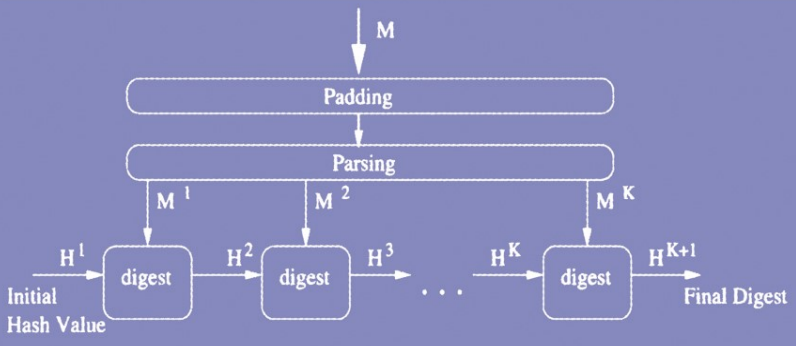
Hot Topics

LNCS 3362

Gilles Barthe
Lilian Burdy
Marieke Huisman
Jean-Louis Lanet
Traian Muntean (Eds.)

Construction and Analysis of Safe, Secure, and Interoperable Smart Devices

International Workshop, CASSIS 2004
Marseille, France, March 2004
Revised Selected Papers



Springer

Lecture Notes in Computer Science 3362

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Gilles Barthe Lilian Burdy
Marieke Huisman Jean-Louis Lanet
Traian Muntean (Eds.)

Construction and Analysis of Safe, Secure, and Interoperable Smart Devices

International Workshop, CASSIS 2004
Marseille, France, March 10-14, 2004
Revised Selected Papers

Springer

Volume Editors

Gilles Barthe
Lilian Burdy
Marieke Huisman
Jean-Louis Lanet
INRIA Sophia-Antipolis
2004 Route des Lucioles, BP 93, 06902 Sophia Antipolis, France
E-mail:{Gilles.Barthe, Marieke.Huisman, Jean-Louis.Lanet}@inria.fr
Lilian.Burdy@sophia.inria.fr

Traian Muntean
Université de la Méditerranée
Ecole Supérieure D'Ingénieurs de Luminy
Case 925 - ESIL Parc Scientifique, 13288 Marseille, France
E-mail: Traian.Muntean@esil.univ-mrs.fr

Library of Congress Control Number: 2004117384

CR Subject Classification (1998): D.2, C.3, D.1, D.3, D.4, F.3, E.3

ISSN 0302-9743
ISBN 3-540-24287-2 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11375197 06/3142 5 4 3 2 1 0

Preface

This volume contains a selection of refereed papers from participants of the workshop "Construction and Analysis of Safe, Secure and Interoperable Smart Devices" (CASSIS), held from the 10th to the 13th March 2004 in Marseille, France:

`http://www-sop.inria.fr/everest/events/cassis04/`

The workshop was organized by INRIA (Institut National de Recherche en Informatique et en Automatique), France and the University de la Méditerranée, Marseille, France. The workshop was attended by nearly 100 participants, who were invited for their contributions to relevant areas of computer science.

The aim of the workshop was to bring together experts from the smart devices industry and academic researchers, with a view to stimulate research on formal methods and security, and to encourage the smart device industry to adopt innovative solutions drawn from academic research.

The next generation of smart devices holds the promise of providing the required infrastructure for the secure provision of multiple and personalized services. In order to deliver their promise, the smart device technology must however pursue the radical evolution that was initiated with the adoption of multi-application smartcards. Typical needs include:

- The possibility for smart devices to feature extensible computational infrastructures that may be enhanced to support increasingly complex applications that may be installed post-issuance, and may require operating system functionalities that were not pre-installed. Such additional flexibility must however not compromise security.
- The possibility for smart devices to achieve a better integration with larger computer systems, through improved connectivity, genericity, as well as interoperability.
- The possibility for smart devices to protect themselves and the applications they host from hostile applications, by subjecting incoming applications to analyses that bring strong guarantees in terms of confidentiality or resource control.
- The possibility for application developers to establish through formal verification based on logical methods the correctness of their applications. In addition, application developers should be offered the means to convey to end-users or some trusted third party some verifiable evidence of the correctness of their applications.
- The possibility for smart devices to be modeled and proved correct formally, in order to achieve security evaluations such as Common Criteria at the highest levels.

In order to address the different issues raised by the evolution of smart devices, the workshop consisted of seven sessions featuring one keynote speaker and three or four invited speakers:

1. Trends in smart card research
2. Operating systems and virtual machine technologies
3. Secure platforms
4. Security
5. Application validation
6. Verification
7. Formal modeling

The keynote speakers for this edition were: Eric Vétillard (Trusted Logic), Ksheerabdhi Krishna (Axalto), Xavier Leroy (INRIA), Pieter Hartel (U. of Twente), K. Rustan M. Leino (Microsoft Research), Jan Tretmans (U. of Nijmegen), and J. Strother Moore (U. of Texas at Austin).

In addition, a panel chaired by Pierre Paradinas (CNAM), and further consisting of Jean-Claude Huot (Oberthur Card Systems), Gilles Kahn (INRIA), Ksheerabdhi Krishna (Axalto), Erik Poll (U. of Nijmegen), Jean-Jacques Quisquater (U. of Louvain), and Alain Sigaud (Gemplus), examined the opportunities and difficulties in adapting open source software for smart devices execution platforms.

We wish to thank the speakers and participants who made the workshop such a stimulating event, and the reviewers for their thorough evaluations of submissions. Furthermore, we gratefully acknowledge financial support from Conseil Général des Bouches-du-Rhône, Axalto, France Télécom R&D, Gemplus International, Microsoft Research and Oberthur Card Systems.

November 2004

Gilles Barthe
Lilian Burdy
Marieke Huisman
Jean-Louis Lanet
Traian Muntean

Organizing Committee

Gilles Barthe	INRIA Sophia Antipolis, France
Lilian Burdy	INRIA Sophia Antipolis, France
Marieke Huisman	INRIA Sophia Antipolis, France
Jean-Louis Lanet	INRIA DirDRI, France
Traian Muntean	University de la Méditerranée, Marseille, France

Reviewers

Cuihtlauac Alvarado	Rajeev Joshi	Judi Romijn
John Boyland	Florian Kammüller	Vlad Rusu
Michael Butler	Laurent Lagosanto	Peter Ryan
Koen Claessen	Yassine Lakhnech	David Sands
Alessandro Coglio	Xavier Leroy	Gerardo Schneider
Adriana Compagnoni	Gerald Lüttgen	Ulrik Pagh Schultz
Pierre Crégut	Anil Madhavapeddy	David Scott
Jean-Michel Douin	Claude Marché	Robert de Simone
Hubert Garavel	Ricardo Medel	Christian Skalka
Nikolaos Georgantas	Greg Morisett	Oscar Slotosch
Mike Gordon	Laurent Mounier	Kim Sunesen
Chris Hankin	Christophe Muller	Sabrina Tarento
Rene Rydhof Hansen	Alan Mycroft	Hendrik Tews
Klaus Havelund	Brian Nielsen	Mark Utting
Lex Heerink	David von Oheimb	Eric Vétillard
Ludovic Henrio	Arnd Poetzsch-Hefftner	Willem Visser
Charuwalee Huadmai	Erik Poll	Olivier Zendra
Thierry Jéron	Christophe Rippert	Elena Zucca

Table of Contents

Mobile Resource Guarantees for Smart Devices*

David Aspinall[1], Stephen Gilmore[1], Martin Hofmann[2],
Donald Sannella[1], and Ian Stark[1]

[1] Laboratory for Foundations of Computer Science, School of Informatics,
The University of Edinburgh
[2] Lehr- und Forschungseinheit für Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians-Universität München

Abstract. We present the Mobile Resource Guarantees framework: a system for ensuring that downloaded programs are free from run-time violations of resource bounds. Certificates are attached to code in the form of efficiently checkable proofs of resource bounds; in contrast to cryptographic certificates of code origin, these are independent of trust networks. A novel programming language with resource constraints encoded in function types is used to streamline the generation of proofs of resource usage.

1 Introduction

The ability to move code and other active content smoothly between execution sites is a key element of current and future computing platforms. However, it presents huge security challenges – aggravating existing security problems and presenting altogether new ones – which hamper the exploitation of its true potential. Mobile Java applets on the Internet are one obvious example, where developers must choose between sandboxed applets and working within a crippled programming model; or signed applets which undermine portability because of the vast range of access permissions which can be granted or denied at any of the download sites. Another example is open smart cards with multiple applications that can be loaded and updated after the card is issued, where there is currently insufficient confidence in available security measures to take full advantage of the possibilities this provides.

A promising approach to security is *proof-carrying code* [26], whereby mobile code is equipped with independently verifiable certificates describing its security properties, for example type safety or freedom from array-bound overruns. These certificates are condensed and formalised mathematical proofs which are by their very nature self-evident and unforgeable. Arbitrarily complex methods may be used by the *code producer* to construct these certificates, but their verification by the *code consumer* will always be a simple computation. One may compare this to the difference between the difficulty of producing solutions to combinatorial

* This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 1–26, 2005.

problems such as Rubik's cube or satisfiability, and the ease of verifying whether an alleged solution is correct or not.

A major advantage of this approach is that it sidesteps the difficult issue of *trust*: there is no need to trust either the code producer, or a centralized certification authority. If some code comes with a proof that it does not violate a certain security property, and the proof can be verified, then it does not matter whether the code (and/or proof) was written by a Microsoft Certified Professional or a monkey with a typewriter: the property is guaranteed to hold. The user does need to trust certain elements of the infrastructure: the code that checks the proof (although a paranoid user could in principle supply a proof checker himself); the soundness of the logical system in which the proof is expressed; and, of course, the correctness of the implementation of the virtual machine that runs the code – however these components are fixed and so can be checked once and for all. In any case, trust in the integrity of a person or organization is not a reliable basis for trusting that the code they produce contains no undiscovered accidental security bugs! In practice it seems best to take advantage of *both* existing trust infrastructures, which provide a degree of confidence that downloaded code is not malicious and provides desired functionality, *and* the strong guarantees of certain key properties provided by proof-carrying code.

Control of resources (space, time, etc.) is not always recognized as a security concern but in the context of smart cards and other small devices, where computational power and especially memory are very limited, it is a central issue. Scenarios of application which hint at the security implications include the following:

- a provider of distributed computational power may only be willing to offer this service upon receiving dependable guarantees about the required resource consumption;
- third-party software updates for mobile phones, household appliances, or car electronics should come with a guarantee not to set system parameters beyond manufacturer-specified safe limits;
- requiring certificates of specified resource consumption will also help to prevent mobile agents from performing denial of service attacks using bona fide host environments as a portal;

and the one of most relevance in the present context:

- a user of a handheld device, wearable computer, or smart card might want to know that a downloaded application will definitely run within the limited amount of memory available.

The usual way of dealing with programs that exceed resource limits is to monitor their usage and abort execution when limits are exceeded. Apart from the waste that this entails – including the resources consumed by the monitoring itself – it necessitates programming recovery action in the case of failure.

The Mobile Resource Guarantees (MRG) project is applying ideas from proof-carrying code to the problem of resource certification for mobile code. As with other work on proof-carrying code for safety properties, certificates contain

formal proofs, but in our case, they claim a resource usage property. Work in MRG has so far concentrated mainly on bounds on heap space usage, but most of the infrastructure that has been built is reusable for bounds on other kinds of resources. One difference between MRG and other work on proof-carrying code is that proof certificates in MRG refer to bytecode programs rather than native code. One bytecode language of particular interest is JVML [22] but there are others, including the CIL bytecode of the Microsoft .NET framework [24], JavaCard [33], and the restricted version of JVML described in [32]. An elegant solution to the tension between the engineering requirement to make theorem proving and proof checking tractable, while at the same time remaining faithful to the imperative semantics of these underlying bytecode languages, is the Grail intermediate language (see Sect. 5) which also targets multiple bytecode languages.

One of the central issues in work on proof-carrying code is how proofs of properties of code are produced. One traditional approach is for object code and proofs to be generated from source code in a high-level language by a *certifying compiler* like Touchstone [10], using types and other high-level source information[1]. The MRG project follows this approach, building on innovative work on linear resource-aware type systems [14, 15], whereby programs are certified by virtue of their typing as satisfying certain resource bounds. For instance, in a space-aware type system, the type of an in-place sorting function would be different from the type of a sorting function, like merge sort, that requires extra working space to hold a copy of its input; still different would be the type of a sorting function that requires a specific number of extra cells to do its work, independent of the size of its input. A corresponding proof of this behaviour at the bytecode level can be generated automatically from a typing derivation in such a system in the course of compiling the program to bytecode. It even turns out to be possible to *infer* heap space requirements in many situations [16]. This work has been carried out in a first-order ML-like functional language, Camelot (described in Sect. 3), that has been developed as a testbed by the MRG project. The underlying proof-carrying code infrastructure operates at the bytecode (Grail) level; Camelot is just an example of a language that a code producer might use to produce bytecode together with a proof that it satisfies some desired resource bound.

This paper is an overview of the achievements of the MRG project as of the summer of 2004. It is self-contained, but due to space limitations many points are sketched or glossed over; full technical details can be found in the papers that are cited below. The main contribution of the paper is a presentation of the overall picture into which these technical contributions are meant to fit.

In the next section, we describe the overall architecture of the MRG framework, including the rôle of the two language levels (Grail and Camelot), and how MRG-style proof-carrying code fits with standard Java security. Sections 3 and 4

[1] A slightly different approach was taken by the work on Typed Assembly Language ([25] and later), where a fixed type system is provided for the low-level language, and certification amounts to providing a typing in this low-level type system.

focus on the "upper" language level, introducing Camelot and space-aware type systems. Section 5 focuses on the "lower" language level, describing the Grail intermediate language and the way that it provides both a tractable basis for proof and relates to (multiple) imperative bytecode languages. Section 6 ties the two language levels together by explaining the logic for expressing proofs of resource properties of bytecode programs and the generation of proofs from resource typings. A conclusion outlines the current status of the MRG project and summarizes its contributions.

2 Architecture and Deployment

In this section we discuss the architecture of a smart device-based system which deploys the technology of the MRG project in a novel protocol for certifying resource bounds on downloaded code from an untrusted source. Our protocol is designed so that it can be integrated with the built-in mechanism for Java bytecode checking, via the *Security Manager*. In the JVM, the Security Manager is entrusted with enforcing the security policy designated by the user, and ensuring that no violations of the security policy occur while the code runs.

In our protocol, a *Resource Manager* is responsible for verifying that the certificate supplied with a piece of code ensures that it will execute within the advertised resource constraints. A *Proof Checker* is invoked to do this. If the check succeeds, we have an absolute guarantee that the resource bounds are met, so it is not necessary to check for resource violations as the code runs. Our Resource Manager is not a replacement for the standard Java Security Manager but instead forms a *perimeter defense* which prevents certain non-well-behaved programs from being executed at all.

The Mobile Resource Guarantees framework provides a high-level language, Camelot, and a low-level language, Grail, into which this is compiled. (Camelot is presented in more detail in Sect. 3 and Grail is discussed in Sect. 5.) Application developers work in the high-level language and interact with resource typing judgements at the appropriate level of abstraction for their realm of expertise. For this approach to be successful it is necessary for the compilation process to be *transparent* [23] in that the resource predictions made at the high-level language level must survive the compilation process so that they remain true at the low level. This places constraints on the expressive power of the high-level language, prohibiting the inclusion of some more complex language features. It also places constraints on the nature of the compilation process itself, requiring the compiler to sometimes sacrifice peak efficiency for predictability, which is the familiar trade-off from development of real-time software.

A consumer of proof-carrying code (such as Grail class files with attached proofs of resource consumption) requires an implementation technology which enforces the security policy that they specify. The *Java agents* introduced in the J2SDK version 1.5.0 provide the most direct way to implement these policies. An agent is a "hook" in the JVM allowing the PCC consumer to attach their own implementation of their security policy as an instance of a general-purpose PCC Security Manager.

Java agents can be used for several resource-bound-specific purposes:

1. to query the attached proof and decide to refuse to load, build and execute the class if necessary;
2. to apply *per-class* or *per-package* use restrictions by modifying each method in the class with entry and exit assertions that inspect resource consumption measures; and
3. to apply *per-method* constraints on heap-allocation and run-time by instrumenting method bodies.

Each of these checks can be unloaded at JVM instantiation time to allow a mobile-code consumer to vary their security policy between its tightest and laxest extrema.

3 Space Types and Camelot

This section describes the high-level language Camelot and the space type system which together allow us to produce JVM bytecode endowed with guaranteed and certified bounds on heap space consumption.

Syntactically, and as far as its functional semantics is concerned, Camelot is essentially a fragment of the ML dialect O'Caml [29]. In particular, it provides the usual recursive datatypes and recursive (not necessarily primitive recursive) definition of functions using pattern matching, albeit restricted to flat patterns.

One difference to O'Caml is that Camelot compiles to JVM bytecode and provides (via the O'Camelot extension [36]) a smooth integration of genuine Java methods and objects.

The most important difference, however, lies in Camelot's memory model. This uses a freelist, managed directly by the compiled code, rather than relying exclusively on garbage collection. All non-primitive types in a Camelot program are compiled to JVM objects of a single class Diamond, which contains appropriate fields to hold data for a single node of any datatype. Unused objects are released to the freelist so that their space can be immediately reused. The compiler generates the necessary code to manage the freelist, based on some language annotations described below.

This conflation of types into a single allocation unit is standard for memory recycling in constrained environments; there is some loss of space around the edges, but management is simple and in our case formally guaranteed to succeed. If required, we could duplicate our analysis to manage a range of cell sizes in parallel, but we have not yet seen compelling examples for this.

3.1 The Diamond Type

Following [14], Camelot has an abstract type denoted <> whose members are heap addresses of Diamond-objects. The only way to access this type is via datatype constructors. Suppose for example that we have defined a type of integer lists as follows[2]

[2] The annotation ! ensures that the constructor Nil is represented by a null pointer rather than a proper object.

```
type iList = !Nil | Cons of int * iList
```

If this is the only type occurring in a program then the Diamond class will look as follows (in simplified form and Java notation):

```
public class Diamond extends java.lang.Object {
    public Diamond R0;
    public int V1;
}
```

If, say, x1 is an element of type iList, hence compiled to an object reference of type Diamond, we can form a new list x2 by

```
let x2 = Cons(9,x1) in ...
```

The required object reference will be taken from the aforementioned freelist providing it is non-empty. Otherwise, the JVM new instruction will be executed to allocate a new object of type Diamond.

If, however, we have in our local context an element d of type <> then we can alternatively form x2 by

```
let x2 = Cons(9,x1)@d in ...
```

thus instructing the compiler to put the new Cons cell into the Diamond object referenced by d, whose contents will be overwritten.

Using these phrases in the context of pattern matching provides us with elements of type <> and also refills the freelist. A pattern match like

```
match x with
  Cons(h,t)@d -> ...
```

is evaluated by binding h, t and d to the contents of the "head" (h) and "tail" (t) fields and the reference to x itself (d). Thus, in the body of the pattern match d is an element of type <> available for constructing new Cons cells.

Alternatively, the syntax

```
match x with
  Cons(h,t)@_ -> ...
```

returns the cell occupied by x to the freelist for later use.

Finally, an unannotated pattern match such as

```
match x with
  Cons(h,t) -> ...
```

performs ordinary non-destructive matching.

3.2 Linear Typing

When a list x is matched against a pattern of the form Cons(h,t)@d or Cons(h,t)@_ it is the responsibility of the programmer to ensure that the list x itself is not

used anymore because its contents will be overwritten subsequently. For this purpose, the Camelot compiler has an option that enforces (affine) linear use of all variables. If all variables are used at most once in their scope then there can in particular be no reference to x in the body of the pattern match above. In [14] a formal proof is given that such a program behaves purely functionally, i.e., as if the type <> was replaced by the unit type. Linear typing is, however, a fairly crude discipline and rules out many sound programs. In [6] we present an improved type system that distinguishes between modifying and read-only access to a data structure and in particular allows multiple read-only accesses, which would be ruled out by the linear discipline. This is not yet implemented in Camelot. Alternatively, the programmer can turn off the linear typing option and rely on his or her own judgement, or use some other scheme.

3.3 Extended Example

The code in Figure 1 shows a standalone Camelot application containing a function start : string list −> unit which serves as an entry point. It is assumed that the program is executed by applying start to an (ordinary) list of strings obtained, e.g., from the standard input.

We see that the function ins destroys its argument, whereas the sorting function sort : ilist −> ilist, as well as the display function show_list : ilist −> unit, each leave their argument intact.

3.4 Certification of Memory Usage

The idea behind certification of heap-space usage in MRG is as follows: given a Camelot program containing a function start : string list −> unit, find a linear function $s(x) = ax + b$ with the property that evaluating (the compiled version of) start on an input list of length n will not invoke the new instruction *provided* that the freelist contains initially no less than $s(n)$ cells.

Once we have such a linear function s we can then package our compiled bytecode together with a wrapper that takes input from stdin or a file, initialises (using new) the freelist to hold $s(n)$ cells where n is the size of the input, and then evaluates start.

3.5 Inference of Space Bounds

Such linear space bounds can efficiently be obtained using the type-based analysis described in [16] which has subsequently been implemented and tuned to Camelot in [17]. In summary, this analysis infers for each function contained in the program a numerically annotated type describing its space usage. The desired bounding function can then be directly read off from the type of start.

The result of running the analysis on our example program is given in Figure 2. The entry

```
ins : 1, int −> iList[0|int,#,0] −> iList[0|int,#,0], 0;
      ↑
```

```
type iList = !Nil | Cons of int * iList

let ins a l = match l with
                Nil -> Cons(a,Nil)
              | Cons(x,t)@_ -> if a < x then Cons(a,Cons(x,t))
                               else Cons(x, ins a t)

let sort l = match l with
               Nil -> Nil
             | Cons(a,t) -> ins a (sort t)

let show_list0 l = match l with
                     Nil -> ""
                   | Cons(h,t) -> begin
                       match t with
                         Nil -> string_of_int h
                       | Cons(h0,t0) -> (string_of_int h) ^ ", " ^ (show_list0 t)
                     end

let show_list l = "[" ^ (show_list0 l) ^ "]"

let stringList_to_intList ss =
      match ss with
        [] -> Nil
      | (h::t) -> Cons((int_of_string h),(stringList_to_intList t))

let start args =
      let l1 = (stringList_to_intList args)
    in let _  = print_string ("\nInput list:\n l1 = " ^ (show_list l1))
    in let l2 = sort l1
    in let _  = print_string ("\nResult list:\n l2 = " ^ (show_list l2))
    in ()
```

Fig. 1. A standalone Camelot program

```
ins           : 1, int -> iList[0|int,#,0] -> iList[0|int,#,0], 0;
int_of_string : 0, string -> int, 0;
print_string  : 0, string -> unit, 0;
show_list     : 0, iList [0|int,#,0] -> string, 0;
show_list0    : 0, iList [0|int,#,0] -> string, 0;
sort          : 0, iList [0|int,#,1] -> iList[0|int,#,0], 0;
start         : 0, list_1 [string,#,2|0] -> unit, 0;
stringList_to_intList : 0, list_1 [string,#,2|0] -> iList[0|int,#,1], 0;
string_of_int : 0, int -> string, 0;
```

Fig. 2. Output of space analysis on the program in Fig. 1

indicates that a successful run of ins requires the freelist to contain 1 cell to begin. The entry

```
stringList_to_intList : 0, list_1 [string,#,2|0] -> iList[0|int,#,1], 0;
                                                    ↑                  ↑
```

indicates that a call to stringList_to_intList on an input list of length n requires a freelist of size $2n$ and upon completion leaves a freelist of size $1m$ where m is the length of the *resulting* iList.

Finally, the entry

```
start : 0,  list_1 [string,#,2|0] -> unit, 0
                          ↑
```

indicates that a call to start requires a freelist of size $2n$ where n is the length of the input, so the desired bounding function can be chosen as $s(n) = 2n$ in this case.

More generally, a (hypothetical) entry

```
f : 3,  iList [0|int,#,17] -> iList[0|int,#,13],11
```

would indicate that a call to f : iList −> iList with an argument of length n requires a freelist of minimum size $3 + 17n$ to succeed without invoking new. Moreover, if the resulting list has length m then the freelist will have size at least $11 + 13m$ plus of course the number of cells left over from the initial freelist in case its size was above the minimum specified by the typing.

Actually, the meaning of the constant 17 in the typing is "17 per Cons-cell of the argument" which in the case of linear lists specialises to $17n$ with n being the length. In the case of data structures with more than one constructor, nested data-structures, and tree-like data structures this view is more fine-grained than linear functions in the overall size.

Other examples discussed in [17] include functions on trees such as heap sort and computation of Huffman codes, as well as functions where the space bounding function has fractional coefficients, e.g. $s(n) = \frac{4}{3}n$.

Regarding the functionality of the space inference we note two important aspects. First, the numerical annotations arise as solutions of a system of linear inequalities which is in turn obtained from a skeleton type derivation which has a numerical variable wherever a numerical annotation would be required. Second, the soundness of destructive pattern matches in the sense of Sect. 3.2 arises also as a precondition to the correctness of the space analysis. For further detail we refer to [16].

4 Parameter Size

Other guarantees of resource properties that are under consideration in MRG include execution time, stack size, and size of parameters supplied to system calls. Of these three, the third has been studied in more detail albeit not to the same extent as heap space consumption. We summarise the partial results achieved so far.

Suppose we are given a system call

```
brake : int * int -> unit
```

where it is "safety critical" that whenever brake is called with parameters (x,y) then some proposition P(x,y), e.g. a conjunction of linear inequalities describing

some "safe window", must be satisfied. We emphasize that the brake function itself will not be implemented in Camelot but assumed as given, for example by being part of the system architecture.

A possible scenario for such a situation could consist of a car manufacturer providing an API which allows for third party firmware updates. The manufacturer would publicise a certain safety policy concerning the parameters supplied in calls to the methods provided in the API. The manufacturer would guarantee that adherence to this safety policy will prevent severe hazards. Within the safety policy the third party provider will try to optimise the behavioural properties of the system. Whether or not such optimisation actually happens need not be formally established; our goal is only to ensure adherence to the manufacturer-supplied safety policy.

In order to express such a safety property on the bytecode level one can use the instrumented operational semantics and bytecode logic which will be described below in Sect. 6.

Here we are concerned with the question of how such safety policies can be expressed on the level of Camelot through a type system in such a way that functions can be checked individually once their typing is known, following the usual typing principles, e.g.: to show that a function has a certain type show that its body has that type assuming it to hold for any recursive call within the body.

We claim that a solution based on *dependent types* in the style of Dependent ML [37] fits these requirements very closely. In a nutshell, the idea is as follows. In order to guarantee that a function main calls brake exclusively with such safe parameters, we may try to type main using dependent types under the following assumed typing for brake. (Since the function brake is not itself implemented in Camelot, we can assume an arbitrary typing for it.)

$$\mathsf{brake} : \{(\mathsf{x},\mathsf{y}) : \mathsf{int} * \mathsf{int} \mid \mathsf{P}\} \mathrel{-\!\!>} \mathsf{unit}$$

We will now explain how this idea has been elaborated within MRG. We re-emphasise that these results are partial as yet and that this section must be understood as a report on on-going work.

4.1 Extending Camelot with Dependent Types

In order to express the desired typing constraints without overly interfering with the language design, Pfenning and Xi's Dependent ML (DML) [37] appears to be particularly suitable.

DML assumes a simply (i.e. non-dependently) typed base language on top of which dependent types are added in such a way that every dependently typed program also makes sense in the simply typed base language. Moreover, the question whether or not a given simply typed program admits a given dependent typing translates into a constraint solving problem in a certain constraint language over which DML is parametric. For our purposes, we choose linear arithmetic over the integers as constraint language.

DML types may depend on types from the constraint language, but not on other DML types. In particular, the types from the constraint language, the so-called *index sorts*, do not themselves form DML types but can be reflected into DML using singleton types if so desired. Likewise, code may not depend on values of index sorts, only on values of DML types. In this way, all index information may be erased from a DML program and a simply typed program is obtained.

We remark that this is not the case for more radical approaches to dependent typing such as Cayenne [7].

For our purposes, we use linear arithmetic for the constraint language which means that the index sorts include int and are closed under cartesian product. We need two type families: Int, Bool : int −> Type with the intention that Int(i) contains just the integer i and Bool(i) contains true if $1 \leq i$ and false otherwise.

We assume the following constants with typing as indicated:

```
0 : Int(0)
1 : Int(1)
plus : Pi x,y:int.Pi xx:Int(x).Pi yy:Int(y).Int(x+y)
true : Pi x:int|1<=x.Bool(x)
false : Pi x:int|x<=0.Bool(x)
leq : Pi x,y:int.Int(x) -> Int(y) -> Bool(1+y−x)
```

The type former Pi obeys the usual rule for dependent function space due to Martin-Löf: if e : Pi x:t.A(x) and i:t then e[i] : A(i). We use square brackets for dependent application to mark that the argument of such an application is always a term of the constraint language which can be automatically inferred using Xi's elaboration algorithm [37]. Moreover, index application is irrelevant to the actual behaviour of a program; it only affects typeability.

Subset constraints in a Pi-type express that such index arguments must obey a certain constraint and are written using vertical bars as in the typing of true. They can be viewed as syntactic sugar for ordinary dependent application if one closes the index sorts under subsets of index sorts with respect to predicates expressible in the constraint language.

In order to be able to reflect knowledge about branches in a case distinction into the typing we use the following typing rule for if-then-else:

$$\frac{\Gamma \vdash t_1 : \mathsf{Bool}(i) \qquad \Gamma, 1 \leq i \vdash t_2 : T \qquad \Gamma, i \leq 0 \vdash t_3 : T}{\Gamma \vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T} \qquad (\textsc{If})$$

We remark here that typing contexts contain bindings of ordinary variables to types, of index variables to index sorts, and constraints on the index variables.

The remaining types and rules of Camelot remain unchanged. We emphasize that since DML has been developed as an extension to Standard ML and Camelot is equivalent to a subset of Standard ML this extension is unproblematic and not at all innovative. What is innovative is the application of DML to the problem of certifying parameter sizes and also, perhaps, the particular DML signature given above.

4.2 Example

We are now in a position to formally express the constraints on calls to brake as a DML typing.

```
brake : Pi x,y:int|P(x,y).Int(x) -> Int(y) -> Unit
```

For the sake of concreteness assume that P(x,y) is x+y <= 10.

Now suppose that we are given a main function that calls brake with two given parameters but prior to the call checks that these parameters indeed satisfy the required constraint, in order to prevent unsafe calls to brake:

```
brake : Int -> Int -> Unit
main : Int -> Int -> Unit
main = lambda xx:Int.lambda yy:Int.
          if leq(plus(xx,yy),10)
             then brake(xx,yy)
             else brake(0,0)
```

Here is how this program can be typed in our dependently typed extension. As already mentioned the index abstractions and applications can be inferred automatically.

```
brake : Pi x,y:int|x+y<=10.Int(x) -> Int(y) -> Unit
main : Pi x,y:int.Int(x) -> Int(y) -> Unit
main = lambda x,y:int.lambda xx:Int(x).lambda yy:Int(y).
          if leq[x+y,10](plus[x,y](xx,yy),10)
             then brake[x,y](xx,yy)
             else brake[0,0](0,0)
```

Let us see how the typing rule If above allows us to typecheck this definition. Put

$$\Gamma = \mathsf{x} : \mathsf{int}, \mathsf{y} : \mathsf{int}, \mathsf{xx} : \mathsf{Int}(\mathsf{x}), \mathsf{yy} : \mathsf{Int}(\mathsf{y})$$

and then we have

$$\Gamma \vdash \mathsf{leq}[\mathsf{x}+\mathsf{y}, 10](\mathsf{plus}[\mathsf{x},\mathsf{y}](\mathsf{xx},\mathsf{yy}), 10) : \mathsf{Bool}(1 + 10 - (\mathsf{x}+\mathsf{y}))$$

So, in the "then" branch of the conditional we have the additional constraint

```
1 <= 1+10-(x+y)
```

or equivalently, x+y <= 10, which is what is required to typecheck the application brake[x,y](xx,yy).

4.3 Summary

We have shown in this section how an extension of Camelot with dependent types in the style of DML [37] can be used to automatically check adherence to bounds on parameters to system calls. Of course, this is only a first step. In

contrast to the case of heap space certification, we have not yet developed the automatic generation of certificates in bytecode logic for this application. More significantly, generalisations of the described per-call policy will be needed for some applications. For instance, one may want to require some constraint on the parameters supplied to all calls of a given function during a certain interval. As a special case, one might require an upper bound on the number of calls to some function (e.g., network connections) or on the sum of the parameters, e.g., in the case where a system function performs an automatic payment. It remains to be seen to what extent dependent typing provides solutions to these problems as well.

5 Grail

We now move on to the low-level *Grail* language that is the target of the Camelot compiler, and our vehicle for proof-carrying code[3]. The challenge here is to identify a language that not only supports formal proofs of resource usage, based on information from Sect. 3, but also maps directly onto executable bytecodes. To do this we give Grail two distinct semantics, one functional and one imperative. These are provably compatible, and the two viewpoints allow flexible reasoning about resources.

An extended discussion of the properties of Grail appears in [9]. Here we begin by outlining the constraints that shape it. The language for our proof-carrying code needs to be all of the following:

- The target for the Camelot compiler;
- A basis for attaching resource assertions;
- Amenable to formal proof about resource usage;
- The format for sending and receiving guaranteed code;
- Executable.

The first three of these suggest a simple functional language, suitable as the output of a transforming Camelot compiler. This is strengthened by the fact that we must also perform transparent compilation, to preserve resource information computed at the Camelot level. However, the final two requirements demand a ruder machine language: what we guarantee should be the actual resource profile of runnable code.

Our solution is to arrange that Grail programs, such as that in Fig. 3, can be both evaluated functionally, using call-by-value, and executed imperatively, with state and `goto` – with both routes giving exactly the same result. In the functional reading "`x=5`" is a lexically-scoped declaration; on imperative execution it updates a named storage cell. This dual approach then satisfies all the requirements for our low-level language. The final two requirements are satisfied by providing an assembler from (the imperative interpretation of) Grail to JVM classfiles, which can be executed and transported over a network, and a disassembler that reconstructs the original Grail code.

[3] Grail stands for "Guaranteed Resource Allocation Intermediate Language".

```
method static int fib (int n) =
    let
        val a = 0                              // Local variable declarations
        val b = 1

        fun loop (int a, int b, int n) =       // Local function declaration
            let
                val b = add a b                // Lexically scoped variables
                val a = sub b a                // hide outer declarations
                val n = sub n 1
            in
                test(n,a,b)                    // Tail recursive function call
            end

        fun test (int n, int a, int b) =       // Another function declaration
            if n<=1 then b else loop(a,b,n)    // Conditional recursive call
    in
        test(n,a,b)                            // Main expression
    end
```

Fig. 3. Grail code to compute the Fibonacci number F_n. For speed, we keep track of both F_k and F_{k+1} in accumulating parameters a and b

The functional semantics is comparatively standard: Grail has strong static typing, call-by-value first-order functions, mutually recursive local declarations, and lexical scoping. Within this, we make several simplifications appropriate to a compiler target language. For example, local function declarations may not nest, functions are only applied to values, and expressions can contain just one basic operation; later we shall see some further constraints on control and dataflow.

The Fibonacci code of Fig. 3 is the body of a single method. Above this, Grail provides precisely the class and object structure built into the Java virtual machine. Thus the basic expression operators include not just add and sub but also primitives to create and manipulate objects on the Java heap. We use these to implement the space management inferred for Camelot programs by the analyses of Sect. 3.

The comments in Figure 4 present an alternative view of the same code, as a purely imperative stream of assignment statements and jumps. Instead of local functions we have a collection of basic blocks, function calls are merely jumps, and parameter lists now track which variables are live. This imperative reading gives a direct map onto Java bytecode: Grail variables are JVM variables, and each statement expands to a short sequence of instructions, which compose exactly as laid out in the Grail source. For example:

```
val b = add a b
val a = sub b a      becomes
val n = sub n 1
```

```
 9 iload_1     13 iload_2     17 iload_0
10 iload_2     14 iload_1     18 iconst_1
11 iadd        15 isub        19 isub
12 istore_2    16 istore_1    20 istore_0
```

```
method static int fib (int n) =
    let
        val a = 0                          // Initial assignment
        val b = 1                          //   to variables

        fun loop (int a, int b, int n) =   // Labelled basic block, with
            let                            //   live variable annotation
                val b = add a b
                val a = sub b a            // Sequence of assignments
                val n = sub n 1            //   updating named registers
            in
                test(n,a,b)                // Goto, with live variable
            end                            //   annotation

        fun test (int n, int a, int b) =   // Another labelled basic block
            if n<=1 then b else loop(a,b,n)  // Conditional return or jump
    in
        test(n,a,b)                        // Initial entry label
    end
```

Fig. 4. Imperative Grail code to calculate the Fibonacci number F_n. Comments indicate semantics for execution on the Java virtual machine

This gives bytecode that is highly stereotyped, and our disassembler recovers the original Grail simply by clustering instruction sequences. We can even identify variable names from standard JVM metadata.

These different views on Grail allow us to support sound formal reasoning, using the logical rules presented in the next section, at the same time as effective transmission and execution, following the architecture of Sect. 2. However, this is only justified if the functional and imperative semantics coincide. We ensure this by placing some additional constraints on Grail. A method declaration is *well-formed* if:

- Local functions are closed (all variables appear in their parameter lists);
- Invocations of local functions are all tail calls;
- The arguments of every function call syntactically match its declared parameters – for example, **fun** f (int x) is always invoked as f(x).

We have a formal semantics for both the functional and imperative views of Grail, defined by induction over the structure of programs. For well-formed code we can prove a strong correspondence between these:

Theorem 1. *Every well-formed Grail method body can be presented either as functional declarations or decomposed into imperative basic blocks:*

$$\mathsf{mbody} \underset{\text{functional}}{\overset{\text{imperative}}{\rightleftharpoons}} \mathsf{blocklist}.$$

Suppose now that E is a variable environment and s is a matching initial state, appropriate for mbody *and* blocklist *respectively:*

$$E =_{var} s \qquad \textit{where} \qquad var = fv(\mathsf{mbody}) = Var(\mathsf{blocklist})\ .$$

Then functional evaluation and imperative execution coincide: for any final value v

$$E \vdash \mathsf{mbody} \Downarrow_{fun} v \qquad \textit{if and only if} \qquad s \vdash \mathsf{blocklist} \Downarrow_{imp} v\ .$$

Moreover, these evaluations also have identical effect on the heap, and make matching use of time and memory space.

Sect. 6 has more detail on the functional operational semantics, and its accompanying logic. We can apply this theorem to show that evaluation metrics for functional Grail match execution steps of the corresponding imperative Java bytecode [30]. Part of this development includes a formalisation within the Isabelle theorem prover of both functional and imperative semantics, as well as the translation between them.

Further results on the properties of well-formed Grail appear in [9]: relating (functional) free variables to (imperative) liveness; and matching dataflow analysis of imperative single-use registers to a functional linear type system.

To take advantage of these results, our Camelot compiler must of course generate well-formed Grail. To do this it carries out a range of standard transformations, such as λ-lifting, variable renaming, and insertion of intermediate declarations. These are all legitimate functional rearrangements; but in the light of Theorem 1 we can also show that these correspond directly to imperative compilation techniques: namely conversion to static single-assignment form (SSA) and then elimination of Φ-functions [3].

This is an instance of a more general observation, that low-level transformations on registers and imperative variables map to functional transformations of Grail. Thus we can carry out bytecode optimisations like register allocation and sharing while still in the intermediate language of our compiler [35].

Many of the transformations used in compiling Camelot to Grail are familiar from other functional languages; ideas like A-normal and CPS form, types in compilation, and typed low-level languages [2, 5, 11, 12, 25, 34]. We have taken particular inspiration from λ-JVM, a functional language for expressing general JVM programs [19]. The novelty of Grail, by comparison with these other schemes, lies in the fact that it is strict enough to support a reversible translation to bytecode which preserves execution costs.

Grail generates bytecode with a regular form that makes it particularly straightforward to analyze: for example, the JVM operand stack is always empty between Grail statements, and local variables keep the same type throughout a method body. Simplifications like these appear in other proposals for effective use of Java on smart devices – thus, for example, all Grail programs immediately satisfy Leroy's conditions for fast on-card JavaCard verification [21]. Similarly, the *Squawk* JVM architecture runs on very small devices with a tripartite memory

structure (ROM/NVRAM/RAM) [32]: we already satisfy many of the conditions for Squawk bytecode, and we believe that the remaining ones can be ensured by manipulation at the Grail level.

In building a PCC framework with Java classfiles as the transport format, the natural question is: why not just use Java bytecode as the base language? The results presented here give the answer: Grail *is* Java bytecode, but with a stern discipline over the flow of control and data that makes it efficient and straightforward to analyze.

6 Bytecode Logic and Certificate Generation

Our proof-carrying code infrastructure equips Grail programs with *certificates* concerning their resource usage. Certificates contain a claim of resource usage together with (instructions for generating) a proof of the claim. The proof is expressed in a program logic for Grail that we have designed specifically for the purpose. In this section we give an overview of the program logic and then of the process of certificate generation. Full technical details of this work appear in [4, 8].

6.1 Resource-Counting Operational Semantics

We want assertions to express properties of program execution as defined by the Grail (functional) operational semantics. The operational semantics is defined as a big-step relation which is annotated with resource measurements. An expression e is evaluated in an environment E and heap h, written

$$E \vdash h, e \Downarrow (h', v, p).$$

to yield a value v, an updated heap h', and a *resource component* p. As usual, an environment is a mapping from variables to values, and a heap is modelled as a finite map from a set of locations to values.

The resource component p is a tuple which includes a measure of the number of instructions executed when evaluating e, and the maximum size of the frame stack. The amount of heap space consumed when evaluating e is not included in p, because it can be calculated as the size of the difference between the domains of the input heap and output heap, $|dom(h') - dom(h)|$. This is possible because we do not model garbage collection; indeed, the JVM specification [22] does not even require garbage collection to occur (and it does not take place on the versions 1.X of the JavaCard platform).

The rules defining the operational semantics are straightforward to write, given knowledge of the translation from Grail into Java bytecode outlined in the previous section. Example rules for `if` statements are shown further below.

6.2 A Logic for Grail

A possible starting point for the logic would be to take an existing program logic for Java bytecode, perhaps based on cutting down a logic for Java, and extend it

to express the resource-related properties of interest. However, to do this would be to ignore the advantages brought by Grail: rather than attaching assertions to sequences of bytecode instructions, we may attach them to Grail functions, and relate them rather directly to the types used by our Camelot compiler. Moreover, the functional viewpoint afforded by Grail allows more elegant rules in several cases than are possible in a Hoare-style logic.

This led us to design a custom logic of partial correctness for Grail. Sequents are of the form:

$$\Gamma \rhd e : P,$$

relating a Grail expression e to a specification P under some set of assumptions Γ of the same form. The specification P denotes a predicate which constrains possible executions of e as defined by the resource-counting operational semantics.

Satisfaction of a specification P by a program e is denoted by $\models e : P$ and asserts that every (terminating) execution lies within the domain of P, that is

$$\forall E,\ h,\ h',\ v,\ p. \quad E \vdash h, e \Downarrow (h', v, p) \qquad \text{implies} \qquad P(E,\ h,\ h',\ v,\ p).$$

Similarly to VDM, our specification predicates allow us to relate the environment and the initial heap to the result, the final heap and the resources consumed. This means that there is no need for auxiliary variables that are necessary in a Hoare-style logic to relate results in the post-condition to inputs in the pre-condition. This has a particular technical advantage in that we do not require the often rather complicated *adaptation rules* of Hoare logic when using proven (or assumed) specifications for procedures[4].

So far we have not introduced a syntax for writing specification predicates. Instead we use the higher-order logic of the theorem prover, Isabelle/HOL [28], in which we have formalised the entire Grail-based PCC framework. This particular form of shallow embedding for propositions is known as the *extensional* approach. As a rather trivial example of a specification, the predicate $|dom(h)| = |dom(h')|$ is satisfied by programs which do not allocate heap space.

Many of the rules of our logic correspond closely with rules of the operational semantics. For example, the rule for an if statement looks like this:

$$\frac{\Gamma \rhd e_1 : P_1 \qquad \Gamma \rhd e_2 : P_2}{\begin{array}{l}\Gamma \rhd \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \lambda E\, h\, h'\, v\, p.\, \exists p'.\, p = tick_2(p') \wedge \\ \qquad (E\langle x\rangle = \mathsf{true} \longrightarrow P_1(E,\ h,\ h',\ v,\ p')\ ) \wedge \\ \qquad (E\langle x\rangle = \mathsf{false} \longrightarrow P_2(E,\ h,\ h',\ v,\ p')\ ) \wedge \\ \qquad (E\langle x\rangle = \mathsf{true} \vee E\langle x\rangle = \mathsf{false})\end{array}} \quad (\textsc{if})$$

In fact, every if statement satisfies a predicate which is equivalent to this form: either the environment binds x to true and we have P_1, or the environment binds x to false and we have P_2. But P_1 and P_2 are not satisfied exactly: we have to

[4] In our case: Grail methods and function calls.

adjust the resource component p to account for two extra bytecode instructions (corresponding to the variable lookup and branch).

This rule in the logic captures the behaviour of evaluating either branch of an if statement, expressed in the operational semantics by the two cases:

$$\frac{E\langle x\rangle = \mathsf{true} \quad E \vdash h, e_1 \Downarrow (h_1, v, p)}{E \vdash h, \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow (h_1, v, \mathit{tick}_2(p))} \quad (\textsc{if-true})$$

$$\frac{E\langle x\rangle = \mathsf{false} \quad E \vdash h, e_2 \Downarrow (h_1, v, p)}{E \vdash h, \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow (h_1, v, \mathit{tick}_2(p))} \quad (\textsc{if-false})$$

which also advance the clock resource component by two steps.

The only place in which the domain of the heap is altered is in Grail's new statement, which corresponds to a `new` statement in Java. This uses a special constructor for a class c which assigns the contents of x_i to each of the n fields z_i in the newly constructed object. The rule in the logic is this:

$$\frac{}{\Gamma \rhd \mathsf{new}\ c\ [\overline{z_i := x_i}] : \lambda\, E\, h\, h'\, v\, p.\ p = \mathit{tick}_{n+1}() \wedge v = \mathsf{Ref}\ \mathit{newloc}(h) \wedge h' = h[\mathit{newloc}(h) \mapsto (c, \{\overline{z_i := E\langle x_i\rangle}\})]} \quad (\textsc{vnew})$$

The function *newloc* models the JVM memory allocator's assignment of a new location which isn't already in the domain of h. An object is modelled as a pair (c, flds) where c is a class name and *flds* is a record assigning field names to values. As usual, the resource component counts the clock ticks: in this case the time taken by a new instruction is $n+1$ ticks. The resulting heap contains a new object with the appropriate fields. This rule captures exactly the behaviour of object construction. Obviously, unrestricted new instructions can lead to uncontrolled growth of the heap. The crux of our memory management and resource assertion system is to severely restrict where new can be used.

Compared with directly expanding operational semantics, the power of the logic comes in the rules for function and method calls. The rules are similar to Hoare's original rule for parameterless procedures (but lacking preconditions, since we have a logic for partial correctness). For a function call, the rule is:

$$\frac{\Gamma, (f(x_1, \ldots, x_n)\ :\ P) \rhd \mathsf{mbody}_f : \lambda\, E\, h\, h'\, v\, p.\ P(E,\ h,\ h',\ v,\ \mathit{tick}_1(\mathit{call}_1(p)))}{\Gamma \rhd f(x_1, \ldots, x_n) : P} \quad (\textsc{call})$$

This allows one to recursively use the assumption that a call to f satisfies a specification when proving that the unfolded definition of f (mbody_f) indeed satisfies the specification[5]. Again, however, we must adjust the specification to

[5] Because of the restrictions of Grail described in Sect. 5, the actual parameters $x_1, \ldots, x_n$ coincide with the formal parameters as mentioned in mbody_f. For method calls, however, the appropriate rule must instantiate parameters, substituting into the method body.

take account of the resources used in evaluating the function call itself. In this case, the clock advances by one tick and a counter for depth of calls is incremented. In the case of method calls, the depth count corresponds to the number of frames on the framestack; we make a similar count for functions so that we might measure the effects of tail-recursion optimisation. When resource components are combined in let-expressions (corresponding imperatively to sequential composition), the resulting resource component takes the maximum of the depth values in each sub-expression.

As well as the rules corresponding to each syntactic element of Grail, the logic has two essential structural rules:

$$\frac{e : P \in \Gamma}{\Gamma \rhd e : P} \qquad \text{(VAX)}$$

$$\frac{\Gamma \rhd e : P \qquad \forall E\, h\, h'\, v\, p.\ P(E,\, h,\, h',\, v,\, p) \longrightarrow P'(E,\, h,\, h',\, v,\, p)}{\Gamma \rhd e : P'} \qquad \text{(VCONSEQ)}$$

We have established strong results about the bytecode logic, including soundness and (relative) completeness.

Theorem 2. *(Soundness) If* $\Gamma \rhd e : P$ *then* $\Gamma \models e : P$.

Theorem 3. *(Completeness) If* $\models e : P$ *then* $\rhd e : P$.

The obvious statement of these theorems belies the complexity of their proofs. A delicate inductive argument on the depth of evaluation and function call nesting is needed to prove the call and method invocation rules sound. To prove completeness, we used a novel technique based on the admissibility of a cut rule for the logic. Full details of the development appear in [4].

6.3 The Role of the Theorem Prover

If we are to believe in the correctness of our approach, it is an essential requirement that the program logic is sound for the semantics of Grail. Plausibility of the whole framework lies with Theorem 2 above.

Taking this point seriously led us to formalise both the program logic and the semantics of Grail within a theorem prover, providing machine-checked proofs of the above theorems. This follows the approach of several other researchers (most closely, Kleymann [18] and Nipkow [27]). In previous work, this methodology was advocated to increase confidence in meta-theoretical results for program logics, especially soundness, to avoid the possibility of embarrassment (experienced by several authors previously) of proposing unsound or inconsistent logics because of subtle flaws in paper-based arguments[6].

[6] Of course, just as paper-based arguments may be scrutinised by many readers, we should encourage at least the *statements* of our formal theorems and the requisite definitions to be examined by others; we may delegate trust in the proofs themselves to the community's trust in the implementation of the theorem prover.

For our PCC infrastructure, this approach provides also the possibility of using the formally derived proof rules to represent proof evidence directly (or indirectly as the result of applications of tactics). This gives us an easy way of constructing certificates, which may be represented simply as proof script texts for Isabelle with a certain format. To check a certificate, we must extract the claim it makes, and see if the proof successfully replays when applied to the code which has been delivered. An advantage of this approach is that the logic is automatically extensible: to satisfy particular resource policies we may draw on additional stock lemmas which amount to derived proof rules in the logic.

The obvious drawback of using Isabelle proof scripts directly is that Isabelle is now required on the client (code consumer) side, and the size of Isabelle's code base and memory footprint precludes its use on most small devices! Of course, one may change the point at which proof-checking is done to be on a securely-connected and trusted *proof server* (employing the strategy known as *off-device verification*), but our viewpoint is that while our present implementation is ideal for an experimental research prototype, it ought to be replaced by a dedicated proof checker for real deployment. A dedicated checker for our logic could be much smaller and more efficient than a general purpose theorem prover.

6.4 Generating Certificates

While the bytecode logic outlined above enjoys the property of being relatively complete, our experience is that it is rather too low-level for the straightforward construction of certificates. Our initial strategy was to use (formalised versions of) the space assertions obtained from the space type system as specifications of the corresponding compiled functions. Syntax-directed backwards application of the proof rules for the program logic would then generate purely logical verification conditions arising from side-conditions which should be provable automatically.

Unfortunately, the hope that this could be achieved turned out to be too naive. Firstly, the generated verification conditions contained many quantifiers, which were not automatically instantiated using Isabelle's standard solvers. More seriously, stronger invariants than just freelist balance were required, in particular invariants concerning separation of certain data structures in the heap (cf. [31]).

Our solution to both problems is to introduce a notion of *derived assertion* which more directly expresses in the logic the semantic intention of notions from the high-level type system described in Sect. 3. These derived assertions do not encompass the full power of the program logic, only that needed to capture the meaning and invariants underlying the space type system.

More concretely, derived assertions have the form

$$e : \{\!|\, \Delta, m \triangleright T, n, U \,|\!\}$$

Here Δ is a typing environment assigning numerically annotated types as in Sect. 3.5 to variables in e, T is a numerically annotated type, and m, n are numbers. Additionally, U records the set of variables that are actually used in e.

A derived assertion expands into an ordinary assertion in the program logic which expresses the semantic meaning of the typing judgement

$$\Delta, m \triangleright e : T, n$$

in the system from [16]. Intuitively, this semantic meaning is as follows: given a stack S and a heap h such that S and h are type-correct with respect to Δ (when Δ says x : iList then $S[x]$ should indeed point to a linked list in h), then provided e evaluates under S, h to some value v under a purely functional semantics without any space constraints then it will do so in the freelist-based memory model without invoking new provided the freelist has minimum size M. Upon completion the freelist will contain N cells. Here M equals m plus the number of cells obtained from the number of nodes in the data structures pointed to by S according to the numerical annotations in Δ. Likewise, N equals n plus the number of cells obtained from the number of nodes in the data structures pointed to by v according to the numerical annotations in T, plus of course any excess in the initial size of the freelist.

All of this is under the additional assumption that during the evaluation of e no live cell (reachable from the current stack) will be returned to the freelist. As mentioned in Sect. 3.2, this condition is guaranteed under linear typing, and this is currently what is modelled in the derived assertion scheme. That is, in addition to what has been explained already, the derived assertion expresses that the heap regions corresponding to distinct variables listed in U do not overlap.

There are some other technical conditions which turned out to be required. For example, the final heap will equal the initial heap on those locations that are aliased with neither the arguments nor the freelist (i.e. contents of locations not affected by the evaluation should not change).

In total, the definition of the meaning of derived assertions consists of a few hundred lines of Isabelle code. Fortunately, though, we were able to prove once and for all a set of derived proof rules for these derived assertions which roughly follow the typing rules from [16] and allow us to prove derived assertions in a syntax-directed fashion rather than by unfolding definitions. The only non-trivial side-conditions that arise during this syntax-directed backwards application of derived rules are numerical inequalities, all of which turn out to be easily provable provided the derived assertions to start with were constructed from results of the analysis in [17].

Since the analysis [17] speaks about high-level Camelot code whereas the program logic is about compiled Grail, the derived rules sometimes apply to canonical sequences of Grail instructions which arise e.g. from compiling a match construct. It should also be noted that the inference is run on intermediate code in monomorphised A-normal form which is (although syntactically correct Camelot) already quite close to the compiled Grail.

Overall, our approach can be compared to the idea of Foundational Proof-Carrying Code (FPCC) [1], which also takes a formalised machine semantics as a starting point (although for a real machine rather than bytecode), and then derives high-level rules and typing principles. However, whereas FPCC aims at building general derived rules from the ground up, involving complex model

constructions, we have instead started from a specific high-level analysis and derived its type soundness directly.

At the time of writing we have conducted very promising experiments (in particular insertion sort and heap sort) where we prove a concrete space bound by first deducing it from an appropriate derived assertion and then proving the latter by backwards application of derived proof rules, where the choice of rule to use is always clear. In principle it is now indeed possible to generate proof scripts automatically during compilation, giving the correct invariants and auxiliary lemmas to be able to establish derived assertions. To make that happen we have designed an Isabelle tactic that can solve derived assertions automatically, given partial typing information from the Camelot compiler. We are now extending this to more examples and connecting to the Camelot compiler to complete the PCC infrastructure. Full details of the certificate generation strategy are given in [8].

7 Conclusions

The MRG project has delivered a prototype framework for guaranteeing resource security in mobile applications, based on proof-carrying code for the Java Virtual Machine. We have demonstrated the feasibility of PCC for resource verification, based on the technologies developed in the project. As part of this, the MRG work has made specific contributions to the relevant state of the art:

- Type systems for memory management in high-level programming languages. These allow static checks on heap usage and automatic inference of space bounds. For devices with severe memory constraints, this offers the opportunity to raise the current cautious programming model: from manual control of fixed allocation to an automated freelist, without compromising memory safety.
- Resource-exact compilation. Camelot extends the standard task of a compiler – to preserve the meaning of a program – to also reliably preserve resource behaviour. Thus we can use source language types and assertions to correctly describe resource usage for the corresponding executables.
- Grail. Our target language shows not only the practicality of carrying out formal proofs on bytecode; also that a PCC consumer can recover enough structure from the corresponding JVM executable to repeat and verify these proofs.
- The Grail bytecode logic. With its shallow embedding into Isabelle/HOL, this allows us to derive VDM-style assertions of time and space usage for programs. We have formally verified this implementation as sound and complete for a resource-counting operational semantics of Grail.
- The system of derived assertions in the Grail logic. Our logical interpretation of space types provides a toolkit for transferring source-level statements of heap usage into machine-checkable bytecode proofs.
- The MRG architecture. This brings all these components together in an end-to-end PCC framework.

The current system serves as a demonstrator and experimental platform. For practical applications there remain issues of size and performance: although our

certificates are small, the trusted code base is large, and programmed for flexibility rather than speed. The present framework, with a full theorem prover at both producer and consumer side, is sufficient for *wholesale* PCC; for example where a software developer passes certified code to a device vendor for approval. A targeted checker built just for the bytecode logic would be considerably smaller, enough to support some retail PCC, where an individual consumer can check downloaded code on their PC before installing on a smart device. Proof-checking on the device itself remains an extremely challenging goal.

Certain components in the MRG framework are natural targets for future development. Automatic certificate generation, as sketched in the last section, has been demonstrated for individual examples, but we need to extend this success to more general settings. *Resource policies* are a user-level description of what a consumer requires of incoming code. We need to investigate how best to express these, and how to map them into specifications in the bytecode logic. Finally, we require a treatment of termination to complement the bytecode logic, which is a logic of partial correctness (all its assertions are contingent on termination). There are established approaches to proving termination; these are in general different to those for correctness, so decoupling them is appropriate, and moreover leads to simpler rules for method and function invocation.

In future MRG work, we look to broaden our programme to address other scenarios for PCC application. These include different kinds of resources, like network connections or concurrent threads; as well as other application domains, such as microcontrollers for embedded systems, or mobile code in the Grid. In this last case, for example, existing systems like the Globus "Resource Specification Language" [13] state hoped-for space and time requirements, possibly even just from back-of-the-envelope calculations, whereas we would aim for static checks of correctness.

At the language level, we propose to transfer some of our work on type systems and logics across to Java itself. We would do this by expressing resource assertions in the industry standard Java Modeling Language (JML), which already has a certain amount of formal tool support [20].

The continuing progress of the project will be publicized on the MRG website, `http://www.lfcs.ed.ac.uk/mrg`. This carries papers and downloadable software as well as a web-based demonstration.

Acknowledgments

We acknowledge the excellent work of the research assistants and students of the Mobile Resource Guarantees project. Kenneth MacKenzie and Nicholas Wolverson led the implementation work on the Camelot compiler. Kenneth developed the Grail assembler and disassembler tools `gf` and `gdf`; Laura Korte and Matthew Prowse also contributed towards `gf`. Lennart Beringer, Hans-Wolfgang Loidl, Alberto Momigliano, Matthew Prowse and Olha Shkaravska developed theorem proving infrastructure for the formalisation of the bytecode logic and proved results about the logic itself. Michal Konečný worked on the type system for the Camelot language, and Robert Atkey investigated related systems. Steffen Jost implemented the `lfd_infer` tool. Roberto Amadio contributed many useful ideas on resource types.

References

1. A. Appel. Foundational proof-carrying code. In *Proceedings of LICS'01*, pages 247–256. IEEE Computer Society Press, 2001.
2. A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
3. A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr. 1998.
4. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proceedings of the 17th International Conference on Theorem Proving in Higher-Order Logics, (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 34–49. Springer, 2004.
5. D. Aspinall and A. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning*, 31(3–4):261–302, 2003.
6. D. Aspinall and M. Hofmann. Another type system for in-place update. In D. Le Métayer, editor, *Programming Languages and Systems (Proceedings of ESOP 2002)*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.
7. L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, pages 239–250, 1998.
8. L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Towards certificate generation for linear heap consumption. In *Proceedings of the ICALP/LICS Workshop on Logics for Resources, Processes, and Programs (LRPP2004)*, 2004.
9. L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. In *Foundations of Global Computing: Proceedings of the 2nd EATCS Workshop*, number 85.1 in Electronic Notes in Theoretical Computer Science. Elsevier, June 2003.
10. C. Colby, P. Lee, G. C. Necula, F. Blau, K. Cline, and M. Plesko. A certifying compiler for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00), Vancouver, Canada*, 2000.
11. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proc. PDLI '93*, ACM SIGPLAN Notices 28(6), pages 237–247, 1993.
12. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. Retrospective on "The essence of compiling with continuations". In *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*. ACM Press, 2003.
13. I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the IEEE/IFIP 7th International Workshop on Quality of Service*, 1999.
14. M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
15. M. Hofmann. Linear types and non size-increasing polynomial time computation. *Information and Computation*, 183:57–85, 2003.
16. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, New Orleans, 2003.
17. S. Jost. `lfd_infer`: an implementation of a static inference on heap space usage. In *Proceedings of Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE 2004)*, 2004.

18. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1999.
19. C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proc. 5th SCI World Multiconference*, Workshop on Intermediate Representation Engineering for the Java Virtual Machine. Internat. Inst. of Informatics and Systemics, July 2001.
20. G. Leavens, R. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106, 2000.
21. X. Leroy. Bytecode verification on Java smart cards. *Software Practice & Experience*, 32:319–340, 2002.
22. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Jan. 1997.
23. K. MacKenzie and N. Wolverson. Camelot and Grail: resource-aware functional programming for the JVM. In *Trends in Functional Programming*, volume 4, pages 29–46. Intellect, 2004.
24. Microsoft. Overview of the .NET framework. In *.NET Framework Developer's Guide*. `http://msdn.microsoft.com`.
25. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
26. G. Necula. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1997.
27. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.
28. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Jan. 2002.
29. O'Caml Web site. The O'Caml Language. `http://www.ocaml.org`.
30. M. Prowse. Proving Grail resource bounds. University of Edinburgh, May 2003.
31. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.
32. N. Shaylor, D. N. Simon, and W. R. Bush. A Java virtual machine architecture for very small devices. In *Language, Compiler, and Tool Support for Embedded Systems: Proceedings of LCTES '03*, number 38(7) in ACM SIGPLAN Notices, pages 31–41, July 2003.
33. Sun Microsystems. *Java Card 2.2 Platform Specification*, 2003. available online at `http://java.sun.com/products/javacard/specs.html`.
34. M. Wand. Correctness of procedure representations in higher-order assembly language. In *Proc. MFPS '91*, LNCS 298, pages 294–311. Springer, 1992.
35. N. Wolverson. Optimisation and resource bounds in Camelot compilation. Laboratory for Foundations of Computer Science, University of Edinburgh, 2003.
36. N. Wolverson and K. MacKenzie. O'Camelot: Adding objects to a resource aware functional language. In *Trends in Functional Programming*, volume 4, pages 47–62. Intellect, 2004.
37. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

History-Based Access Control and Secure Information Flow

Anindya Banerjee[1,⋆] and David A. Naumann[2,⋆⋆]

[1] Department of Computing and Information Sciences
Kansas State University, Manhattan KS 66506, USA
ab@cis.ksu.edu
[2] Department of Computer Science
Stevens Institute of Technology, Hoboken NJ 07030, USA
naumann@cs.stevens-tech.edu

Abstract. This paper addresses the problem of static checking of programs to ensure that they satisfy confidentiality policies in the presence of dynamic access control in the form of Abadi and Fournet's history-based access control mechanism. The Java virtual machine's permission-based stack inspection mechanism provides dynamic access control and is useful in protecting trusted callees from untrusted callers. In contrast, history-based access control provides a stateful view of permissions: permissions after execution are *at most* the permissions before execution. This allows protection of both callers and callees.

The main contributions of this paper are to provide a semantics for history-based access control and a static analysis for confidentiality that takes history-based access control into account. The static analysis is a type and effects analysis where the chief novelty is the use of security types dependent on permission state. We also show that in contrast to stack inspection, confidential information can be leaked by the history-based access control mechanism itself. The analysis ensures a noninterference property formalizing confidentiality.

1 Introduction

Since Denning and Denning's early work on static certification of secure information flow [6], there have been several advances in specifying static analyses; these advances have been comprehensively summarized in Sabelfeld and Myers's survey [16]. Many of these analyses are given in the style of a security type system that is shown to enforce a *noninterference* property [8]. Noninterference is expressed in terms of a lattice, with $L \leq H$ the canonical example: the property says that output channels labeled L are not influenced by input channels labeled H. The mnemonic is H for high security and L for low: with this interpretation,

⋆ Supported in part by NSF grants CCR-0209205 and CCR-0296182.
⋆⋆ Supported in part by NSF grant CCR-0208984 and New Jersey Commission on Science and Technology.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 27–48, 2005.

noninterference says that public outputs do not reveal secret inputs. But noninterference also formalizes integrity[1]. Security type systems use labels not only for external channels but also for program variables and other internal channels, in order to impose restrictions such as absence of assignment from an H variable to a L one.

Despite the advances, security type systems have not seen much use as noninterference is difficult to achieve in practice for various reasons, e.g., covert channels and declassification. One way to introduce flexibility is to consider security type systems for information flow that take access control into account. As Sabelfeld and Myers note, access control mechanisms, by themselves, control the release of information but not the flow of information once access has been granted [16]. In previous work [4, 2], we studied the access control mechanism of Java [9] and the .NET CLR [10], called stack inspection, and established a connection between authorization of information access and the subsequent flow of the information. With respect to security type systems, the chief technical novelty was the use of a *permission-dependent* security type system and the formalization of noninterference for such a type system. Permissions are typically used to license sensitive operations. Permission-dependent types can express, for example, that to obtain a secret by reading a confidential file a certain permission is required; moreover the read operation yields no secret if the permission is absent. Assumptions about permissions are used in the typing system to allow certain subprograms to be ignored, namely in branches conditioned on permission tests known never to succeed.

In simple terms, the noninterference property guarantees that the access control mechanism is serving correctly to enforce flow policy, in the sense that once access has been granted, there is no subsequent leak of secret information.

This paper continues the investigations of our previous work [4, 2] but considers *history-based* access control proposed by Abadi and Fournet [1]. Stack inspection is designed for extensible systems, where computation proceeds with trusted and untrusted code calling each other. The stack inspection mechanism is useful in providing protection to trusted code when it is called by untrusted code; untrusted code can execute the trusted code with reduced powers – namely, with permissions common to both. This provides protection because the untrusted code is prevented from using trusted code as deputy and executing sensitive operations. However, as Abadi and Fournet note, stack inspection is not useful in providing protection to the *caller*. Thus, if trusted code calls untrusted code and proceeds with the result returned by the latter – using the *same permissions* as it used for the call – undesired results may occur: in proceeding with the result returned by untrusted code, stack inspection forgets that security may depend on

[1] Whereas confidentiality is about what information is leaked, integrity is about what information is corrupted: highly trusted data should not be influenced by untrusted inputs. If we use the same lattice as for confidentiality then integrity is dual to confidentiality. But if we read H as "hacked" and L as "licensed" then to check integrity is to check that H does not influence L. So for simplicity we confine attention to confidentiality.

how, i.e., with what permissions, the result was computed in the first place. This is because the stack frame containing the permissions is popped upon return, so the permissions are no longer available on the stack.

To illustrate the problem, we recall the central example from Abadi and Fournet's paper. The Main method of a trusted class NaiveProgram, with permission *FileIO*, among others, calls method TempFile of an untrusted class, BadPlugIn, whose permissions do not include *FileIO*. The method happens to return a sensitive document. Next, the Delete method of class File is called, with the sensitive document as argument. Class File has all permissions, and method Delete first checks whether *FileIO* is present in the currently enabled permissions; if so, the document is deleted, otherwise, the method aborts.

In a stack inspection régime, the call NaiveProgram.Main results in deletion of the sensitive document. This is because Delete's code is executed with permissions common to NaiveProgram and File, so the check for *FileIO* succeeds.

In a history-based régime, the document survives, because we track permission state both before and after the call. Where stack inspection is functional, in the sense that the security context is passed as an argument to sub-commands including method invocations, the history based mechanism is imperative, in the sense that the security context is a (special) state variable. The call to TempFile returns the document together with permissions in the intersection of NaiveProgram and BadPlugIn, so that *FileIO* is excluded. Effectively, the history of how the return result was created is recorded. Now the call to Delete takes place with this reduced set of permissions; the check fails.

The contributions of this paper are to formalize the informal development of common programming patterns of history-based access control in [1] and to provide a type-based analysis for secure information flow that takes history-based access control based on such patterns into account. Some familiarity with our previous work [3, 4, 2] will be helpful; in fact, readers familiar with our previous work will readily observe the substantial overlap with that work. The modest variation in this work is that the type-based analysis rules also involve an effect analysis; this effect analysis tracks not only assumptions about the initial permission state, as in stack inspection, but also provides a conservative approximation of the final permission state. A second difference is that, in contrast to the stack inspection mechanism, the history-based mechanism is itself a covert channel subject to nontrivial attacks. To account for such flows, our rule for method calls in high contexts requires that the invoked method has certain permissions; by contrast, the other rules require absence of permissions in certain contexts.

One of our goals is to demonstrate the flexibility of the framework described in our previous work by showing how to handle a different access control mechanism. To reinforce the similarity, the technical development in this paper is structured as in [2]. Proof cases are omitted since they are easily adapted from corresponding ones in [2].

In passing, we note that different access control mechanisms are possible. Our previous work on stack inspection is motivated in part because it is widely

deployed. The limitations of stack inspection with respect to method calls were explained by Abadi and Fournet [1] who proposed history-based access control as a way out. Their mechanism is designed to be similar to stack inspection which increases its potential for use in practice and thus it merits study. It would also be interesting to seek a more general notion of access control which subsumes these mechanisms and others. There may be some interesting parallels between such mechanisms and recent work on resource usage analysis, history effects, etc. [12, 14, 13].

The rest of the paper. Section 2 introduces code-based access control via an example and discusses stack-based and history-based access control. It also explains the security type formalism used in type-based information flow analyses. Section 3 is the key section of the paper. It shows how access control can be used to provide more fine-grained confidentiality policies. It also shows how, in contrast to stack inspection, the history-based access control mechanism can itself be employed to leak secrets. Section 4 formalizes the syntax and semantics of the object-oriented language we study. Section 5 gives a type-based static analysis that enforces confidentiality. Section 6 shows the analysis at work on the central example in Abadi and Fournet's paper [1]. Section 7 and Section 8 give the technical details of the static analysis; the latter section states the main result that a program deemed safe by the analysis satisfies the noninterference property. Section 9 concludes.

2 Access Control and Information Flow

Access control via stack inspection. In the Java access control mechanism [9], each class C has a set $Auth\,C$ of permissions associated with it; this comprises a local access control policy. A typical policy grants few permissions to code from remote sites and many to code residing on the local disk. The most interesting policies concern trusted remote sites: Code which has been cryptographically authenticated as originating at a trusted site may be granted particular permissions.

Permission checks are used to guard sensitive operations. Following previous work [7], we refrain from modeling exceptions and instead consider a construct, **test** p **then** S_1 **else** S_2, which checks for permission p, executing S_1 if the check succeeds and S_2 if it fails.

Example. We consider the following example from Section 4.1 of Abadi and Fournet's paper [1], adapted to our notation (e.g., type **unit** for `void`).

```
class BadApplet extends Object { // some permissions but not FileIO
  unit Main() {
    result:= NaiveLibrary.CleanUp(..."password file"...); }}
```

The comment indicates our assumption about the static access policy: BadApplet does not have permission *FileIO*, whereas NaiveLibrary and File below are trusted classes with all permissions.

```
class NaiveLibrary extends Object { // all permissions
  unit CleanUp((string, L) s) {
    File.Delete(s);}}

class File extends Object { // all permissions
  unit Delete((string, L) s) {
    test FileIO then Win32.Delete(s) else abort;} }
```

The call, BadApplet.Main() will abort. The call to NaiveLibrary.CleanUp will occur in a context with permissions common to NaiveLibrary and BadApplet and this context does not contain *FileIO*. As CleanUp calls Delete, the body of Delete will also be executed with permissions common to the current permissions (without *FileIO*) and Delete; hence the test in Delete fails – the password file survives. This is a situation where stack inspection is satisfactory, and the history based mechanism works the same way.

History based access control. At runtime, both stack inspection and the history based mechanism involve a set of currently enabled permissions; it is a subset of the statically authorized permissions of the class of the currently-executing code. When a method is invoked, the initial permission set for the method body is the intersection of the caller's current set and the static permissions of the called code. (Note that it is the class of the dynamically dispatched code that matters, not the class of the target object.) In stack inspection, the method call has no effect on the current permission set of the caller. In the history based mechanism, the caller's permissions become the intersection of their initial value with the final permissions of the called method.

Both mechanisms include means to test and branch on the currently enabled permissions, which we model with the **test** construct.

Permissions P get enabled by the construct **enable** P **in** S in stack inspection and a similar construct **grant** P **in** S for the history-based mechanism; in both cases, what gets added to the current permission set is the intersection of P with the statically authorized permissions, $Auth\, C$, of the current code's class, C.

Whereas the permission set on termination of **enable** P **in** S is the same as its initial value, the history based construct **grant** P **in** S deals with the final permissions of S. Specifically, the final permissions of **grant** P **in** S are the intersection of its initial permissions and the final ones of S. Together with the behavior of method calls described above, this ensures that for any command, the final permission set is a subset of its initial value (see Lemma 1). (In stack inspection the final permissions are equal to the initial ones, so permission passing can be modeled as just a parameter in the semantics.)

The history based mechanism needs a construct, **accept**, to allow a privileged caller to retain its permissions after calling less privileged code (e.g., to delete a file named by an untrusted applet, after checking that the named file is not important). The effect of **accept** P **in** S is to execute S with the current permissions Q. This results in a final permission set Q' from S, which may be a proper subset of Q. The permission set after **accept** P **in** S becomes $Q' \cup (Q \cap P)$.

Note that constructs **grant** and **accept** abstract two useful programming patterns for modifying permissions in code. Abadi and Fournet show how they can be implemented using low-level constructs [1]. For purposes of formal analysis, however, we will stick to **grant** and **accept** in the sequel.

Checking information flow using security types. Based on earlier work on static certification of information flow by Denning and Denning [6], the idea developed by Volpano *et al.* [18] is to label not only inputs and outputs but also variables and parameters by security types, for example replacing a variable declaration $x:T$ by $x:(T,\kappa)$ where κ is the security level. As usual, we consider the representative two-element lattice $L \leq H$ of levels. Syntax-directed typing rules specify conditions that ensure secure flow. Overt flows, like an assignment of an H-variable to an L-variable, are disallowed by the typing rules for assignment, argument passing, etc. To preclude covert flow via control flow, commands are given types $com\,\kappa$ with the meaning that all assigned variables have at least level κ. For a conditional, **if** e **then** S_1 **else** S_2, with e high, both S_1 and S_2 are required to have type $com\,H$.

In an object-oriented language, covert flow also happens via dynamically dispatched method call. Moreover, there is the possibility of observing differing behavior of the allocator if objects allocated conditionally are accessible. Such issues are treated in [15, 3]. In [3], commands are given types $(com\,\kappa_1, \kappa_2)$ where κ_1 is a lower bound on the level of assigned variables and κ_2 is a lower bound on the heap effect (field assignments and newly allocated objects). Annotated arrow types are used for modular checking in the case of methods (or procedures or functions [11]): the type $(T,\kappa_1)\text{–}\langle\kappa_2\rangle\!\rightarrow(U,\kappa_3)$ designates an assumed input level at most κ_1; on this assumption, the heap effect (min level of fields written) is at least κ_2 and result level at most κ_3. A method body is checked with respect to its type, which is used as an assumption for checking method calls. As with ordinary types in Java-like languages, the same type is used for all overriding declarations.

3 Using Access Control for Confidentiality

As mentioned in the introduction, a primary aim of our work is the static checking of method bodies to ensure that they satisfy confidentiality policies. However, we also want our confidentiality policies to be flexible enough to admit a large number of programs. We allow a method to be given several types, to allow different information flow policies to be imposed for callers with different permissions. We explain the idea with the motivating example from our previous work [4, 2].

Consider a trusted class Kern, having permissions *stat* and *sys*, and with a method getStatus that can be called in more than one context. If called by untrusted code, getStatus returns public (L) information. Trusted code, however, can obtain private (H) information.

```
class Kern extends Object {
  String Hinfo; // H
  String Linfo; // L
  String getHinfo() { // type () → H
    test sys then result := self.Hinfo else abort }
  String getStatus() { // types () −⟨∅⟩⟶ H and () −⟨{stat}⟩⟶ L
    test stat
    then enable sys in result := self.getHinfo()
    else result := self.Linfo }
... "other methods that manipulate Linfo and Hinfo"}
```

Method getHinfo is useful only to callers with permission sys. Because the security type of self.Hinfo is H, it can only be assigned to a H variable. In this case, the body of getHinfo, it is the special variable, result, that gives the method result. So, for the policy expressed by the type $() \to H$, the code can be accepted under a Smith-Volpano style analysis [18].

For getStatus, the Smith-Volpano analysis will also insist that result be typed H, so that it satisfies the policy is $() \to H$. But getStatus is useful both for callers with permission $stat$ and for those without; only the former obtain H info. So we parameterize the policy by permission $stat$: if called in a context that does *not* have permission $stat$, getStatus returns L; otherwise it returns H. The use of negative permissions allows types that mention only permissions relevant to implementations of the method.

More formally, method types in [4, 2] have the form

$$\kappa_0, \bar{\kappa} - \langle P; \kappa \rangle \rightarrowtail \kappa_1 \tag{1}$$

This means: suppose the level of self is at most κ_0, and the level of the parameters of the method have level at most $\bar{\kappa}$, and the method is called in a context with permissions disjoint from P. Then the result has level at most κ_1, and fields written have level at least κ.

Using this notation, getStatus can be assigned both types $L, ()-\langle\{stat\}; H\rangle\rightarrowtail L$ and $L, ()-\langle\varnothing; H\rangle\rightarrowtail L$.

In the BadApplet example in section 2, Delete can be assigned both types $L, L-\langle\{FileIO\}; H\rangle\rightarrowtail()$ and $L, L-\langle\varnothing; H\rangle\rightarrowtail()$. The body of BadApplet.Main needs to be typechecked in a context where $FileIO$ is absent. Hence BadApplet.Main can be assigned the type $L, ()-\langle\{FileIO\}; H\rangle\rightarrowtail()$.

A method body must be checked against each of its declared types. A particular type gives an assumption that certain permissions are absent, and under this assumption certain branches are known not to be taken. Typically the ignored branches involve H assignments and thus ignoring these branches allows the result to be considered L.

History based. In (1), the levels $\kappa_0, \bar{\kappa}$ can be read as the input channel and κ_1 as the output level. For a sound static analysis, the type also tracks the write effect κ and the corresponding judgement for commands gives a command level. In the sequel we carry out this same idea for the history based mechanism, for

which we have to track the update effect on permissions. Thus method types and command judgements include an upper bound on the final permissions; this is made precise in section 5.

Unlike stack inspection, the history based access mechanism introduces a covert channel. The reason is that permission changes made in the context of a high conditional can persist outside the scope of that conditional. If some command S changes the final permission state then it can be used to leak information by executing S under a H guard and then, after the conditional, updating L state based on testing whether the permission state is changed. In [2] we point this out in regard to a variation on **enable** that does not have a scoped sub-command in which the permission is enabled.

The history based mechanism does not allow persistent increase of permissions, but it allows persistent decrease. Here is an example attack, using an explicit **grant** to establish some permission which may then be decreased by a suitable method call.

```
grant p // assuming p statically authorized
in
  if H-exp
  then e.m() // assuming p not authorized for m
  else skip;
  test p then L-var := "H-exp is false" else L-var := "H-exp is true"
```

A variation uses the **accept** statement to allow m to be invoked in every case.

```
grant p // assuming p statically authorized
in
  if H-exp
  then e.m() // assuming p not authorized for m
  else accept p in e.m();
  test p then L-var := "H-exp is false" else L-var := "H-exp is true"
```

The security typing rule for method call disallows any writes to L fields by m – this is enforced by requiring that its heap effect be H, as it is called in the scope of a high guard. Because any code is allowed to invoke **grant**, **test**, and **accept**, the (implicit) variable holding the current permissions must be treated as L and thus changes to it too must be disallowed in H conditionals. It turns out that this can be achieved in the static analysis by requiring that permissions that can be enabled at the site of the call must be included in the static permissions of all classes that provide an implementation of m. For a method call in a H command, any implementation that could be dispatched to at that method call site should be *at least as trusted as* the caller. (This condition has no counterpart in [2].)

Just as a method type serves to specify all implementations thereof, one can imagine stipulating the permissions that all implementations are required to have (a link-time requirement). But in this paper we express the restriction explicitly in the call rule.

Table 1. Grammar.

T	$::= \mathbf{bool} \mid \mathbf{unit} \mid C$	data type; C ranges over class names
CL	$::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{T}\ \bar{f};\ \bar{M}\ \}$	public fields $\bar{f}$, public methods $\bar{M}$
M	$::= T\ m(\bar{T}\ \bar{x})\ \{S\}$	method; result type T, param. types $\bar{T}$
S	$::= x := e \mid \mathbf{if}\ e\ \mathbf{then}\ S\ \mathbf{else}\ S \mid S;\ S$	assign to variable; conditional; sequence
	$\mid\ T\ x := e\ \mathbf{in}\ S \mid x := e.m(\bar{e})$	local variable block; method call
	$\mid\ e.f := e \mid x := \mathbf{new}\ C$	assign to field; construct object
	$\mid\ \mathbf{grant}\ P\ \mathbf{in}\ S$	enable permissions
	$\mid\ \mathbf{accept}\ P\ \mathbf{in}\ S$	accept permissions
	$\mid\ \mathbf{test}\ P\ \mathbf{then}\ S\ \mathbf{else}\ S$	branch on permissions
e	$::= x \mid \mathbf{null} \mid \mathbf{true} \mid \mathbf{false}$	variable, constant
	$\mid\ e.f \mid e = e \mid e\ \mathbf{is}\ C \mid (C)\ e$	field access; equality test; type test; cast

4 Language

This section formalizes the sequential class-based language for which our results are given. It is adapted from [2], but with the history-based constructs and with the semantics changed to return the permission state from commands and methods as discussed in Section 2. We assume given a finite set, $Perms$, of permissions. The semantics is given with respect to a given function $Auth : ClassNames \to \mathcal{P}(Perms)$ that specifies access policy. The semantic definitions are independent of any particular information flow policy.

4.1 Syntax

The grammar is given by Table 1. It is based on given sets of class names (with typical element C), field names (f), method names (m), and variable/parameter names x (including distinguished names "self" and "result" for the target object and return value). Identifiers like $\bar{T}$ with bars on top indicate finite lists, e.g., $\bar{T}\ \bar{f}$ stands for a list $\bar{f}$ of field names with corresponding types $\bar{T}$. We let P, Q, R range over sets of permissions.

We include unrestricted recursion but omit loops, super calls, and constructor methods.

A complete program is given as a *class table*, CT, that associates each declared class name with its declaration. The typing rules make use of auxiliary notions that are defined in terms of CT, so the typing relation $\vdash$ depends on CT but this is elided in the notation. Because typing of each class is done in the context of the full table, methods can be recursive (mutually) and so can field types.

The subtyping relation $\leq$ on types is defined as follows. For base types, $\mathbf{bool} \leq \mathbf{bool}$ and $\mathbf{unit} \leq \mathbf{unit}$. For classes C and D, we define $C \leq D$ iff either $C = D$ or the class declaration for C is $\mathbf{class}\ C\ \mathbf{extends}\ B\ \{\ \ldots\ \}$ for some $B \leq D$. The typing rules are syntax-directed: Subsumption is built into the rules

Table 2. Selected typing rules for commands.

$$\frac{\Gamma \vdash e_1 : C \qquad (f : T) \in \mathit{fields}\, C \qquad \Gamma \vdash e_2 : U \qquad U \leq T}{\Gamma \vdash e_1.f := e_2}$$

$$\frac{\Gamma \vdash e : D \qquad \mathit{mtype}(m, D) = \bar{T} \to T \qquad T \leq \Gamma\, x \qquad \Gamma \vdash \bar{e} : \bar{U} \qquad \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self}}{\Gamma \vdash x := e.m(\bar{e})}$$

$$\frac{P \subseteq \mathit{Perms} \qquad \Gamma \vdash S}{\Gamma \vdash \mathbf{grant}\ P\ \mathbf{in}\ S} \qquad \frac{P \subseteq \mathit{Perms} \qquad \Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash \mathbf{test}\ P\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2}$$

rather than appearing as a separate rule, so that the semantics can be defined by recursion on typing derivations.

Some auxiliary notations: let $CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}_1\ \bar{f};\ \bar{M}\ \}$ and let M be in the list $\bar{M}$ of method declarations, with $M = T\ m(\bar{T}_2\ \bar{x})\{S\}$; let $\mathit{mtype}(m, C) = \bar{T}_2 \to T$ record typing information and let $\mathit{pars}(m, C) = \bar{x}$ record the parameter names. Let $\mathit{super}\, C = D$. For fields, we define $\mathit{fields}\, C = \bar{f} : \bar{T}_1 \cup \mathit{fields}\, D$ and assume $\bar{f}$ is disjoint from the names in $\mathit{fields}\, D$. The built-in class Object has no methods or fields. If m is inherited in C from B then $\mathit{mtype}(m, C)$ is defined to be $\mathit{mtype}(m, B)$, so that $\mathit{mtype}(m, C)$ is defined iff m is declared or inherited in C.

A class table is well formed if each of its method declarations is well formed according to the following rule.

$$\frac{\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S \qquad \begin{array}{c}\mathit{mtype}(m, \mathit{super}\, C) \text{ is undefined or equals } \bar{T} \to T \\ \mathit{pars}(m, \mathit{super}\, C) \text{ is undefined or equals } \bar{x}\end{array}}{C \vdash T\ m(\bar{T}\ \bar{x})\{S\}}$$

A typing environment Γ is a finite function from variable names to types, written with the usual notation $x : T$. A judgement of the form $\Gamma \vdash e : T$ says that e has type T in the context of a method of class $\Gamma\,\mathsf{self}$, with parameters and local variables declared by Γ. A judgement $\Gamma \vdash S$ says that S is a command in the same context. Note that access policy has no influence on typing, though of course it does influence semantics. Typing rules for some commands appear in Table 2. The rule for **accept** is the same as for **grant**. For other elided rules we refer the reader to our earlier papers [4, 2].

4.2 Semantics

The state of a method in execution is comprised of a *heap* h, which is a finite partial function from locations to object states, a *store* η, which assigns locations and primitive values to local variables and parameters, and the currently enabled *permissions* P_1. Every store of interest includes the distinguished variable self

Table 3. Semantic domains, for given policy *Auth*. We write *dom* and *rng* for the domain and range of a function.

$\theta ::= T$	values of type T
$\mid \Gamma$	store (maps variables to values)
$\mid \mathit{state}\, C$	object state (maps fields to values)
$\mid \mathit{Heap}$	heap (maps locations to object states) with no dangling loc.
$\mid \mathit{Heap} \otimes \Gamma \otimes \mathcal{P}(\mathit{Perms})$	(global) states with no dangling locations
$\mid \mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms})$	triples (h, d, P) where value d is not a dangling loc. w.r.t. h
$\mid \theta_\perp$	lifting
$\mid \mathit{perms}\, C$	permission sets authorized for C
$\mid (C, \bar{x}, \bar{T} \to T)$	method of C with parameters $\bar{x} : \bar{T}$ and return type T
$\mid \mathit{MEnv}$	method environments

$$
\begin{aligned}
[\![\mathbf{bool}]\!] &= \{\mathit{true}, \mathit{false}\} \\
[\![\mathbf{unit}]\!] &= \{\mathit{it}\} \\
[\![C]\!] &= \{\mathit{nil}\} \cup \{\ell \mid \ell \in \mathit{Loc} \wedge \mathit{type}\, \ell \leq C\} \\
[\![\Gamma]\!] &= \{\eta \mid \mathit{dom}\, \eta = \mathit{dom}\, \Gamma \wedge \eta\, \mathsf{self} \neq \mathit{nil} \wedge \forall x \in \mathit{dom}\, \eta \bullet \eta\, x \in [\![\Gamma\, x]\!]\} \\
[\![\mathit{state}\, C]\!] &= \{s \mid \mathit{dom}\, s = \mathit{dom}(\mathit{fields}\, C) \wedge \forall (f : T) \in \mathit{fields}\, C \bullet s\, f \in [\![T]\!]\} \\
[\![\mathit{Heap}]\!] &= \{h \mid \mathit{dom}\, h \subseteq_{\mathit{fin}} \mathit{Loc} \wedge \mathit{closed}\, h \wedge \forall \ell \in \mathit{dom}\, h \bullet h\ell \in [\![\mathit{state}\ (\mathit{type}\ \ell)]\!]\} \\
&\quad \text{where } \mathit{closed}\, h \text{ iff } \mathit{rng}\, s \cap \mathit{Loc} \subseteq \mathit{dom}\, h \text{ for all } s \in \mathit{rng}\, h \\
[\![\mathit{Heap} \otimes \Gamma \otimes \mathcal{P}(\mathit{Perms})]\!] & \\
&= \{(h, \eta, P) \mid h \in [\![\mathit{Heap}]\!] \wedge \eta \in [\![\Gamma]\!] \wedge P \subseteq \mathit{Perms} \wedge \mathit{rng}\, \eta \cap \mathit{Loc} \subseteq \mathit{dom}\, h\} \\
[\![\mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms})]\!] & \\
&= \{(h, d, P) \mid h \in [\![\mathit{Heap}]\!] \wedge d \in [\![T]\!] \wedge P \subseteq \mathit{Perms} \wedge (d \in \mathit{Loc} \Rightarrow d \in \mathit{dom}\, h)\} \\
[\![\theta_\perp]\!] &= [\![\theta]\!] \cup \perp \quad (\text{where } \perp \text{ is some fresh value not in } [\![\theta]\!]) \\
[\![\mathit{perms}\, C]\!] &= \{P \mid P \subseteq \mathit{Auth}\, C\} \\
[\![C, \bar{x}, \bar{T} \to T]\!] &= [\![\mathit{Heap} \otimes (\bar{x} : \bar{T}, \mathsf{self} : C) \otimes \mathcal{P}(\mathit{Perms})]\!] \to [\![(\mathit{Heap} \otimes T \otimes \mathcal{P}(\mathit{Perms}))_\perp]\!] \\
[\![\mathit{MEnv}]\!] &= \{\mu \mid \forall C, m \bullet \mu C m \text{ is defined iff } \mathit{mtype}(m, C) \text{ is defined,} \\
&\qquad \text{and } \mu C m \in [\![C, \mathit{pars}(m, C), \mathit{mtype}(m, C)]\!] \text{ if } \mu C m \text{ defined } \}
\end{aligned}
$$

which points to the target object. A command denotes a function from initial state to either a final state or the error value $\perp$. States are self-contained in the sense that all locations in fields and in variables are in the domain of the heap.

For locations, we assume that a countable set *Loc* is given, along with a distinguished entity *nil* not in *Loc*. We treat object states as mappings from field names to values. To track the object's class we assume given a function $\mathit{type} : \mathit{Loc} \to \mathit{ClassNames}$ such that for each C there are infinitely many locations ℓ with $\mathit{type}\ \ell = C$. We assume that, like *nil*, the three primitive values *it*, *true*, and *false* are not in *Loc*. The semantic definitions and results are given for an arbitrary allocator. An *allocator* is a location-valued function *fresh* such that $\mathit{type}(\mathit{fresh}(C, h)) = C$ and $\mathit{fresh}(C, h) \notin \mathit{dom}\, h$, for all C, h.

Methods are associated with classes, in a *method environment*. For any data type T, the domain $[\![T]\!]$ is the set of values of type T. For any typing environment Γ, $[\![\Gamma]\!]$ is the set of stores assigning values of appropriate type to the variables

in $dom\,\Gamma$. There are several other domains for which there is no corresponding notation in the syntax. Such semantic categories, θ, together with semantic domains, $[\![\theta]\!]$, for each category are defined in Table 3; T and Γ are included in θ. In a language like Java with garbage collection and without pointer arithmetic, dangling locations (those not in the domain of the heap) never occur in program states or as expression values. Capturing this in the semantics is the purpose of the special cartesian products $Heap \otimes \Gamma \otimes \mathcal{P}(Perms)$ and $Heap \otimes T \otimes \mathcal{P}(Perms)$.

For expressions and commands, the semantics is defined only for derivable typing judgements. The meaning of an expression $\Gamma \vdash e : T$ is a function $[\![Heap \otimes \Gamma]\!] \to [\![T_\perp]\!]$ that takes $(h, \eta) \in [\![Heap \otimes \Gamma]\!]$ and returns either a value $d \in [\![T]\!]$, such that $(h, d) \in [\![Heap \otimes T]\!]$, or the improper value $\perp$ which represents errors. The errors are null dereferences and cast failure; the other expression constructs are strict in $\perp$. We omit the semantics of expressions and refer the reader to our previous work [2].

The meaning of a command $\Gamma \vdash S$ is a function

$$[\![MEnv]\!] \to [\![Heap \otimes \Gamma \otimes perms(\Gamma\,\mathsf{self})]\!] \to [\![(Heap \otimes \Gamma \otimes perms(\Gamma\,\mathsf{self}))_\perp]\!] \quad (2)$$

that takes a method environment μ (see below)and a state (h, η, R), where the enabled permissions $R \in perms(\Gamma\mathsf{self})$; it returns a possibly updated state together with the updated permissions, or $\perp$ which indicates divergence or error. In history-based access control, permissions get updated, e.g., in method calls. The semantics of command, in Table 4, is defined by recursion on the typing derivation. A nontrivial proof obligation is that if $\Gamma \vdash S$ is derivable then its semantics satisfies (2), which embodies the important property that if the permissions before execution are included in $Auth(\Gamma\mathsf{self})$, permissions after execution are also included in $Auth(\Gamma\mathsf{self})$.

The main distinction between stack inspection and history-based access control is that in the former, permission state (i.e., enabled permissions) after evaluation equals the permission state before evaluation; in the latter, permission state after evaluation is *at most* the permission state before evaluation. Accordingly, the semantics of commands satisfies the following property, shown by a structural induction on commands.

Lemma 1. *Suppose* $(h_0, \eta_0, Q_0) = [\![\Gamma \vdash S]\!]\mu(h, \eta, R)$. *Then* $Q_0 \subseteq R$.

A method environment μ maps each class name C and method name m (declared or inherited in C) to a meaning $\mu\, C\, m$ which is an element of $[\![C, \bar{x}, \bar{T} \to T]\!]$, i.e., $[\![Heap \otimes \Gamma \otimes \mathcal{P}(Perms)]\!] \to [\![(Heap \otimes T \otimes \mathcal{P}(Perms))_\perp]\!]$ where T is the return type and $\Gamma = \mathsf{self} : C, \bar{x} : \bar{T}$ is the parameter store with $\bar{x} = pars(m, C)$. The result from a method, if not $\perp$, is a triple (h, d, Q) with d in $[\![T]\!]$ and Q in $\mathcal{P}(Perms)$ such that, if d is a location then d is in the domain of h.

For a method declaration $M = T\ m(\bar{T}\ \bar{x})\{S\}$ in class C, define

$$\begin{array}{l}
[\![M]\!]\mu(h, \eta, R) \\
\quad = \mathsf{let}\ R' = R \cap Auth\, C\ \mathsf{in}\ \mathsf{let}\ \eta_1 = [\eta \mid \mathsf{result} \mapsto \mathit{default}]\ \mathsf{in} \\
\qquad \mathsf{let}\ (h_0, \eta_0, Q_0) = [\![\bar{x} : \bar{T}, \mathsf{self} : C, \mathsf{result} : T \vdash S]\!]\mu(h, \eta_1, R')\ \mathsf{in}\ (h_0,\ \eta_0\, \mathsf{result}, Q_0)
\end{array}$$

Table 4. Semantics of selected commands, for given policy *Auth* and allocator *fresh*. The metalanguage construct, $\mathsf{let}\ d = E_1\ \mathsf{in}\ E_2$, has the following meaning: If the value of E_1 is $\perp$ then that is the value of the entire let expression; otherwise, its value is the value of E_2 with d bound to the value of E_1. Function update or extension is written, e.g., $[\eta \mid x \mapsto d]$.

$$
\begin{array}{l}
[\![\Gamma \vdash e_1.f := e_2]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ \ell = [\![\Gamma \vdash e_1 : C]\!](h,\eta)\ \mathsf{in} \\
\qquad \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{let}\ d = [\![\Gamma \vdash e_2 : U]\!](h,\eta)\ \mathsf{in}\ ([h \mid \ell \mapsto [h\,\ell \mid f \mapsto d]], \eta, R) \\
[\![\Gamma \vdash x := e.m(\bar{e})]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ \ell = [\![\Gamma \vdash e : D]\!](h,\eta)\ \mathsf{in}\ \mathsf{if}\ \ell = \mathit{nil}\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{let}\ \bar{x} = \mathit{pars}(m,D)\ \mathsf{in} \\
\qquad \mathsf{let}\ \bar{d} = [\![\Gamma \vdash \bar{e} : \bar{U}]\!](h,\eta)\ \mathsf{in}\ \mathsf{let}\ \eta_1 = [\bar{x} \mapsto \bar{d}, \mathsf{self} \mapsto \ell]\ \mathsf{in} \\
\qquad \mathsf{let}\ (h_0, d_0, Q_0) = \mu(\mathit{type}\,\ell)m(h,\eta_1,R)\ \mathsf{in}\ (h_0, [\eta \mid x \mapsto d_0], R \cap Q_0) \\
[\![\Gamma \vdash \mathbf{grant}\ P'\ \mathbf{in}\ S]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ (h_0,\eta_0,Q_0) = [\![\Gamma \vdash S]\!]\mu(h,\eta,R \cup (P' \cap \mathit{Auth}(\Gamma\,\mathsf{self})))\ \mathsf{in}\ (h_0,\eta_0,R \cap Q_0) \\
[\![\Gamma \vdash \mathbf{accept}\ P'\ \mathbf{in}\ S]\!]\mu(h,\eta,R) \\
\quad = \mathsf{let}\ (h_0,\eta_0,Q_0) = [\![\Gamma \vdash S]\!]\mu(h,\eta,R)\ \mathsf{in}\ (h_0,\eta_0,Q_0 \cup (P' \cap R \cap \mathit{Auth}(\Gamma\,\mathsf{self}))) \\
[\![\Gamma \vdash \mathbf{test}\ P\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2]\!]\mu(h,\eta,R) \\
= \quad \mathsf{if}\ P \subseteq R\ \mathsf{then}\ [\![\Gamma \vdash S_1]\!]\mu(h,\eta,R)\ \mathsf{else}\ [\![\Gamma \vdash S_2]\!]\mu(h,\eta,R)
\end{array}
$$

The semantics of a class table CT is a method environment, written $[\![CT]\!]$, given as a least upper bound. Specifically, $[\![CT]\!] = \mathit{lub}\,\mu$ where the ascending chain $\mu \in \mathbb{N} \to [\![\mathit{MEnv}]\!]$ is defined as follows.

$$
\begin{array}{l}
\mu_0\,C\,m = \lambda(h,\eta,R) \bullet \perp \\
\mu_{j+1}\,C\,m = [\![M]\!]\mu_j \quad \text{if } m \text{ is declared as } M \text{ in } C \\
\mu_{j+1}\,C\,m = \mu_{j+1}\,B\,m \quad \text{if } m \text{ is inherited from } B \text{ in } C
\end{array}
$$

5 Safety

The syntactic property given by static analysis is called *safety*. The analysis is specified by a typing system.

In this section we annotate the syntax of Section 4 with security labels. Where types T occur in declarations of fields and local variables, we use pairs (T,κ) where κ is a security level, L or H. Such a pair, written τ, is called a *security type*. The grammar is revised as follows.

$$
CL \ ::= \mathbf{class}\ C\ \mathbf{extends}\ C\ \{\ \bar{\tau}\ \bar{f};\ \bar{M}\ \} \qquad\qquad S \ ::= \ \ldots \mid \tau\ x := e\ \mathbf{in}\ S \mid \ldots
$$

Note that there is no change for cast and test.

We refrain from giving concrete syntax for the security types of method parameters, results, and effects. By analogy with the auxiliary function *mtype* which gives the declared type of a method (see Section 4.1), we assume that a function *smtypes* is given. It may assign multiple security types for a method, each of the form $\kappa, \bar{\kappa} \text{–} \langle P; \kappa_1; Q \rangle \rightarrowtail \kappa_2$. The intended meaning is as follows: if the

method is called with arguments compatible with $\bar\kappa$, target object compatible with κ, and enabled permissions disjoint from P, then the heap effect is $\geq \kappa_1$ and the result level $\leq \kappa_2$ and the enabled permissions after the call are disjoint from Q.

There is an ordering on method typings $\kappa, \bar\kappa{-}\langle P; \kappa_1; Q\rangle{\rightarrowtail}\kappa_2$. It is contravariant on inputs $\kappa, \bar\kappa$ and P and on assignables κ_1, covariant on the result value κ_2 and Q.

Definition 1 (subtyping). $\kappa, \bar\kappa{-}\langle P; \kappa_1; Q\rangle{\rightarrowtail}\kappa_2 \;\leq\; \kappa', \bar\kappa'{-}\langle P'; \kappa_1'; Q'\rangle{\rightarrowtail}\kappa_2'$ *iff* $\kappa' \leq \kappa$, $\bar\kappa' \leq \bar\kappa$, $P \subseteq P'$, $\kappa_1' \leq \kappa_1$, $Q' \subseteq Q$, *and* $\kappa_2 \leq \kappa_2'$. □

Note that P, Q are interpreted negatively, so the conditions $P \subseteq P'$ and $Q' \subseteq Q$ are effectively contravariant and covariant respectively.

Definition 2 (annotated class table). *An annotated class table is a class table with annotations according to the grammar above, together with a partial function smtypes satisfying the following conditions. First, smtypes(m, C) is defined iff mtype(m, C) is defined. Second, if smtypes(m, C) is defined then it is a* non-empty *set of annotations of the form $\kappa, \bar\kappa{-}\langle P; \kappa_1; Q\rangle{\rightarrowtail}\kappa_2$. Third, if $C \leq D$ and mtype(m, D) is defined then smtypes(m, C) = smtypes(m, D).* □

Note that we do not require $P \subseteq \mathit{Auth}\, C$ or $Q \subseteq \mathit{Auth}\, C$. A method may be declared in one class and inherited or overridden in a subclass with different permissions. The third condition allows us to reason about method calls in terms of the static type of a called method, because any implementation that can be invoked by dynamic dispatch is checked with respect to the same security types.

We use the symbol † to erase annotations: $(T, \kappa)^\dagger = T$, and this extends to erasure for typing environments, commands, and method declarations in an obvious way.

Definition 3 (safe class table and method declaration). *An annotated class table CT is* safe *provided that each class satisfies the rule*

$$\frac{C\ \textbf{\textit{extends}}\ D \vdash M \text{ for each } M \in \bar M}{\vdash \textbf{\textit{class}}\ C\ \textbf{\textit{extends}}\ D\ \{\ \bar\tau\ \bar f;\ \bar M\ \}}$$

The hypothesis of this rule requires that each method declaration be checked with respect to its security types according to the following.

$$\frac{\begin{array}{c}\mathit{mtype}(m, C) = \bar T{\rightarrowtail}T \qquad \mathit{pars}(m, C) = \bar x \\ \mathsf{self}\colon (C, \kappa_0), \bar x : (\bar T, \bar\kappa), \mathsf{result}\colon (T, \kappa_4); (P{\cap}\mathit{Auth}\, C) \vdash S : (\mathit{com}\ L, \kappa_3); (Q{\cap}\mathit{Auth}\, C) \\ \text{for each } (\kappa_0, \bar\kappa{-}\langle P; \kappa_3; Q\rangle{\rightarrowtail}\kappa_4) \in \mathit{smtypes}(m, C)\end{array}}{C\ \textbf{\textit{extends}}\ D \vdash T\ m(\bar T\ \bar x)\{S\}}$$

This rule depends on rules for expressions and commands. The rules for commands are given in Table 5 but the rules for expressions are exactly the same as the ones in [2] and hence elided[2]. □

[2] A method can have more than one type so for flexibility in checking method declarations the rule must allow local variable declarations to be annotated differently for different types. The precise formulation [2] uses † but we omit the unilluminating complication here.

Table 5. Security typing rules for commands, for given $Auth$.

$$\frac{x \neq \mathsf{self} \qquad \Delta, x:(T,\kappa) \vdash e:(U,\kappa) \qquad U \leq T}{\Delta, x:(T,\kappa); P \vdash x := e : (com\ \kappa, H); P}$$

$$\frac{\Delta \vdash e_1:(C,\kappa_1) \qquad f:(T,\kappa) \in sfields\, C \qquad \Delta \vdash e_2:(U,\kappa) \qquad U \leq T \qquad \kappa_1 \leq \kappa}{\Delta; P \vdash e_1.f := e_2 : (com\ H, \kappa); P}$$

$$\frac{x \neq \mathsf{self} \qquad B \leq D}{\Delta, x:(D,\kappa); P \vdash x := \mathbf{new}\ B : (com\ \kappa, H); P}$$

$$\frac{\begin{array}{c}\Delta, x:(T,\kappa) \vdash e:(D,\kappa_0) \qquad mtype(m,D) = \bar{T} \to T' \qquad \Delta, x:(T,\kappa) \vdash \bar{e}:(\bar{U},\bar{\kappa}) \\ \bar{U} \leq \bar{T} \qquad x \neq \mathsf{self} \qquad T' \leq T \qquad \kappa_0', \bar{\kappa}' \text{–}\langle P'; \kappa_1'; Q' \rangle \!\rightarrow\! \kappa' \in smtypes(m,D) \\ \kappa_0', \bar{\kappa}' \text{–}\langle P'; \kappa_1'; Q' \rangle \!\rightarrow\! \kappa' \leq \kappa_0, \bar{\kappa} \text{–}\langle P'; \kappa_1; Q' \rangle \!\rightarrow\! \kappa \\ P' \cap Auth(\Delta^\dagger \mathsf{self}) \subseteq P \qquad Q \subseteq Q' \cap Auth(\Delta^\dagger \mathsf{self}) \\ \kappa_0 \leq \kappa \sqcap \kappa_1 \qquad \kappa = H \wedge \kappa_1 = H \;\Rightarrow\; (Auth(\Delta^\dagger \mathsf{self}) - P) \subseteq (\cap_{E \leq D} Auth\, E)\end{array}}{\Delta, x:(T,\kappa); P \vdash x := e.m(\bar{e}) : (com\ \kappa, \kappa_1); Q}$$

$$\frac{\Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2); Q_1 \qquad \Delta; Q_1 \vdash S_2 : (com\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash S_1;\ S_2 : (com\ \kappa_1, \kappa_2); Q}$$

$$\frac{\begin{array}{c}\Delta \vdash e : (\mathbf{bool}, \kappa) \\ \Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2); Q \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2); Q \qquad \kappa \leq \kappa_1 \sqcap \kappa_2\end{array}}{\Delta; P \vdash \mathbf{if}\ e\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2); Q}$$

$$\frac{\Delta \vdash e:(U,\kappa) \qquad U \leq T \qquad \Delta, x:(T,\kappa); P \vdash S : (com\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash (T,\kappa)\ x := e\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2); Q}$$

$$\frac{\Delta; (P - (P' \cap Auth(\Delta^\dagger \mathsf{self}))) \vdash S : (com\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash \mathbf{grant}\ P'\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2); Q \cup (P - (P' \cap Auth(\Delta^\dagger \mathsf{self})))}$$

$$\frac{\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash \mathbf{accept}\ P'\ \mathbf{in}\ S : (com\ \kappa_1, \kappa_2); Q - (P' \cap Auth(\Delta^\dagger \mathsf{self}))}$$

$$\frac{\begin{array}{c}P' \cap P = \varnothing \wedge P' \subseteq Auth(\Delta^\dagger \mathsf{self}) \\ \Delta; P \vdash S_1 : (com\ \kappa_1, \kappa_2); Q \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2); Q\end{array}}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2); Q}$$

$$\frac{P' \cap P \neq \varnothing \vee P' \not\subseteq Auth(\Delta^\dagger \mathsf{self}) \qquad \Delta; P \vdash S_2 : (com\ \kappa_1, \kappa_2); Q}{\Delta; P \vdash \mathbf{test}\ P'\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 : (com\ \kappa_1, \kappa_2); Q}$$

$$\frac{\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q \qquad \kappa_3 \leq \kappa_1 \qquad \kappa_4 \leq \kappa_2 \qquad P \subseteq P' \qquad Q' \subseteq Q}{\Delta; P' \vdash S : (com\ \kappa_3, \kappa_4); Q'}$$

In the rules for expressions and commands, we write Δ for typing environments that assign security types. A judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ says that if no permissions in set P are enabled initially, then S is safe, assigns only to variables (locals and parameters) of level $\geq \kappa_1$ and to object fields of level $\geq \kappa_2$ (see Lemma 4), and no permissions in set Q are enabled finally, i.e., after execution of S.

The rules use versions of the auxiliary functions that take security levels into account. Let $CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{\ \bar{\tau}_1\ \bar{f};\ \bar{M}\ \}$. Corresponding to *fields*, we define $\mathit{sfields}\, C = \bar{f} : \bar{\tau}_1 \cup \mathit{sfields}\, D$.

Judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ is derivable provided $P \subseteq Auth(\Delta^\dagger \mathsf{self})$ and $Q \subseteq Auth(\Delta^\dagger \mathsf{self})$ and the judgement is derivable using the security typing rules.

The last rule in Table 5 is a subsumption rule.

The rule for method call is different from our work on stack inspection [2] in that it has an extra condition for high commands: permissions that may be enabled at the site of the call (namely, $Auth(\Delta^\dagger \mathsf{self}) - P$) must be included in the static permissions of all classes that provide an implementation of the method (namely, $\cap_{E \leq D} Auth\, E$). This condition essentially disallows high commands from making calls that may dynamically dispatch to untrusted code. Because permission state is an implicit low variable, and a call to untrusted code causes a loss of permissions, this loss can be tested by a low observer and used to reveal secrets – recall the example attacks in section 2. Those attacks are untypable in our system because they require permission p at the call site for m to be included in the static permissions of m.

Note that the rule for method declaration does not restrict assignments to local variables, i.e., it allows effect L in the hypothesis. Subsumption may be used to get L there from H. The rule for method call has a form of subsumption built in: it requires there to be some declared type for the method that matches its invocation.

The second rule for **test** is the one that removes from consideration a branch that cannot be taken under the assumption: the test of P' fails if P' contains permissions assumed to be excluded or permissions that are not authorized for the class in which this command occurs. The first rule for **test** handles the case where it cannot be statically determined, from the information tracked in the judgements, whether the test of P' succeeds.

Properties of security typing. For any judgement $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$ derivable using the security typing rules for expressions and commands, the erased judgement $\Delta^\dagger \vdash S^\dagger$ is derivable using the ordinary typing rules for commands. Conversely, any program typable using the ordinary typing rules for commands can be annotated everywhere by L and typed by the security typing rules for expressions and commands, taking $\mathit{smtypes}(m, C) = \{L, \bar{L} \multimap \langle \varnothing; L; \varnothing \rangle \rightarrow L\}$ for all m, C. In other words, for the trivial security policy encoded by the above security type, the analysis rejects no well formed program.

6 Example

The following example, discussed in the introduction, shows our type system at work; in contrast to the scenario in section 3, where untrusted code calls trusted code, here we consider the dual scenario of trusted code calling untrusted code.

```
class NaiveProgram extends Object {//static permissions contain all permissions
  unit Main() {
    (string, L) s := BadPlugIn.TempFile();
    File.Delete(s); }
```

Suppose that NaiveProgram is a trusted class with all permissions. Next, we consider the partially trusted class BadPlugIn whose static permissions do not include *FileIO*.

```
class BadPlugIn extends Object {//static permissions do not contain FileIO
  (string, L) TempFile() { result := "...password file..."} }
```

The trusted class File has all permissions and contains the method Delete, where the file deletion operation is protected by a test of permission *FileIO*.

```
class File extends Object {//static permissions contain all permissions
  unit Delete((string, L) s) {
    test FileIO then Win32.Delete(s) else abort;} }
```

We decorate BadPlugin.TempFile() with the history-based flow policy

$$smtypes(\mathsf{TempFile}, \mathsf{BadPlugin}) = \{L, ()\text{–}\langle\varnothing; H; \{FileIO\}\rangle\!\!\rightarrow L\}$$

The code for File.Delete can be checked against the flow policy

$$\{L, L\text{–}\langle\{FileIO\};\ H\ ;\{FileIO\}\rangle\!\!\rightarrow(),\quad L, L\text{–}\langle\varnothing;\ H\ ;\varnothing\rangle\!\!\rightarrow()\}$$

The first is used in a context where *FileIO* is absent and asserts that *FileIO* is absent after the call is finished. Indeed, for this policy, the type system checks the body of Delete in the permission context $\{FileIO\} \cap Auth(File)$, i.e., the context $\{FileIO\}$. To type check the test, note that $\{FileIO\} \cap \{FileIO\} \neq \varnothing$, and Delete is accepted.

If we check NaiveProgram.Main for the policy $L, ()\text{–}\langle\varnothing; H; \{FileIO\}\rangle\!\!\rightarrow()$, we have the following situation: the call to TempFile results in the excluded permission set *FileIO*, which is the excluded permission set for the call to Delete. We have two possibilities for the type of Delete now, but only the type $L, L\text{–}\langle\{FileIO\}; H; \{FileIO\}\rangle\!\!\rightarrow()$ will do: from Table 5, rule for method call, we have to establish $\{FileIO\} \cap Auth(\mathsf{NaiveProgram}) \subseteq \{FileIO\}$ and $\{FileIO\} \subseteq \{FileIO\} \cap Auth(\mathsf{NaiveProgram})$. Both succeed. Hence NaiveProgram.Main is well-typed.

Note that we could not have chosen the type $L, L\text{–}\langle\varnothing; H; \varnothing\rangle\!\!\rightarrow()$ as the type of Delete, since we cannot establish $\{FileIO\} \subseteq \{FileIO\} \cap \varnothing$ for the postcondition.

In our previous work [2], on stack inspection, the typechecker rejected NaiveProgram.Main as ill-typed. It is instructive to recall the reason: after the call to TempFile, the call to Delete occurs in the context where *no* permissions are excluded, and then the antecedent $\{FileIO\} \cap Auth(\mathsf{NaiveProgram}) \subseteq \varnothing$ fails.

7 Indistinguishability and Confinement

In this section we show that if an expression is *safe*, i.e., accepted by the security typing rules of Section 5, and has level L, then it is *read confined*: its value does not depend on H-fields or H-variables. Moreover, if a command is safe and it has level $com\,H, H$ then it is *write confined*: it does not assign to L-fields or L-variables. These two properties are the semantic counterparts of the rules "no read up" and "no write down" that underly information flow control; the terms "simple security" and "*-property" are also used [5].

The formalization uses the indistinguishability relation $\sim$ which is also used to formulate noninterference in Section 8. States (h, η, P) and (h', η', P') may be indistinguishable to an L observer while having different allocation of objects visible only to H. For this reason, indistinguishability is formalized using a bijective correspondence between those locations in $dom\,h$ and $dom\,h'$ that, informally, are or have been visible to L.

Definition 4. *A* typed bijection *is a bijective finite partial function, σ, from Loc to Loc, such that $type(\sigma\,\ell) = type\,\ell$ for all ℓ in $dom\,\sigma$.* □

In the sequel, σ and its decorated variants range over typed bijections. We treat partial functions as sets of ordered pairs, so $\sigma' \supseteq \sigma$ expresses that σ' is an extension of σ.

Definition 5 (indistinguishable by L). *For any σ, we define relations $\sim_\sigma$ for data values, object states, heaps, and stores.*

$$
\begin{array}{lll}
\ell \sim_\sigma \ell' & in\ [\![C]\!] & \iff \sigma\,\ell = \ell' \vee \ell = nil = \ell' \\
d \sim_\sigma d' & in\ [\![T]\!] & \iff d = d' \quad for\ primitive\ types\ T \\
s \sim_\sigma s' & in\ [\![state\,C]\!] & \iff \forall (f : (T, \kappa)) \in sfields\,C \bullet \kappa = L \Rightarrow s\,f \sim_\sigma s'\,f \\
\eta \sim_\sigma \eta' & in\ [\![\Delta^\dagger]\!] & \iff \forall (x : (T, \kappa)) \in \Delta \bullet \kappa = L \Rightarrow \eta\,x \sim_\sigma \eta'\,x \\
h \sim_\sigma h' & in\ [\![Heap]\!] & \iff dom\,\sigma \subseteq dom\,h \ \wedge\ rng\,\sigma \subseteq dom\,h' \ \wedge \\
 & & \quad \forall \ell, \ell' \bullet \ell \sim_\sigma \ell' \Rightarrow h\,\ell \sim_\sigma h'\,\ell' \\
d \sim_\sigma d' & in\ [\![T_\perp]\!] & \iff d = \perp = d' \vee (d \neq \perp \neq d' \wedge d \sim_\sigma d'\ in\ [\![T]\!]) \\
P \sim_\sigma P' & in\ [\![\mathcal{P}(Perms)]\!] & \iff P = P'
\end{array}
$$

□

For classes C, the formulation above exploits the convention that equations involving partial functions are interpreted as false when the function is undefined. Thus, for $\ell \neq nil$, the relation $\ell \sim_\sigma \ell'$ holds only if ℓ is in $dom\,\sigma$. The last clause, for $T_\perp$, is needed to handle errors (null dereferences) in expressions.

In our model, permissions are atomic values and indistinguishability is just equality for permission sets. In a more detailed model, permissions would be heap objects.

Indistinguishability is not symmetric or reflexive in general. But $h \sim_\iota h$ where ι is the identity on $dom\,h$. Limited transitivity and symmetry hold, e.g., if $h_1 \sim_\sigma h_2$ and $h_2 \sim_\tau h_3$ then $h_1 \sim_{\tau \circ \sigma} h_3$.

One use of $\sim$ is to formulate, in Lemma 4 below, that if a command is typable as $(com\ H, \kappa)$ it does not assign to L-variables, and if it is typable as $(com\ \kappa, H)$ it does not assign to L-fields of objects. For this purpose we use $h \sim_\iota h_0$, for initial h and final h_0, where ι is the identity on $dom\, h$. This expresses that no L fields of initially existing objects are changed.

Each of our results about the meaning of a class table CT is proved by induction on the approximation chain by which $[\![CT]\!]$ is defined. The induction step is treated as a separate lemma about commands, in which the induction hypothesis is an assumption about the method environment.

The security typing rules depend on permission effects. Thus, we first show some results on the soundness of effects. The intuition is that if a safe command is executed with permissions disjoint from the initially excluded permissions, then the permissions produced as a result of execution are disjoint from the final set of excluded permissions. Formalizing this intuition is the purpose of Definition 6, Lemma 2 and Lemma 3 below.

For brevity we write $Q \# P$ for $Q \cap P = \varnothing$.

Definition 6 (disjoint effects in method environment). *Method environment μ has* disjoint effects*, written disj μ, if the following holds for all C, m and all $\kappa_0, \bar{\kappa}\text{–}\langle P; \kappa; Q\rangle\text{→}\kappa_1$ in smtypes(m, C). If $R \# P$ and $\mu Cm(h, \eta, R) \neq \bot$ then $Q \# Q_0$, where $(h_0, d, Q_0) = \mu Cm(h, \eta, R)$.* □

Lemma 2 (disjoint effects in commands). *Suppose $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2)$; Q and disj μ. For all η, h, R such that $P \# R$, and $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, if $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ then $Q \# Q_0$.*

Lemma 3 (safe programs have disjoint effects). *If annotated class table CT is safe then disj $[\![CT^\dagger]\!]$ and also disj μ_i for each μ_i in the approximation chain for semantics of CT.*

The purpose of Definition 7, Lemma 4 and Lemma 5 below is to establish that H-commands do not assign to L-variables and L-fields of objects.

Definition 7 (write confined method environment). *Method environment μ is* write confined*, written wconf μ, if the following holds for all C, m and all $\kappa, \bar{\kappa}\text{–}\langle P; H; Q\rangle\text{→}\kappa_1$ in smtypes(m, C). If $R \# P$ and $\mu Cm(h, \eta, R) \neq \bot$ then $h \sim_\iota h_0$ where $(h_0, d, Q_0) = \mu Cm(h, \eta, R)$ and ι is the identity on dom h. Moreover, if $\kappa_1 = H$, then $R' = Q_0$ where $R' = R \cap Auth\, C$.* □

Lemma 4 (write confinement of commands). *Suppose disj μ, wconf μ, and $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$. For all η, h, R such that $P \# R$, and $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, if $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ then: (i) $\kappa_1 = H$ implies $\eta \sim_\iota \eta_0$; (ii) $\kappa_2 = H$ implies $h \sim_\iota h_0$ and (iii) $\kappa_1 = H$ and $\kappa_2 = H$ imply $R = Q_0$, where ι is the identity on dom h.*

Note that no condition is imposed if $[\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R) = \bot$.

Lemma 5 (safe programs are write confined). *If annotated class table CT is safe then wconf $[\![CT^\dagger]\!]$ and also wconf μ_i for each μ_i in the approximation chain for semantics of CT.*

The last result in this section can be seen as a simple form of noninterference. It says that if an expression can be typed $\Delta \vdash e : (T, L)$ then its meaning is the same in two L-indistinguishable states.

Lemma 6 (safe expressions are read confined).
Suppose $\Delta \vdash e : (T, L)$ *and* $h \sim_\sigma h'$ *and* $\eta \sim_\sigma \eta'$. *If* $d = [\![\Delta^\dagger \vdash e : T]\!](h, \eta)$ *and* $d' = [\![\Delta^\dagger \vdash e : T]\!](h', \eta')$ *then* $d \sim_\sigma d'$.

8 Safety Implies Noninterference

This section states the main result: if a class table is accepted by the security typing rules then the method environment that it denotes satisfies noninterference. That is, if it is safe with respect to a given flow policy then its semantics for the given access policy does satisfy the flow policy.

Noninterference for a class table is defined in terms of noninterference of method meanings with respect to their security types. Roughly, the idea is that a method executed under related stores, related heaps, and related permission states, yields related heaps. Provided *smtypes* of the method declares that the level of the return result is L, the return results are also related. This idea can be formalized (as in [2]) by saying that a method meaning d *satisfies* a method type $\kappa_0, \bar{\kappa} \text{–} \langle P; \kappa_1; Q \rangle \rightarrow \kappa_2$ iff the following holds for all $\sigma, h, h', \eta, \eta', P_1, P_1'$: Let $(h_0, d_0, Q_0) = d(h, \eta, P_1)$, and $(h_0', d_0', Q_0') = d(h', \eta', P_1)$. If $h \sim_\sigma h'$, $\eta \sim_\sigma \eta'$, $P_1 \sim_\sigma P_1'$, and $P_1 \# P$ then there is $\tau \supseteq \sigma$ such that $h_0 \sim_\tau h_0'$, $Q \# Q_0$, $Q_0 \sim_\tau Q_0'$, and $(\kappa_2 = L \Rightarrow d_0 \sim_\tau d_0')$

Definition 8 (noninterfering method environment). *A method environment is noninterfering, written nonint* μ, *iff for all* C, m, *the meaning* μCm *satisfies every* $\kappa_0, \bar{\kappa} \text{–} \langle P; \kappa_1; Q \rangle \rightarrow \kappa_2$ *in* $smtypes(m, C)$. □

Our main result is that the method environment denoted by a safe class table is noninterfering. The proof uses lemmas which express noninterference for the expression and command constructs, respectively.

The proof of the main theorem goes by proving noninterference of each method environment in the approximation chain, using the following.

Lemma 7 (safe commands are noninterfering). *Suppose disj* μ, *wconf* μ, *nonint* μ, *and* $\Delta; P \vdash S : (com\ \kappa_1, \kappa_2); Q$. *Suppose also* $R \# P$, $R \subseteq Auth(\Delta^\dagger\ \mathsf{self})$, $\eta \sim_\sigma \eta'$, $h \sim_\sigma h'$, $R \sim_\sigma R'$, $(h_0, \eta_0, Q_0) = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h, \eta, R)$ *and* $(h_0', \eta_0', Q_0') = [\![\Delta^\dagger \vdash S^\dagger]\!]\mu(h', \eta', R')$. *Then there is* $\tau \supseteq \sigma$ *such that* $\eta_0 \sim_\tau \eta_0'$ *and* $h_0 \sim_\tau h_0'$ *and* $Q_0 \sim_\tau Q_0'$.

Theorem 1 (safety implies noninterference). *If annotated class table* CT *is safe then its meaning* $[\![CT^\dagger]\!]$ *is noninterfering.*

9 Discussion

We have formalized the history based mechanism of Abadi and Fournet and shown how, by tracking updates of the permission state, static rules can ensure

noninterference in programs that depend on access control to prevent illegal information flow. Unlike stack inspection, the mechanism itself introduces a new channel of information flow, but one that can be controlled using the same sort of type-and-effect analysis that we previously developed for stack inspection [2]. For modular (per-method) checking of dynamically dispatched method calls, we rely on a flow policy that specifies a set of types for every method; this flow policy is invariant with respect to subclassing. Whereas these flow types describe flows that occur in the absence of certain permissions, a fully modular system would also specify certain permissions required to be present. This set would be used to check method calls in H commands. In this paper we chose to omit the latter form of interface specification, but the call rule imposes essentially the same condition.

Specifying the analysis in terms of a type system is convenient for proving our noninterference result. But for practical application of the result, type inference is needed to reduce the burden of annotation. We have developed a modular inference algorithm for inferring security types for an object-oriented language without dynamic access control [17]. In future work, we plan to report on security type inference for a language with dynamic access control supporting, e.g., the stack inspection mechanism or the history-based mechanism.

Another practical issue is to provide a syntax for flow policy. By Lemma 1, permission state after execution is at most the permission state before execution. So it is reasonable to expect that in the analysis, the final set of excluded permissions contain the initial set of excluded permissions. Thus in a flow policy, $\kappa_0, \bar{\kappa} \text{–} \langle P; \kappa; Q \rangle \rightarrow \kappa_1$, Q can contain the permissions not already in P. To translate this into the framework of this paper, Q can be replaced by $P \cup Q$.

References

1. M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, Feb. 2003.
2. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming, Special Issue on Language-based Security*. To appear.
3. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–270. IEEE Computer Society Press, 2002.
4. A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 155–169. IEEE Computer Society Press, 2003.
5. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.
6. D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
7. C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Prog. Lang. Syst.*, 25(3):360–399, 2003.

8. J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
9. L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
10. J. Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2001.
11. N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 365–377, 1998.
12. A. Igarashi and N. Kobayashi. Resource Usage Analysis. *ACM Trans. Prog. Lang. Syst.*, 2004. To appear.
13. Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the the Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, Aug. 2003.
14. K. Marriott, P. J. Stuckey, and M. Sulzmann. Resource usage verification. In *Proceedings of the First Asian Programming Languages Symposium (APLAS)*, 2003.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.
16. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
17. Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.
18. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

The Spec# Programming System: An Overview

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte

Microsoft Research, Redmond, WA, USA
{mbarnett,leino,schulte}@microsoft.com

Abstract. The Spec# programming system is a new attempt at a more cost effective way to develop and maintain high-quality software. This paper describes the goals and architecture of the Spec# programming system, consisting of the object-oriented Spec# programming language, the Spec# compiler, and the Boogie static program verifier. The language includes constructs for writing specifications that capture programmer intentions about how methods and data are to be used, the compiler emits run-time checks to enforce these specifications, and the verifier can check the consistency between a program and its specifications.

1 Introduction

Software engineering involves the construction of correct and maintainable software. Techniques for reasoning about program correctness have strong roots in the late 1960's (most prominently, Floyd [26] and Hoare [34]). In the subsequent dozen years, several systems were developed to offer mechanical assistance in proving programs correct (see, *e.g.*, [38, 28, 52]). To best influence the process by which a software engineer works, one can aim to enhance the engineer's primary thinking and working tool: the programming language. Indeed, a number of programming languages have been designed especially with correctness in mind, via specification and verification, as in, for example, the pioneering languages Gypsy [2] and Euclid [39]. Other languages, perhaps most well-known among them Eiffel [55], turn embedded specifications into run-time checks, thereby dynamically checking the correctness of each program run.

Despite these visionary underpinnings and numerous victories over technical challenges, current software development practices remain costly and error prone (*cf.* [57, 53]). The most common forms of specification are informal, natural-language documentation, and standardized library interface descriptions (of relevance to this paper, the .NET Framework, see, *e.g.*, [62]). However, numerous programmer assumptions are left unspecified, which complicates program maintenance because the implicit assumptions are easily broken. Furthermore, there's generally no support for making sure that the program works under the assumptions the programmer has in mind and that the programmer has not accidentally overlooked some assumptions. We think program development would be improved if more assumptions were recorded and enforced. Realistically, this will not happen unless writing down such specifications is easy and provides not just long-term benefits but also immediate benefits.

The Spec# programming system is a new attempt at a more cost effective way to produce high-quality software. For a programming system to be adopted widely, it must provide a complete infrastructure, including libraries, tools, design support,

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 49–69, 2005.

integrated editing capabilities, and most importantly be easily usable by many programmers. Therefore, our approach is to integrate into an existing industrial-strength platform, the .NET Framework. The Spec# programming system rests on the Spec# programming language, which is an extension of the existing object-oriented .NET programming language C#. The extensions over C# consist of specification constructs like pre- and postconditions, non-null types, and some facilities for higher-level data abstractions. In addition, we enrich C# programming constructs whenever doing so supports the Spec# programming methodology. We allow interoperability with existing .NET code and libraries, but we guarantee soundness only as long as the source comes from Spec#. The specifications also become part of program execution, where they are checked dynamically. The Spec# programming system consists not only of a language and compiler, but also an automatic program verifier, called Boogie, which checks specifications statically. The Spec# system is fully integrated into the Microsoft Visual Studio environment.

The main contributions of the Spec# programming system are

- a small extension to an already popular language,
- a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks,
- tools that enforce the methodology, ranging from easily usable dynamic checking to high-assurance automatic static verification, and
- a smooth adoption path whereby programmers can gradually start taking advantage of the benefits of specification.

In this paper, we give an overview of the Spec# programming system, its design, and the rationale behind its design. The system is currently under development.

2 The Language

The Spec# language is a superset of C#, an object-oriented language targeted for the .NET Platform. C# features single inheritance whose classes can implement multiple interfaces, object references, dynamically dispatched methods, and exceptions, to mention the features most relevant to this paper. Spec# adds to C# type support for distinguishing non-null object references from possibly-null object references, method specifications like pre- and postconditions, a discipline for managing exceptions, and support for constraining the data fields of objects. In this section, we explain these features and rationalize their design.

2.1 Non-null Types

Many errors in modern programs manifest themselves as null-dereference errors, suggesting the importance of a programming language providing the ability to discriminate between expressions that may evaluate to null and those that are sure not to (for some experimental evidence, see [25, 23]). In fact, we would like to eradicate all null-dereference errors.

We have opted to add type support for nullity discrimination to Spec#, because we think types offer the easiest way for programmers to take advantage of nullity distinctions. Backward compatibility with C# dictates that a C# reference type T denote a

possibly-null type in Spec# and that the corresponding non-null type get a new syntax, which in Spec# we have chosen to be $T!$.

The main complication in a non-null type system arises in addressing non-null fields of partially constructed objects, as illustrated in the following example:

```
class Student : Person {
  Transcript! t;
  public Student(string name, EnrollmentInfo! ei)
    : base(name) {
    t = new Transcript(ei);
  }
```

Since the field t is declared of a non-null type, the constructor needs to assign a non-null value to t. However, note that in this example, the assignment to t occurs after the call to the base-class constructor (as it must in C#). For the duration of that call, t is still null, yet the field is already accessible (for instance, if the base-class constructor makes a dynamically dispatched method call). This violates the type safety guarantees of the non-null type system.

In Spec#, this problem is solved syntactically by allowing constructors to give initializers to fields before the object being constructed becomes reachable by the program. To correct the example above, one writes:

```
class Student : Person {
  Transcript! t;
  public Student(string name, EnrollmentInfo! ei)
    : t(new Transcript(ei)),
      base(name) {
  }
```

Spec# borrows this field-initialization syntax from C++, but a crucial point is that Spec#, unlike C++, evaluates field initializers before calling the base-class constructor. Note that such an initializing expression can use the constructor's parameters, a useful feature that we deem vital to any non-null type design. Spec# requires initializers for every non-null field.

Spec# allows non-null types to be used only to specify that fields, local variables, formal parameters, and return types are non-null. Array element types cannot be non-null types, avoiding both problems with array element initialization and problems with C#'s covariant arrays.

To make the use of non-null types even more palatable for migrating C# programmers, Spec# stipulates the inference of non-nullity for local variables. This inference is performed as a dataflow analysis by the Spec# compiler.

We have settled on this simple non-null type system for three reasons. First, problems with null references are endemic in object-oriented programming; providing a solution should be very attractive to a large number of programmers. Second, our simple solution covers a majority of useful non-null programming patterns. Third, for conditions that go beyond the expressiveness of the non-null type system, programmers can use method and class contracts, as described below.

2.2 Method Contracts

Every method (including constructors and the properties and indexers of C#) can have a specification that describes its use, outlining a contract between callers and implementations. As part of that specification, *preconditions* describe the states in which the method is allowed to be called, and hence are the caller's responsibility. *Postconditions* describe the states in which the method is allowed to return. The *throws set* and its associated *exceptional postconditions* limit which exceptions can be thrown by the method and for each such exception, describe the resulting state. Finally, *frame conditions* limit the parts of the program state that the method is allowed to modify. The postconditions, throws set, exceptional postconditions, and frame conditions are the implementation's responsibility. Method contracts establish responsibilities, from which one can assign blame in case of a contract-violation error.

Uniform error handling in modern programming languages is often provided by an *exception* mechanism. Because the exception mechanisms in C# and the .NET Framework are rather unconstrained, Spec# adds support for a more disciplined use of exceptions to improve the understandability and maintenance of programs. As a prelude to explaining method contracts, we describe the Spec# view of exceptions.

Exceptions. Spec# categorizes exceptions according to the conditions they signal. Looking at exceptions as pertaining to particular methods, Goodenough [29] categorizes exceptions into two kinds of failures, which we call *client failures* and *provider failures*. A client failure occurs when a method is invoked under an illegal condition, that is, when the method's precondition is not satisfied. We further refine provider failures into *admissible failures* and *observed program errors*. An admissible failure occurs when a method is not able to complete its intended operation, either at all (*e.g.*, the parity of a received network packet is wrong) or after some amount of effort (*e.g.*, after waiting on input from a network socket for some amount of time). The set of admissible failures is part of the contract between callers and implementations. An observed program error is either an intrinsic error in the program (*e.g.*, an array bounds error) or a global failure that's not particularly tied with the method (*e.g.*, an out-of-memory error).

An important consideration among these kinds of exceptions is whether or not one expects a program ever to catch the exception. Admissible failures are part of a program's intended possible behaviors, so we expect correct programs to catch and handle admissible failures. In contrast, correct programs never exhibit client failures or observed program errors, and it's not even clear how a program is to react to such errors. If the program handles such failures at all, it would be at the outermost tier of the application or thread.

Because of these considerations, Spec# follows Java [30] by letting programmers declare classes of exceptions as either *checked* or *unchecked*. Admissible failures are signaled with checked exceptions, whereas client failures and observed program errors are signaled using unchecked exceptions.

In Spec#, any exception class that implements the interface $ICheckedException$ is considered a checked exception. For more information about the exception design in Spec#, see our companion paper on exception safety [49].

ArrayList.Insert Method (*Int32*, *Object*)

Inserts an element into the *ArrayList* at the specified index.

public virtual void *Insert*(**int** *index*, **object** *value*);

Parameters

- *index* The zero-based index at which *value* should be inserted.
- *value* The *Object* to insert. The *value* can be a null reference.

Exceptions

Exception Type	Condition
ArgumentOutOfRangeException	*index* is less than zero. *–or–* *index* is greater than *Count*.
NotSupportedException	The *ArrayList* is read-only. *–or–* The *ArrayList* has a fixed size.

Fig. 1. The .NET Framework documentation for the method *ArrayList.Insert*.

Preconditions. Perhaps the most important programmer assumption is the precondition. Here is a simple example of a method with a precondition:

```
class ArrayList {
  public virtual void Insert(int index, object value)
    requires 0 <= index && index <= Count;
    requires !IsReadOnly && !IsFixedSize;
  { ... }
```

The precondition specifies that the index into which the object is to be inserted in the array list must be within bounds, and that the list can grow. To enforce these preconditions, the Spec# compiler emits run-time checks that throw a *RequiresViolationException*, indicating a client failure, if a precondition is not met. If the user invokes Boogie on a call site, then Boogie attempts to verify statically that these preconditions hold at the call site, reporting an error if it cannot.

The .NET Framework documentation for this method is shown in Figure 1. There is a subtle difference between the .NET documentation for *Insert* and our specification of it above. Both specifications state what's expected of the caller; the difference lies in the action taken in the event that preconditions are violated. To support this typical robust-programming style of .NET Framework specifications, Spec#'s preconditions can have **otherwise** clauses. These can be used to tell the compiler to use a specified exception, rather than the default *RequiresViolationException*, in the event that a precondition violation is detected at run time:

```
class ArrayList {
  void Insert(int index, object value)
    requires 0 <= index && index <= Count
      otherwise ArgumentOutOfRangeException;
    requires !IsReadOnly && !IsFixedSize
      otherwise NotSupportedException;
  { ... }
```

Since it represents a client failure, the exception used in an **otherwise** clause must be an unchecked exception.

Postconditions. Method specifications can also include postconditions. For example, one can specify the postconditions of *Insert* as follows:

```
ensures Count == old(Count) + 1;
ensures value == this[index];
ensures Forall{int i in 0 : index;  old(this[i]) == this[i]};
ensures Forall{int i in index : old(Count);  old(this[i]) == this[i + 1]};
```

These postconditions say that the effect of *Insert* is to increase *Count* by 1, to insert the given value at the given index, and to keep all other elements in their same relative positions. This example also shows some other Spec# specification features: In the first line, **old**(*Count*) denotes the value of *Count* on entry to the method. In the third line, the special function *Forall* is applied to the comprehension of the boolean expression **old**(**this**[*i*]) == **this**[*i*], where *i* ranges over the integer values in the half-open interval from 0 to less than *index*. Comprehensions and quantifiers are syntactically restricted in such a way that the compiler can always generate code that computes them.

Boogie attempts to verify each implementation of *Insert* against these postconditions. When Boogie's verification is successful, then the run-time checks (which would throw an *EnsuresViolationException* in this case) are not needed since they would never fail.

For run-time checking, we have adopted Eiffel's mechanism for evaluating **old**(*E*). On entry to a method, the expression *E* of any **old**(*E*) occurring in a postcondition is evaluated and the resulting value is saved away. Then, whenever (and if) this value of **old**(*E*) is needed during the evaluation of the postcondition, the saved value of *E* is used. Note that the value of **old**(*E*) may in fact not be needed during the evaluation of the postcondition due to short-circuit boolean expressions or because the method does not terminate normally.

The example above also illustrates a more general point about the differences between checking contracts statically and dynamically. Boogie has knowledge about the program and its built-in data structures. It also has support for quantifiers and can therefore check the postconditions of *Insert* statically. Contracts that use procedural abstraction, however, can be a problem for static modular checking, since such checking has access only to a limited part of the program. Likewise, contracts that use higher-level data structures can be a problem for static checking, because of limitations of decisions procedures and axiomatizations of some theories. Here, dynamic checking is straightforward. On the other hand, the dynamic checking of postconditions can be quite involved when **old** expressions mention quantified variables, as exemplified above.

Though we expect the bulk of specifications to be simple, the more general point is that Spec# supports expressive specifications even when those specifications push the limits of today's checking technology.

Exceptional postconditions. As in Java, each method whose invocation may result in a checked exception must account for that exception in the method's throws set. For example, the declaration

```
char Read()
  throws SocketClosedException;
{ ... }
```

where *SocketClosedException* is a checked exception class, allows the method to throw any checked exception whose allocated type is a subclass of *SocketClosedException*, but is not allowed to throw any other checked exception. The Spec# compiler holds every implementation to its throws set by a conservative control-flow analysis. A **throws** clause in Spec# can only mention checked exceptions.

Spec# allows a **throws** declaration to be combined with a postcondition that takes effect in the event that the exception is thrown. For example, the exceptional postcondition in

```
void ReadToken(ArrayList a)
  throws EndOfFileException ensures a.Count == old(a.Count);
{ ... }
```

says that the length of a is unchanged in the event that the method results in an *EndOfFileException*.

Without further restrictions, it would be possible for a program to foil the compiler's throws-set analysis, which would then undermine Spec#'s guarantee that every checked exception is accounted for. Consider the following example:

```
void ExceptionScam() {
  Exception e = new MyCheckedException();
  throw e;
}
```

The root of the exception class hierarchy, *Exception*, is an unchecked exception (because it comes from C#, where all exceptions are unchecked). Since checked exceptions are subtypes of *Exception*, the **throw** statement in *ExceptionScam* would have the effect of throwing a checked exception even though the method does not advertise it. Spec# prevents this: whenever the static type of a thrown expression is an unchecked exception and the static analysis cannot guarantee that the dynamic type is likewise unchecked, then the compiler inserts a run-time check that detects any violation of Spec#'s distinction between checked and unchecked exceptions.

For more information about exceptions in Spec#, see our companion paper on exception safety [49].

Frame conditions. Spec# method contracts also include **modifies** clauses (also known as *frame conditions*), which restrict which pieces of the program state a method implementation is allowed to modify. For example, in the class

```
class C {
  int x, y;
  void M() modifies x; { ... }
```

method M is permitted to have a net effect on the value of x, whereas the value of y on exit from the method must have the same value as on entry.

Any realistic design of **modifies** clauses includes some facility for abstracting over program state that for reasons of information hiding cannot be mentioned in the method contract. For example, the implementation of $ArrayList.Insert$ is going to modify the private representation of the $ArrayList$, but private variables are not allowed to be mentioned explicitly in the contract of a public method. Instead, a wildcard can be used. For example, the specification

```
modifies this^ArrayList;
```

allows any field of **this** declared in class $ArrayList$ to be modified. Spec# also supports other flavors of wildcards (see [4]), which additionally address the problem of specifying the modification of state in subclasses (*cf.* [42]).

But wildcards are still just a partial solution to the frame problem, because they don't extend to aggregate objects. For example, the $ArrayList$ implementation consists of an array and a count. The **modifies** clause above allows the count and the reference to the array to be changed, but does not give explicit permission to modify the array elements. To deal with aggregate objects, Spec# uses a concept of *ownership*. We say that the $ArrayList$ owns its underlying array, that the array is *committed* to the $ArrayList$. Modifications to the state of committed objects do not need to be mentioned explicitly in the **modifies** clause. For more details, see [4], which also describes the connection between ownership and object invariants.

Frame conditions serve as documentation and are used and enforced by Boogie, but they are currently not enforced at run time. There are two reasons for not checking **modifies** clauses at run time. First, they can be prohibitively expensive, since the checking must compare arbitrarily large portions of the heap in a method's pre-state and post-state. Second, we are aiming for a smooth transition to Spec# from C#; we do not want to incur run-time errors in C# programs that otherwise are correct.

Inheritance of specifications. In Spec#, a method's contract is inherited by the method's overrides. The run-time checks evoked by the method contract are thus also inherited. Not only does this make the specifications more definitive and reliable than today's documentation, but the Spec# specifications also make the code of an implementation easier to read, since today's manually written code for checking preconditions can be rather lengthy.

A method override can add more postconditions by declaring additional **ensures** clauses. The override can add exceptional postconditions only for those exceptions that are already covered by the throws set. The override is not allowed to give any **modifies** clause: enlarging the frame would be unsound, and shrinking the frame can be done with

an added postcondition. Spec# does not allow any changes in the precondition, because callers expect the specification at the static resolution of the method to agree with the dynamic checking.

Methods declared in an interface can have specifications, just like the methods declared in a class. Interfaces give rise to a form of multiple inheritance, because a class can inherit a method signature from the superclass and its implemented interfaces. Traditionally, these inherited specifications are combined [63], which is what Spec# does for postconditions. Spec# also combines exceptional postconditions, but the inherited specifications must have identical throws sets. If a class implements an interface method, then the interface declaration of the method must have a frame condition that is a superset of the class implementation of the method. Spec# does not combine preconditions, unless they are the same, for the reason explained above. Since the obvious definitions of "the same" are either syntactic and brittle, or semantic and require theorem proving, Spec# uses the radical solution of allowing multiple inherited specifications only when these have no **requires** clauses.

We give an example that shows Spec#'s radical precondition solution not to be too draconian. Consider the following interfaces:

$$\textbf{interface}\ I\ \{\ \textbf{void}\ M(\textbf{int}\ x)\ \textbf{requires}\ x <= 10;\ \}$$
$$\textbf{interface}\ J\ \{\ \textbf{void}\ M(\textbf{int}\ x)\ \textbf{requires}\ x >= 10;\ \}$$

Suppose a class C wants to implement both interfaces I and J. In this case, Spec# does not allow C to provide one shared implementation for $I.M$ and $J.M$. Instead, class C needs to give explicit interface method implementations for M:

$$\textbf{class}\ C : I, J\ \{$$
$$\quad \textbf{void}\ I.M(\textbf{int}\ x)\ \{\ \ldots\ \}$$
$$\quad \textbf{void}\ J.M(\textbf{int}\ x)\ \{\ \ldots\ \}$$

(Explicit interface method implementations are a feature of C#.) Because an explicit interface method implementation cannot be accessed other than through the interface, it gets its contract straight from the interface.

Taken together, the Spec# rules for contract inheritance guarantee that a derived specification always properly obeys the behavioral subtyping rules [22, 24].

2.3 Class Contracts

Specifying the rules for using a library or abstraction is done primarily through method contracts, which spell out what's expected of the caller and what the caller can expect in return from the implementation. To specify the design of an implementation, one primarily uses specifications that constrain the value space of the implementation's data. These specifications are called *object invariants* and spell out what is expected to hold of each object's data fields in the steady state of the object. For example, the class fragment

$$\textbf{class}\ \mathit{AttendanceRecord}\ \{$$
$$\quad \mathit{Student}[\,]!\ \mathit{students};$$
$$\quad \textbf{bool}[\,]!\ \mathit{absent};$$
$$\quad \textbf{invariant}\ \mathit{students.Length} == \mathit{absent.Length};$$

declares that the lengths of the arrays $students$ and $absent$ are to be the same.

As we can see from the simple example above, it is not possible for an object invariant always to hold, because it is not possible in the language to change the lengths of two arrays simultaneously. This is why we say the object invariant holds in *steady states*, which essentially means that the object is not currently being operated on. Following our methodology for object invariants [4, 46, 7], Spec# makes explicit when an object is in its steady state versus when it is *exposed*, which means the object is vulnerable to modifications. Spec# introduces a block statement **expose** that explicitly indicates when an object's invariant may temporarily be broken: the statement

```
expose (o) {
  S;
}
```

exposes the object o for the duration of the sub-statement S, which may then operate on the fields of o. Because field modifications in an object-oriented program tend to be encapsulated in the class that declares the field, the expression o is usually **this**. The object invariant is supposed to hold again at the end of the **expose** statement and Spec# enforces this with a run-time check. Object invariants are also checked at the end of constructors (though there's some flexibility that allows the initial check of an object invariant to be performed elsewhere; we omit the details here).

By default, whenever a class or any of its superclasses has a declared invariant, every public method of the class has an implicit

```
expose (this) { ... }
```

around the method body. Our preliminary experience suggests that this default removes most of the need for explicit **expose** statements. In situations where reentrancy is desired, the default can be disabled by a custom attribute on the method.

Exposing an object is not idempotent. That is, it is a checked run-time error if **expose** (o) ... is reached when o is already exposed. In this way, the expose mechanism is similar to thread-non-reentrant mutexes in concurrent programming, where monitor invariants [35] are the analog of our object invariants. If exposing were idempotent, then one would not be able to rely on the object invariant to hold immediately inside an expose block, in the same way that the idempotence of thread-reentrant mutexes means that one cannot rely on the monitor invariant to hold at the time the mutex is acquired.

For Spec#'s object-invariant methodology to be sound, all modifications of a field $o.f$ must take place while the object o is exposed. Furthermore, the methodology uses an ownership relation to structure objects into a tree-shaped hierarchy. The relation is state dependent, which allows ownership transfer. Such modifications and ownerships are enforced by Boogie, but are not enforced at run time.

Object invariants can be declared in any class. To support modular checking of invariants, so that a class does not need to know the invariants of its superclasses and future subclasses, object invariants are partitioned into *class frames* according to the class that declares each invariant [4, 18]. The **expose** mechanism deals with class frames.

To reduce the programmer's initial cost of adding **expose** statements and to handle non-virtual methods in a more backward compatible way (see [4]), Spec# allows one

expose statement to expose more than one class frame. To explain this feature, we first need to show the more general form of the **expose** statement in Spec#, which is

$$\textbf{expose}\,(o\ \textbf{upto}\ T)\ \{\ \ldots\ \}$$

where T is a superclass of the static type of the expression o. If "**upto** T" is omitted, T defaults to the static type of expression o. More precisely than we described it above, the statement exposes all of o's class frames from above its currently exposed class frame through T (also exposing the class frame T itself). Non-idempotence requires that at least one class frame is exposed as part of the operation. At the end of the **expose** block, the class frames that were exposed on entry are un-exposed, and the object invariant for each of those class frames is checked. This is done at run time using compiler-emitted dynamically dispatched methods that check the invariants.

Exposing an unknown number of class frames, and in particular checking the invariants for class frames whose declarations may not be in scope, poses a problem for modular, static verification. Therefore, we use a stricter model for **expose** in Boogie. In particular, whereas the precondition for

$$\textbf{expose}\,(o\ \textbf{upto}\ T)\ \{\ \ldots\ \}$$

as enforced by run-time checks is that o's T class frame is un-exposed—that is, that the o's most-derived un-exposed class frame is a subclass of T—Boogie strengthens this precondition by requiring o's most-derived un-exposed class frame to be exactly T. This way, Boogie is able to find all the object invariants that it needs to check at the end of the **expose** block. In effect, this difference in policy between the run-time behavior and what's enforced by Boogie means that programmers can start writing and running Spec# programs more easily, but then may need to exert additional effort in order to obtain the higher confidence in the program's correctness assured by Boogie (just as additional effort is required to make sure Boogie's modification and ownership rules are satisfied).

Object invariants are allowed to mention only constants, fields, array elements, state independent methods, and confined methods. A method is state independent if it does not depend on mutable state. A confined method may depend on the state of owned objects. The Spec# compiler includes a conservative effect analysis to check that these properties are obeyed.

Spec# also supports class invariants, which are useful to document assumptions about static fields. Methodology and constraints for class invariants are similar to those for object invariants, except that there is no inheritance [45]. The **expose** statement simply takes a class instead of an object as a parameter.

2.4 Other Details

Exceptions within contracts. If an exception is thrown during the evaluation of a contract in Spec#, then the exception is wrapped in a contract evaluation exception and propagated. This is in contrast to the run-time evaluation of contracts in JML, where such exceptions are caught and the surrounding formula is treated as if it returned a boolean value according to certain rules, see [15].

Custom attributes on specifications. C# provides *custom attributes* as a way to attach arbitrary data to program structures, such as classes, methods, and fields. A custom attribute is compiled into metadata whose standard format allows various applications to read the custom attributes attached to a particular declaration. Spec# also allows each specification clause to be annotated with custom attributes.

Custom attributes allow users of third-party tools to mark up specifications in tool-specific ways. For instance, the Spec# compiler uses the $Conditional$ custom attribute to control which specifications are emitted as run-time checks in the current build. For example, for the following method

$$\begin{array}{l}\mathbf{int}\ \mathit{BinarySearch}(\mathbf{object}[]!\ a, \mathbf{object}\ o, \mathbf{int}\ \mathit{lo}, \mathbf{int}\ \mathit{hi})\\ \quad \mathbf{requires}\ 0 <= \mathit{lo}\ \&\&\ \mathit{lo} <= \mathit{hi}\ \&\&\ \mathit{hi} <= a.\mathit{Length};\\ \quad [\mathit{Conditional}(\text{“}\mathit{DEBUG}\text{”})]\ \mathbf{requires}\ \mathit{IsSorted}(a, \mathit{lo}, \mathit{hi});\\ \{\ \ldots\ \}\end{array}$$

the compiler emits run-time checks for both preconditions in the debug build, but emits a check only for the first precondition in the non-debug build. This supports the common programming style of debugging assertions (see, *e.g.*, [54]).

Purity. We want to have the property that a program that runs correctly with all contract checking enabled also runs correctly if some of the contract checking is disabled. Therefore, we require all expressions appearing in contracts to be *pure*, meaning that they have no side effects and do not throw any checked exceptions. The compiler enforces this condition using a conservative effect system. We are considering more liberal definitions of purity, such as observational purity [8] and that afforded by the heap analysis of Sălcianu and Rinard [59].

3 System Architecture

Architecturally, the Spec# programming system consists of the compiler, a runtime library, and the Boogie verifier. The compiler has been fully integrated into the Microsoft Visual Studio environment in terms of the project system, build process, design tools, syntax highlighting, and the IntelliSense context-sensitive editing and documentation assistance.

The Spec# compiler differs from an ordinary compiler in that it does not only produce executable code from a program written in the Spec# language, but also preserves all specifications into a language-independent format. Having the specifications available as a separate, compiled unit means program analysis and verification tools can consume the specifications without the need to either modify the Spec# compiler or to write a new source-language compiler.

The Spec# compiler can preserve the specifications in the same binary with the compiled code because it targets the Microsoft .NET Common Language Runtime (CLR) [11]. The CLR provides rich metadata facilities for associating many types of information with most elements of the type system (types, methods, fields, *etc.*). The Spec# compiler attaches a specification to each program component for which a specification exists. (Technically, the specifications are preserved as strings in custom attributes. All

names are fully resolved; while this renders the format quite verbose, it makes it much easier for any tools consuming it.)

As a result, we made the design decision to have Boogie consume compiled code, rather than source code. An additional benefit is that Boogie can be used to verify code written in other languages than Spec#, as long as there is an *out-of-band process* for attaching contracts to such code. We use such a process to attach specifications to the .NET Framework Base Class Library (BCL), see Section 3.3.

3.1 Run-Time Checking

Spec# preconditions and postconditions are turned into inlined code. We do this not only for performance reasons, but also to avoid creating extra methods and fields in the compiled code. All such inlined code is tagged so that code corresponding to the Spec# contracts can be differentiated from the code that comes from the rest of the Spec# program. Such separation is required by any analysis tool that consumes Spec# contracts from the metadata. For instance, Boogie must be able to determine if the non-contract code in a method meets its postcondition, rather than the combination of the non-contract code followed by the code that checks the postcondition. The inlined code evaluates the conditions and, if violated, throws an appropriate contract exception.

To check object invariants, the compiler adds a new method to each class that declares an invariant. Special object fields, such as the invariant level [4] and owner of an object [46], are added to the super-most class that uses Spec# features within each subtree of the class hierarchy. As we mentioned in Section 2, the runtime does not enforce the whole methodology; for instance, run-time checking does not check that an object is exposed before updating a field. This means that an error may go undetected at run time that would be caught by Boogie.

3.2 Static Verification

From the intermediate language (including the metadata), Spec#'s static program verifier, Boogie, constructs a program in its own intermediate language, BoogiePL. BoogiePL is a simple language with procedures whose implementations are basic blocks consisting mostly of four kinds of statements: assignments, asserts, assumes, and procedure calls (*cf.* [48]).

An inference system processes the BoogiePL program using interprocedural abstract interpretation [16, 58] to obtain properties such as loop invariants. Any derived properties are added to the program as assert statements or assume statements. The BoogiePL program then goes through several transformations, ending as a verification condition that is fed to an automatic theorem prover. The transformations, such as cutting all loops to derive an acyclic control flow graph by introducing *havoc* statements, are done in a way that preserves the soundness of the analysis. A havoc statement assigns an arbitrary value to a variable; introducing havoc statements for all variables assigned to in a loop causes the theorem prover to consider an arbitrary loop iteration. All feedback from the theorem prover is mapped back onto the source program before it is delivered to the user [44]. The result is that programmers interact with Boogie's

prover only by making changes at the program source level, for instance by adding contracts.

Currently, Boogie uses the Simplify theorem prover [19], but we intend to switch to a new experimental theorem prover being developed at Microsoft Research.

3.3 Out-of-Band Specifications and Other Goodies

All .NET applications use the Base Class Library (BCL) in one form or the other. Thus we want to provide specifications for the entire BCL. This gives any client an immediate benefit even before writing a single contract.

But this raises a problem: how to provide a mechanism for attaching Spec# contracts to code that was written without them? (Note that we cannot modify the BCL even if we would use its implementation, since doing so would break versioning.) *Out-of-band* specifications, that is, specifications for code external to Spec#, are compiled into a Spec# repository. The repository is consulted in case the Spec# compiler or Boogie encounters a method or class for which it requires a specification (*i.e.*, when the compiler emits run-time checks or when Boogie generates verification conditions), but the method or class in the original code does not have an attached specification.

Writing contracts for self-contained examples is easy, but realistic programming is highly dependent on libraries, such as the BCL. A large obstacle then is obtaining contracts for the existing libraries. A companion project is working on semi-automatically generating contracts for existing code. It has automatically extracted almost three thousand preconditions for the current version of the BCL.

We have plans to build an explainer that translates Spec# method contracts into natural-language documentation entries. For example, it seems that one could translate preconditions and throws sets into the stylized exception tables used in the .NET documentation, see Figure 1. This could better keep the documentation accurate and up-to-date.

Lastly, we are planning a tool for translating Spec# into plain C#. (There are still some problems, like figuring out what to do with field initializers, that we need to address.) This tool will allow the use of Spec# within the normal development process. For instance, most Microsoft development groups insist on building their products using only official Microsoft compilers. In this context, Spec# would function as a precompiler; however, it is this invisibility that is important to gaining acceptance in a rigorous build environment.

4 Related Work

A number of programming languages have been designed especially with correctness or verification in mind. These include the pioneering languages Gypsy [2], Alphard [64], Euclid [39], and CLU [50], which offered different degrees of formality. In Gypsy, which was the first language to include specifications as an integral part of the programming language, the specifications integrated in the source program were aimed directly at program verification via an interactive theorem prover. Alphard was designed around a programming methodology for designing and proving object-like data structures, but

the proofs were done by hand. In Euclid, specifications written in the programming language's boolean expressions were checked at run time, with the idea that more complicated specifications, which were supplied in comments, would be used by some external program-verification tool. The CLU programming methodology prominently included specifications, but these were recorded only as stylized comments.

Three modern systems with contracts that have had a direct effect on practical programs are Eiffel [55], SPARK [3], and B [1].

Eiffel [55] is an object-oriented language with almost 20 years of use. The standard library is well documented through contracts, so contracts fall prominently within the purview of programmers. The contracts are enforced dynamically. However, without a full methodology for **modifies** clauses and for object invariants in the presence of callbacks, it would not be possible to obtain modular static verification.

SPARK [3] is a limited subset of Ada, without many dynamic language features like references, memory allocation, and subclassing, yet large enough to be useful for many embedded applications. Praxis Critical Systems has used SPARK in the development of several industrial programs, and their measurements indicate that the rigor provided by SPARK can be cost effective [14]. SPARK offers a selection of static tools, from light-weight sanity checking to full verification with an interactive theorem prover. Compatibility with an existing language has been a high priority in the design of SPARK, just like for Spec#, but their approach is quite different from ours. By ruling out difficult features of Ada, SPARK achieves the property that any SPARK program can be compiled by any standard Ada compiler while retaining its SPARK meaning (all SPARK specifications are placed in stylized Ada comments, and thus they are not used by the compiler). To meet our goal of migrating normally skilled programmers to a higher-integrity language, we have been unable to follow SPARK's approach of designing a subset of an existing language. Instead, we have designed Spec# to be a superset of an existing language, aiming to support easy and gradual adoption of its new features.

The B approach [1] uses a different methodology for writing programs: starting from full specifications and supporting a machine-aided process for stepwise refining the specifications into compilable programs. The resulting programs are similar in expressiveness to SPARK programs. This methodology, which has been used with success for example in constructing the Paris Metro braking system software, produces only correct programs. However, the skills needed to go through the refinement process make for a steep learning curve for the system and become a barrier for many programmers. It is also not obvious how to extend the methodology to more expressive abstractions, like those in object-oriented programs today.

The Java Modeling Language (JML) [40, 41] is a notation for writing specifications for Java programs. JML specifications, which include rich flavors of method contracts, are recorded in Java source code as stylized comments. An impressive array of tools have been build around JML, including tools for documentation, run-time checking, unit testing, light-weight contract checking, and program verification [13]. Spec# provides a more focused methodology than JML, which for example has yet to adopt a full story for object invariants in the presence of callbacks. The design space of Spec# is somewhat less constrained than JML, since JML does not seek to alter the underlying

programming language (which, for example, has let Spec# introduce field initializers and **expose** blocks).

The language AsmL [33] has many of the same aspirations as Spec#: to be an accessible, widely-used specification language tailored for object-oriented .NET systems. However, AsmL is oriented toward supporting model-based development with its facilities for model programs, test-case generation, and meta-level state exploration [6]. Our experiences in using AsmL for interface specification [9], run-time verification [10], and an on-going project with a product group [5] contributed to the design of Spec#. The companion testing tool SpecExplorer [31], currently in use within Microsoft, uses the Spec# language to provide model-based testing with features for test-case generation, explicit-state model checking, and run-time conformance checking.

The Anna [51] specification language for Ada lets programmers write down important design decisions. The specifications are compiled into run-time checks.

The first mechanical systems for proving programs correct were conceived and built several decades ago. These include the early, but not entirely automatic, systems of King [38, 37] and Deutsch [21], Gypsy [28], and the Stanford Pascal Verifier [52]. More recent program verifiers include Penelope (for Ada) [32] and LOOP (for Java and JML) [61], both of which require interactive theorem proving.

Setting early efforts by Sites [60] and German [27] into full motion, the Extended Static Checker for Modula-3 (ESC/Modula-3) [20] changed the rules of the game by leveraging the power of an automatic theorem prover not for proving the full functional correctness of programs, but for the limited aim of finding common errors in programs. Continuing in that tradition, ESC/Java [25] wrapped that technology with a simpler contract language (a subset of JML), aiming to deliver a practical high-precision tool for normally skilled programmers. A key ingredient that enables these ESC tools to do useful checking is the willingness to miss certain errors, since that can lead to a simpler specification language and to better odds for the automatic theorem prover to succeed (see also [43]). Boogie attempts to completely verify a program without missing errors; its ability to do so is bound to depend on the simplicity of the specifications.

Spec# provides a limited type system for non-null types. A more comprehensive type-system solution has been proposed by Fähndrich and Leino [23]. Their design deals with the complication of non-null fields by introducing additional *raw* types for partially-constructed objects.

Various abstraction facilities that help define **modifies** clauses in modern object-oriented languages have been proposed (*e.g.*, [47, 56, 42]).

Our methodology for object invariants and **modifies** clauses relies on object ownership to impose a structure on the heap [4, 46, 7]. Similar effects have been achieved by ownership types and other alias-confinement strategies (*e.g.*, [17, 12]). The earliest such use we've seen dates back to Alphard [64], where the modifier **unique** specifies that a field points to an owned object.

5 Concluding Remarks

The foundation of the Spec# programming system is the Spec# programming methodology, the Spec# language, the Spec# compiler, and the Boogie static program verifier. The methodology prescribes for the first time how to deal soundly with object invariants

and subclasses in a modular setting. The Spec# language embodies the methodology: Spec# enriches C# with non-null types, contracts, checked exceptions, comprehensions, and quantifications. The Spec# compiler uses a combination of static-analysis techniques and run-time checks to guarantee soundness of the language. The verifier tries to check the consistency between a program and its specifications.

We are trying to make the Spec# system a practically useful software tool that enables normally skilled programmers to write down and verify their assumptions. Therefore, we start from a familiar programming language and use the metaphor of type checking for exposing the new capabilities of our static checking technology. We do not offer a way to axiomatize new mathematical theories. Rather our design focus is on limited, partial functional specifications, those that can be written using boolean expressions of the language and quantifiers.

We have designed Spec# to provide incremental benefit as programmers use more of its features. Even without writing their own specifications, programmers get immediate benefit as their Spec# code is checked against the partially specified Base Class Library. Programmers gradually receive more benefit as they add, for example, non-null types and preconditions to their code.

Our design of Spec# has focused on sequential programs, but we are already extending our methodology to styles of concurrent programs [36]. It seems plausible that Spec# could also be of direct help in building secure applications. It would be interesting to explore the combination of our methodology with the stack walking mechanism of code access security in the context of existing libraries for permissions, authentication, and cryptography.

Acknowledgments

Many colleagues have helped make Spec# what it is: Colin Campbell, Rob DeLine, Manuel Fähndrich, Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Peter Müller and David Naumann have contributed to the more advanced versions of our methodology for object invariants. Rob and Manuel are also members of the Boogie project. Jim Larus and Sriram Rajamani have provided support and helpful discussions. Craig Schertz provided the tool for extracting contracts from existing code. A big thanks goes to Herman Venter, who has been invaluable in the implementation of the Spec# programming language and development environment. Finally, we thank Gary Leavens, Peter Müller, and others for their useful comments on a draft of this paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: A language for specification and implementation of verifiable programs. *SIGPLAN Notices*, 12(3):1–10, March 1977.
3. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003. With Praxis Critical Systems Limited.
4. Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

5. Mike Barnett, Wolfgang Grieskamp, Clemens Kerer, Wolfram Schulte, Clemens Szyperski, Nikolai Tillmann, and Arthur Watson. Serious specification for composing components. In Ivica Crnkovic, Heinz Schmidt, Judith Stafford, and Kurt Wallnau, editors, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, May 2003.
6. Mike Barnett, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Towards a tool environment for model-based testing with AsmL. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, October 2003.
7. Mike Barnett and David A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 54–84. Springer, July 2004.
8. Mike Barnett, David A. Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In Erik Poll, editor, *Proceedings of the ECOOP Workshop FTfJP 2004, Formal Techniques for Java-like Programs*, pages 11–19, June 2004. University of Nijmegen, NIII report NIII-R0426.
9. Mike Barnett and Wolfram Schulte. The ABCs of specification: AsmL, behavior, and components. *Informatica*, 25(4):517–526, November 2001.
10. Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts. *The Journal of Systems and Software*, 65(3):199–208, 2003.
11. Don Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2002.
12. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.
13. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. To appear.
14. Roderick Chapman. Industrial experience with SPARK. Presented at SIGAda'00, November 2000. Available from http://www.praxis-cs.co.uk.
15. Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, April 2003. Iowa State University, Department of Computer Science, Technical Report TR #03-09.
16. Patrick Cousot and Rhadia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM, January 1977.
17. Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 36, number 5 in *SIGPLAN Notices*, pages 59–69. ACM, May 2001.
18. Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, June 2004.
19. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.
20. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

21. L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, 1973.
22. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings 18th International Conference on Software Engineering*, pages 258–267. IEEE, 1996.
23. Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003*, volume 38, number 11 in *SIGPLAN Notices*, pages 302–312. ACM, November 2003.
24. Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001*, volume 36, number 11 in *SIGPLAN Notices*, pages 1–15. ACM, November 2001.
25. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.
26. Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposium in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
27. Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.
28. Donald I. Good, Ralph L. London, and W. W. Bledsoe. An interactive program verification system. In *Proceedings of the international conference on Reliable software*, pages 482–492. ACM, 1975.
29. John B. Goodenough. Structured exception handling. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 204–224. ACM, January 1975.
30. James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.
31. Wolfgang Grieskamp, Nikolai Tillmann, and Margus Veanes. Instrumenting scenarios in a model-driven development environment. Submitted manuscript, 2004.
32. David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.
33. Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theoretical Computer Science*, 2005. To appear.
34. C. A. R. Hoare. An axiomatic approach to computer programming. *Communications of the ACM*, 12:576–580,583, 1969.
35. C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.
36. Bart Jacobs, K. Rustan M. Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In *Proceedings of the workshop on Specification and Verification of Component-Based Systems*, 2004. To appear.
37. James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
38. James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, September 1969.

39. Butler W. Lampson, James J. Horning, Ralph L. London, James G. Mitchell, and Gerald J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, October 1981. An earlier version of this report appeared as volume 12, number 2 in *SIGPLAN Notices*. ACM, February 1977.
40. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.
41. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003.
42. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, volume 33, number 10 in *SIGPLAN Notices*, pages 144–153. ACM, October 1998.
43. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175. Springer, January 2001.
44. K. Rustan M. Leino, Todd Millstein, and James B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 2004. To appear.
45. K. Rustan M. Leino and Peter Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, September 2004.
46. K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, June 2004.
47. K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.
48. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999.
49. K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM 2004—Second International Conference on Software Engineering and Formal Methods*, pages 218–227. IEEE, September 2004.
50. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
51. D. C. Luckham. *Programming with Specifications: An Introduction to Anna, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
52. D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.
53. Charles C. Mann. Why software is so bad. *MIT Technology Review*, July/August 2002.
54. Steve McConnell. *Code complete: A practical handbook of software construction*. Microsoft Press, 1993.
55. Bertrand Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, 1988.
56. Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

57. RTI Health, Social, and Economic Research. The economic impact of inadequate infrastructure for software testing. RTI Project 7007.011, National Institute for Standards and Technology, May 2002.
58. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
59. Alexandru Sălcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, MIT, May 2004.
60. Richard L. Sites. *Proving that Computer Programs Terminate Cleanly*. PhD thesis, Stanford University, May 1974. Technical Report STAN-CS-74-418.
61. Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer, April 2001.
62. Mickey Williams. *Microsoft Visual C# .NET*. Microsoft Press, 2002.
63. Jeannette Marie Wing. *A two-tiered approach to specifying programs*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, May 1983. MIT Laboratory for Computer Science TR-299.
64. William A. Wulf, Ralph L. London, and Mary Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2(4):253–265, December 1976.

Mastering Test Generation from Smart Card Software Formal Models

Fabrice Bouquet[1,2], Bruno Legeard[1,2], Fabien Peureux[2], and Eric Torreborre[1]

[1] LEIRIOS Technologies
2C, chemin de Palente – 25000 Besançon, France
{fabrice.bouquet,bruno.legeard,eric.torreborre}@leirios.com
http://www.leirios.com
[2] Laboratoire d'Informatique (LIFC)
Université de Franche-Comté – CNRS – INRIA
16, route de Gray – 25030 Besançon, France
{bouquet,legeard,peureux}@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/∼bztt

Abstract. The growing complexity of new smart card platforms, including multi-subscription or multi-application functionalities, led up to more and more difficulty in testing such systems. In previous work, we have introduced a new method for automated test generation from state-based formal specifications (B abstract machines, UML/OCL models, Z specifications). This method uses cause-effect analysis and boundary computation to produce test cases as sequences of operation invocations. This method is embedded in a model-based test generator which has been exercised on several applications in the domain of smart card software (GSM 11-11 application, electronic purse system and Java Card transaction mechanism). In all these applications, a B abstract machine was built specifically for automatic test generation by an independent validation team. Writing a specific formal model for testing has been shown to be cost-effective, and has the advantages that it can be tailored towards the desired test objectives. This paper focuses on showing the application of this test generation process from formal models in the context of Smart Card applications. We describe how the test generation can be controlled by using several model coverage criteria. These criteria are of three kinds: multiple condition coverage, boundary-value coverage and behavior coverage. This makes it possible to generate a systematic minimal test suite achieving strong coverage results. The test engineer chooses the criteria depending on the application test objectives and then fully controls the test generation process.

Keywords: Automated test generation, functional testing, boundary testing, formal specifications, smart card standard.

1 Introduction

In the current industrial practice, more than 50% of the effort of smart card software development is devoted to dynamic testing, i.e. testing based on code

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 70–85, 2005.

execution. Currently, the validation process results in multiple and expensive phases of testing activity. Testing covers several aspects. Schematically, there are three families of dynamic testing:

- Unit testing is carried out during the programming activity. It makes sure that each elementary element (procedure, module, method...) has a correct behavior, and aims at avoiding errors in these elementary elements during the execution.
- Functional testing aims at ensuring the correctness of operations and their conformance to the functional requirements (standards or in-house specifications).
- Performance testing (load testing or stress testing), aims at ensuring the system performance when it is subject to significant competition in the access to resources (processor, memory, disk, network...).

This paper focuses on functional testing (or functional validation) on the basis of automated test generation and execution from a formal model of the smart card software under validation.

Currently, the industrial practice for functional testing is mainly manual and empirical. On the basis of technical requirements documentation, the validation engineers manually design test cases and write test scripts which are executed on the system under test. The difficulties of this approach are well-known. The empiricism of the test design prevents functional coverage of the specifications to be guaranteed; the quality of functional testing essentially depends on the know-how of the validation engineer, with poor rationale and reproducibility. So it is very difficult to control the duration of the testing phase and to determine its end. This constitutes an indicator of the low maturity of the validation process. The limits of these empirical practices are being reinforced by the increase in the complexity of the systems to be validated. Indeed, what was controllable at a certain level of complexity can not be anymore when the possibilities of interactions within the system strongly increase. Let us take the example of Pin Code management in GSM Smart Cards. In the 2G standard (i.e. GSM 11.11 [1]), manually designing the tests was feasible to obtain various scenarios (including wrong CHV1 or CHV2 attempts, decreasing the CHV counter, blocking the CHV, unblocking, ...) and a good coverage of functionalities. With the arrival of 3G cards and new standards (i.e. ETSI TS 102 221 [2]), the complexity of multiple Pin and administration codes, with security domains and different levels of applications, leads to great difficulty in managing the test design using informal methods.

The need to offer better methods and tools for functional testing has given rise to a large amount of research on generating tests from a formal model of the specifications, see for example [3–7]. Formal methods, and particularly model-oriented notations such as UML/OCL [8] and B [9] allow an abstract formalization of the expected behavior of the system under test. These notations are well-suited for modeling smart card software because of the expressiveness and abstraction level of set-oriented logic constructs. Moreover the definition of

an explicit model helps along both test case generation and expected results (i.e. the test oracle) synthesis.

These last few years, a number of studies and industrial applications on model-based testing for smart card software validation were conducted with documented success. It concerns smart card applications or operating systems. For example, [10] presents results on the CEPS (Common Electronic Purse Specifications) standard, [11] shows validation results on the GSM 11-11 standard, [12] uses automated test generation on the WAP Identity Module (WIM) and [13] describes results applied to the Java Card transaction mechanism. So, model-based techniques can be apply at different levels of smart card software: from applications on the basis of the modeling of the APDU to the validation of some parts of the JCVM or JCRE. This approach makes it possible to really validate a smart card product at the system, subsystem or component level, against its requirement specifications.

In previous work [14, 15], we have presented a new technology for automated test generation from B formal models which produces test cases as sequences of operations invocations. The underlying method is based on a symbolic animation of the formal model and uses several testing strategies like cause-effect analysis and boundary computation [16]. This technology has been extensively exercised for smart card applications and is now productized under the name LEIRIOS Test Generator (LTG). LTG makes it possible to generate tests cases and oracle from the formal model and to translate these abstract generated test cases into executable test scripts for a particular smart card test execution environment. This allows a fully automated process for functional testing using a formal model of the smart card application under test as an input and producing the verdict assignment for each generated test case.

This paper focuses on mastering the test generation process from formal models in the context of smart card applications. We describe how test generation can be controlled by using several model coverage criteria. These criteria are of three kinds: multiple condition coverage, boundary-value coverage and behavior coverage. This makes it possible to systematically generate a minimal test suite achieving strong coverage results. The test engineer chooses the criteria depending on the application test objectives and then fully controls the test generation process.

Therefore, the paper is organized as follows. Firstly, we present the test generation process using the LTG technology in the context of smart card software validation. Secondly, we describe the implementation of this method using a simplified version of a smart card application. This application is modeled with the B notation.

2 Test Generation Using Leirios Test Generator Tool

The Leirios Test Generator (LTG) tool results from research undertaken in the Laboratory of Computer Science of the University of Franche-Comté (LIFC) [14, 15, 17]. LTG is based on the symbolic animation of formal specifications and various test generation strategies (i.e. cause-effect testing and boundary testing [16]).

The symbolic animation makes it possible to traverse the reachability graph of the formal model to generate test cases (as sequences of operations) on the system under test.

LTG provides three user interfaces (see Figure 1):

- the animation interface allows to validate the formal model by simulating its execution,
- the test generation interface makes it possible to drive the generation process through coverage and selection criteria,
- the reification interface makes it possible to concretize the generated abstract test cases into test scripts executable on the system under test.

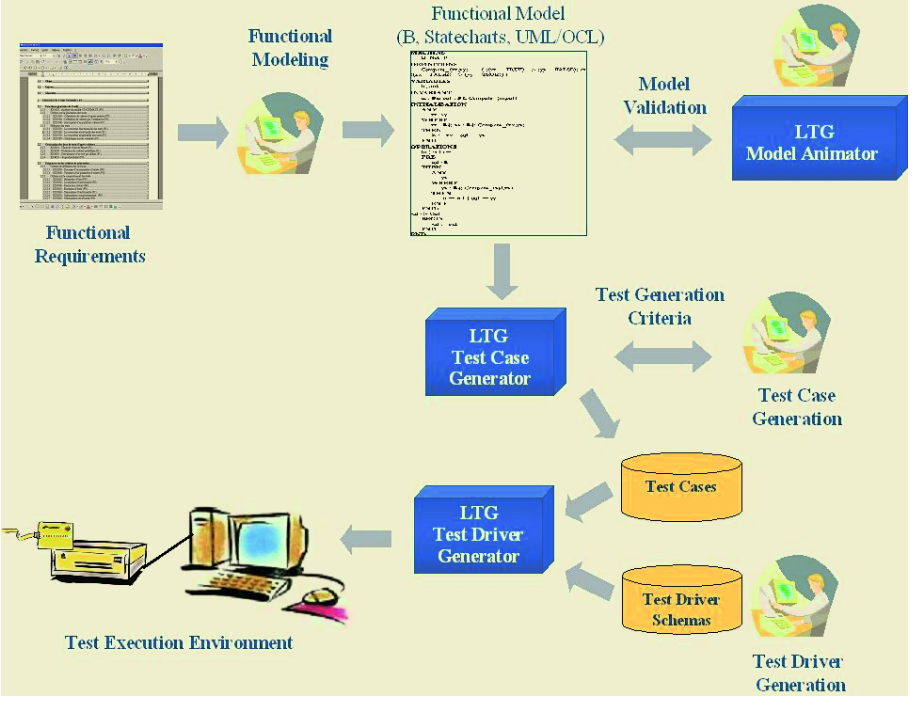


Fig. 1. Test generation process

LTG can currently process three notations:

- statecharts (STATEMATE) [18],
- B notation [9],
- UML class and state diagram with OCL constraints [8].

Each of these notations is translated into the LTG tool intermediate format [19]. This format is itself translated into a set of constraints that makes it

possible to symbolically evaluate the system, basis of animation and test generation process. In this section, we describe each stage of the LTG generation process in the context of smart card applications on the basis of a B notation modeling.

2.1 Formal Modeling and Test Generation

The automatic generation of functional tests is based on a formal model of the functional specifications (standard or proprietary requirements) of the system under test. This model shows several characteristics. First of all, it is an abstract functional model: it represents the expected visible behaviors of the system under test, but it does not integrate the implementation details (it is not a program nor an architectural representation). However, abstraction of some features like cryptographic algorithms for example can be done in the formal model. Then, this model ought to be sufficiently precise to allow the generation of test cases (as sequences of operations) including expected outputs. The test cases/expected outputs pair are directly used during test cases execution on the system under test, to obtain an automated verdict (success or fail). Moreover, the model can be adapted to the test objectives for a given validation campaign: if the test purposes only relate to one part of the system (for example because other parts have been tested elsewhere), the model has to take it into account, in particular to avoid a useless combinatorial explosion in the test generation. Finally, the model has to take into account the control and observation points of the system under test. On one hand, the system under test provides various commands or APIs to activate the execution. Basically, for smart card software, it is mainly APDUs or APIs of JCVM byte-code. On the other hand, the behaviors of the system can be observed through various output data like status words for example. These elements must appear in the model so that the tests can be executed on the system under test (it is not needed to model control points that cannot be activated, or data that cannot be observed).

In practice, these modeling rules for the test generation are not very constraining. It enforces, in fact, good engineering practice. That way, only one functional model of the system under test is used either as a reference to clarify technical requirements and functional specifications, and as an input of the automated test generation process. So, this model can be developed before the test generation phase. In case of proprietary specifications, it helps validating the requirements during the phase of the requirements elicitation. The model can also be developed in a specific way for the test generation as a basis of the functional validation process.

The state of the art in software engineering presents a lot of modeling notations. Various paradigms were exploited to make it possible to characterize the expected behaviors of a system, and to offer techniques and tools for validation and verification of the model. LTG uses various notations depending on application target:

– Statechart diagrams constitute a notation adapted to use the operational semantics of automata, embedded controllers for example.

- The B notation allows to model systems with complex data on which specific treatments are performed. This notation is well-suited to the modeling of smart card applications or operation systems. B allows a good abstraction level and balance between data and control modeling.
- The UML notation covers an extremely broad diversity of diagrams (11 diagrams in UML 2.0); this constitutes an advantage from the point of view of expressivity but a disadvantage since some UML diagrams do not have precise semantics. The test generation with LTG takes as input UML class diagrams and state diagrams with OCL constraints in order to obtain enough precise models.

The smart card application introduced in section 3, and used to illustrate the LTG test generation process, is formalized with the B notation.

2.2 Driving the Test Generation Process

The generation of tests is steered by the validation engineer on the basis of coverage and selection criteria set on the model.

The coverage criteria make it possible to choose the model coverage to use during test generation. Three families of criteria are proposed: the coverage of decisions, the coverage of variables boundary values, and the coverage of the behaviors. These criteria allows the validation engineer to control the test case explosion on either a coarse or fine basis. The selection criteria make it possible to focus the generation on a part of the model, variables or states in order to achieve a specific test objective [20].

2.2.1 Coverage Criteria. The user interface allows the validation engineer to drive the test generation process using some coverage criteria.

Multiple Conditions in the Decisions

A decision is a predicate which determines a specific behavior. It is built from elementary conditions. When a decision is constituted by a disjunction of conditions, then several levels of coverage can be applied. Each level corresponds to a coverage criterion as following:

Decision Coverage (DC): A test set achieves DC when each decision is tested with a true result, and also with a false result.

Decision/Condition Coverage (D/CC): A test set achieves D/CC when each condition and each decision in the program are tested with a true result, and also with a false result (it also achieves both DC and CC coverage).

Modified Condition/Decision Coverage (MC/DC): A test set achieves MC/DC when each condition in the program is forced to true and to false, in a scenario where that condition is directly correlated with the outcome of the decision. A condition is directly correlated with its decision if the result of the decision changes when the result of the condition changes.

Multiple Condition Coverage (MCC): A test set achieves MCC if it exercises all possible combinations of condition outcomes in each decision.

These criteria correspond to various classical structural coverage criteria (code coverage). The choice of a criterion directly influences the number of generated tests. For example, for a decision $Cond_1 \vee Cond_2 \vee Cond_3$ (where $Cond_1$, $Cond_2$ and $Cond_3$ are atomic conditions), the DC criterion produces one test, both D/CC and MC/DC criteria produce three tests whereas MCC criterion produces eight tests to cover the true decision (each criterion produces one more test to cover the false decision).

Equivalent Values Coverage

The behaviors resulting from the model can often be activated with some variable valuation.

The choice for this criterion is a choice of validation. If the test engineer considers that the values are uniform, then the choice will be made with the criterion *one value*. If it appears useful to generate some tests with various values of a variable, then the criterion *All the values* will be selected.

Behavior Coverage

The LTG tool aims at covering all the modeled behaviors of the system. The concept of behavior corresponds to a set of modifications of the system state during the execution of an operation (representing a command, an API or an APDU). In the B abstract machine notation, a behavior corresponds to an independent path through the control-flow graph of an operation.

Two coverage criteria are proposed:

- *All behaviors*: each behavior of each operation is activated at least once in the test set,
- *Pairs of behaviors*: each behavior of each operation is activated at least once in the test set, and is followed by all the executable behaviors of each operation.

The criterion *Pairs of behaviors* is to be used with the greatest caution, because it provides, for N behaviors, at worst N^2 test cases.

2.2.2 Selection Criteria. The selection criteria aim at focusing the test generation on a fragment of the model to finely control the adequacy of the generated tests with the objectives of the test campaign and to control the number of generated tests. The following selection criteria are proposed in the LTG tool:

- *Operation selection*: the user can choose to test just a part of the modeled command,
- *State variable value selection*: the user can choose to focus the test generation on specific state variables (updated with specific values).

2.3 Generation Method

The LTG test generation method consists of testing all the possible behaviors of the specification operations, by traversing the boundary states of the system,

which are states where at least one state variable has a value at an extremum – minimum or maximum – of its subdomains. This strategy is controlled by the coverage and selection criteria previously defined. This method is performed as follows:

1. Partitioning of the model operation to generate all the possible behaviors,
2. Computation of variable domain boundaries from each behavior (called boundary goals),
3. Generation of test cases obtained, for each boundary goal, by traversing the constrained reachability graph of the specifications from the initial state to reach a state satisfying a boundary goal (this state is called boundary state).

In this approach, a test case is a sequence of the abstract operations defined in the model. The obtained test cases are thus abstract. Therefore, it is necessary to translate these sequences of invocations in order to obtain executable test scripts. The step that makes it possible to transform the abstract test cases into executable concrete scripts, is called reification or concretization step.

2.4 Executable Test Script Generation

Generated test cases define sequences of operation invocations at an abstract level. More precisely, each operation invocation appears with the signature of the formal model and the input values are the same as those from the abstract data model. We defined and implemented a solution for translating the generated test cases into executable scripts [13]. This solution is as follows. The test engineer defines two inputs: a test script pattern and a mapping table. The test script pattern is a source code file in the target language with some tags indicating where sequences of operation invocations have to be inserted (cf. Figure 2).

The mapping table contains three kinds of information (cf. Figure 3):

1. The operations and the substituted variables on the abstract operations.
2. The monitoring of the variables and the associated operations.
3. The source equivalence instructions of the operations and the variables.

The first kind of information is extracted automatically from the model. The two others must be given by the test engineer. This table is similar to the representation mappings defined in [21]. In fact, the variables represent all kinds of data from the model (as constants). It is possible to define all data values into the reification process. For example, the modeled values `on` or `off` can be translated into 2 concrete system values `1` and `0`.

The mapping table can be used in two ways:

- Putting directly into the table the translation code. This is more simple, but it has three issues:
  - The first issue is mapping, it is not always possible to have a simple translation from the abstract level data to the concrete one. For example, if you don't model the encryption process and you need it for an external authentication, you must call it into your reification process to compute the expected result.

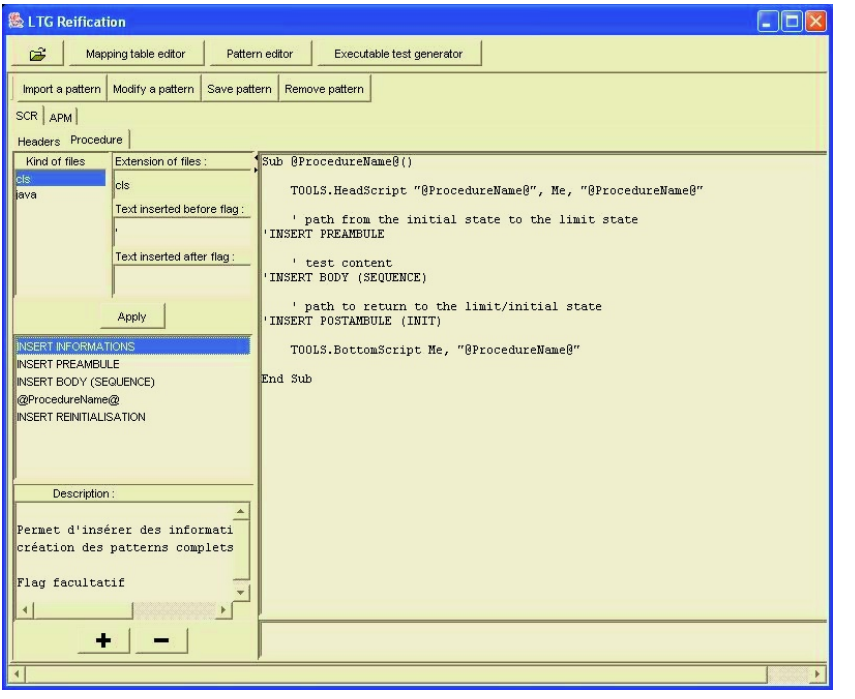


Fig. 2. Pattern example

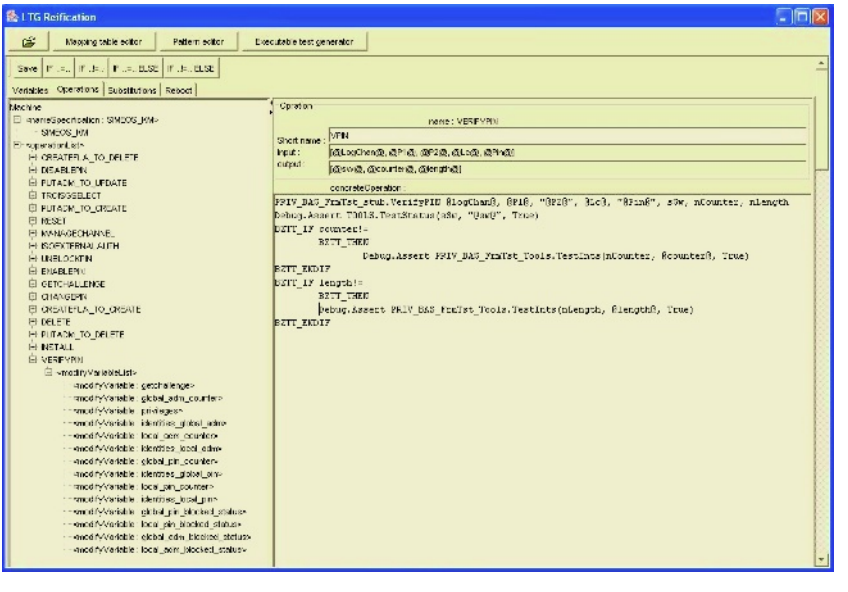


Fig. 3. Mapping Table example

  - The second issue is source code validation. It is difficult to validate all code in subpart of the table.
  - The third issue is the use of exceptions. It is the case with the JavaCard. When you use a JavaCard, some expected exceptions must specifically be catched. This treatment adds complexity into the code and validation, and it appears to be not practicable to include these exceptions into the formal model.
- Using an adapter library. This is better for two reasons:
  - Table readability because all knowledge is coded into the library
  - Library validation can be done before the test generation.

3 Case Study Example

In this section, we show the use of the LTG approach on a simplified example of the ETSI TS 102 221 smart card standard [2]. This standard specifies the interface between the Integrated Circuit Card (ICC) and the terminal independently of the manufacturer, card issuer or operator. During the communication, the ICC is passive: it only answers the requests sent by the terminal. The applications contained in the terminal access and modify the files of the ICC through defined functions and by respecting access conditions of each file. The abstraction level of the presented example only concerns the function used to verify the pin code.

3.1 B Formal Model

Every elementary file of the ICC containing data has its own specific read access conditions. The relevant access condition must be fulfilled before the requested action can take place. Three types of access condition can be found in the standard:

- $ALWays$: the action can be performed without any restriction,
- $NEVer$: the action cannot be performed over the ICC terminal interface (the ICC may perform the action internally),
- PIN (user verification): the action is possible if a correct PIN code has already been presented to the terminal during the current session.

The PIN level, once satisfied, remains valid until the end of the card session as long as the corresponding PIN code remains unblocked, i.e. after three consecutive wrong attempts, not necessarily in the same card session, the access rights previously granted by the PIN code are lost immediately.

The function $VERIFY\_PIN$ is used to verify the PIN code presented by the terminal by comparing it with the relevant one stored in the ICC. The verification process is allowed only if PIN is not blocked. If the presented PIN code is correct, the number of remaining PIN attempts shall be reset to its initial value of 3 and the PIN access condition is satisfied. If the presented PIN code is false, the number of remaining PIN attempts shall be decremented. After 3 consecutive false PIN presentations, not necessarily in the same card session, the

PIN shall be blocked. This function always returns a status word. This response, which has the form of a hexadecimal code, takes one of the values described in Table 1.

Table 1. Potential response codes

Responses codes	Descriptions
9000	Successful PIN verification
$63CX$	Unsuccessful PIN verification (X indicates the number of further allowed retries)
6983	PIN blocked

A simplified B operation of the $VERIFY\_PIN$ function is presented in Figure 4.

$sw1, sw2 \longleftarrow$ **VERIFY_PIN**$(code) \mathrel{\hat{=}}$
PRE
$code \in \{code_1, code_2, code_3\}$
THEN
$IF\ (counter\_PIN = 0)$
$THEN$
$sw1 := 698 \parallel$
$sw2 := 3$
$ELSE$
$IF\ (PIN = code)$
$THEN$
$counter\_PIN := 3 \parallel$
$sw1 := 900 \parallel$
$sw2 := 0$
$ELSE$
$counter\_PIN := counter\_PIN - 1 \parallel$
$sw1 := 63C \parallel$
$sw2 := counter\_PIN - 1$
END
END
END;

Fig. 4. $VERIFY\_PIN$ B formal specification

3.2 Partition Analysis

To compute test cases, firstly a translation scheme from B generalized substitutions to before-after predicates is performed on the operations of the specification. This translation scheme is precisely defined in the B-Book [9]. Basically, it consists of unfolding predicates along branches, and introducing primed variables to denote the after values, using the *prd* rules from [9, Chap. 6].

Next, the before-after predicates of each operation can be seen as a control graph. This control graph is built with arcs and nodes where the arcs out of a node are alternative choices, each arc contains a decision predicate conjoined with a substitution in predicate form, and each path through the graph is the conjunction of its arcs. Note that it is a specific kind of control graph because B abstract machines have no loops. Figure 5 shows the control graph of the $VERIFY\_PIN$ operation (only decision predicates are printed on the arc).

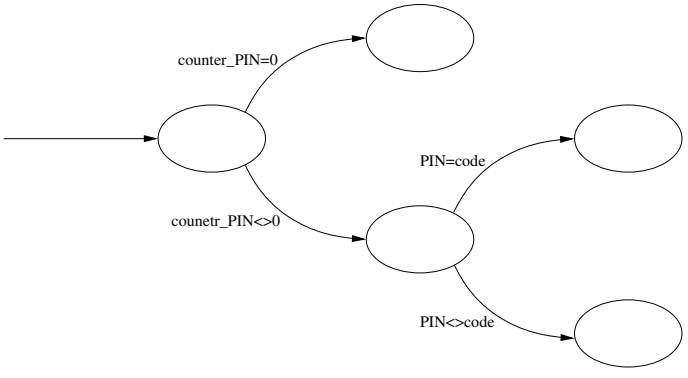


Fig. 5. Control graph of $VERIFY\_PIN$

Each path through this control graph corresponds to one *effect* (or *behavior*) of the operation. That is why one path is called *Effect Disjunctive Normal Form* predicate (*EDNF*) [22], or more simply, effect predicate. The set of all the effect predicates corresponds to the set of all possible behaviors of the original B operation. The use of effect predicate is similar to the commonly-used disjunctive normal form introduced in previous work [3], except that the disjunction operator is not initially considered as an alternative choice. This view makes it possible to independently analyze multiple conditions in the decisions applying one of the coverage criteria introduced in Section 2.2.1.

This way, each coverage criterion on multiple conditions in the decisions results in one specific control graph. The translation rules to be applied for each criterion are precisely explained in [22]. The use of such criteria on the running example has no sense because all the decisions of the $VERIFY\_PIN$ operation are build without disjunction.

The effect predicates, resulting from the control graph of the $VERIFY\_PIN$ operation (Figure 5), are shown in Table 2. Although it does not appear in this running example, the state invariant is assumed to hold for all the identified effect predicates.

These generated effect predicates are directly used to generate boundary goals and hence boundary states, basis of the LTG test generation process.

Table 2. Effect predicates of $VERIFY\_PIN$

No	Effect predicates	
	Before	**After**
P1	$code \in \{code_1, code_2, code_3\} \wedge$ $counter\_PIN = 0$	$sw1' = 698 \wedge$ $sw2' = 3$
P2	$code \in \{code_1, code_2, code_3\} \wedge$ $counter\_PIN \neq 0 \wedge$ $PIN = code$	$counter\_PIN' = 3 \wedge$ $sw1' = 900 \wedge$ $sw2' = 0$
P3	$code \in \{code_1, code_2, code_3\} \wedge$ $counter\_PIN \neq 0 \wedge$ $PIN \neq code$	$counter\_PIN' = counter\_PIN - 1 \wedge$ $sw1' = 63C \wedge$ $sw2' = counter\_PIN - 1$

3.3 Boundary Goal Calculation

The next step of the method consists in calculating one or more boundary goals from the before-state of each effect predicate. This calculation is performed by applying variable value coverage criteria, introduced in Section 2.2.1, on the before-state of effect predicates. Each resulting boundary goal is also a set of constraints that defines a specific value or domain for each variable used in the considered before-state predicate.

Table 3 shows the boundary goals calculated from the effect predicates of the $VERIFY\_PIN$ operation using the criterion *one value* and a minimization and maximization of the natural variable $counter\_PIN$.

Table 3. Boundary goals from $VERIFY\_PIN$

Effect Predicates	Boundary goals	No
P1	$counter\_PIN = 0$	$BG1$
P2	$counter\_PIN = 1$ $PIN \in \{code_1, code_2, code_3\}$	$BG2_1$
	$counter\_PIN = 3$ $PIN \in \{code_1, code_2, code_3\}$	$BG2_2$
P3	$counter\_PIN = 1$ $PIN \in \{code_1, code_2, code_3\}$	$BG3_1$
	$counter\_PIN = 3$ $PIN \in \{code_1, code_2, code_3\}$	$BG3_2$

3.4 Test Case Generation

A generated test case corresponds to a sequence of operation invocations. The test case is divided into four parts (Figure 6) and thus follows the ISO9646 standard [23].

The first stage of the method is to generate a path (the preamble) from the initial state to each boundary state of the specification. A boundary state is any state that satisfies a boundary goal.

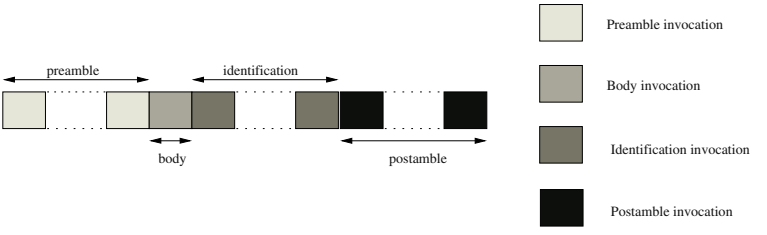


Fig. 6. Test case constitution

At this boundary state, one or more boundary values are chosen for the input parameters of the operation invocations of the body, applying one of the value coverage criteria previously introduced in Section 2.2.1. The selected criterion is also applied on the input parameters of the effect predicates in a similar way that it was done on the state variables. The number of operations to be activated in the body directly depends on the selection criteria introduced in Section 2.2.1 (one operation for the *All the behaviors* criterion, and two for the *Pairs of behaviors* criterion).

Next, an identification part consists of operation invocations whose aim is to determine certain observable aspects of the system at the end of the test body. In addition to the possible output data returned by the operation(s) in the body, the output values returned by this third part enable to assign a verdict: the results obtained by the simulation of the specifications are those expected during the execution of the same sequence on the implementation.

Finally, the postamble makes it possible to take the system from the final state of the identification part to the final state of the test case. This last part is used either to reach a specific state in order to link several test cases together, or to put the system into a final state in order to generate consistent traces.

Table 4 shows the test cases, generated from the $VERIFY\_PIN$ operation and the boundary goals of the Table 3, using the criterion *one value* for the *code* input parameters (the correct code specified through the PIN state variable is arbitrary assigned to the value $code_1$), and the *All the behaviors* criterion for the body. Due to the simplicity of the example, this table only presents the preamble and the body of the test cases (the initial state of the machine respectively assigns 3 and $code_1$ for the state variables $Counter\_PIN$ and PIN).

4 Conclusion

Model-based testing technologies are now mature enough to be introduced in the industrial process of smart card functional testing. This will help to face the challenge of functional validation of more and more complex smart card software.

These new validation technologies are based on formalizing the specifications of the smart card products. This will represent a real effort. Yet, the test generation makes it possible to obtain an immediate return of investment of this formalization task. In [11], we have shown that including the modeling task, the automated test generation process saves 30% of manpower compared with the

Table 4. Test cases generated from $VERIFY\_PIN$

Boundary Goals	Preambles	Bodies
$BG1$	VERIFY_PIN($code_2$) VERIFY_PIN($code_2$) VERIFY_PIN($code_2$)	VERIFY_PIN($code_1$)
$BG2_1$	VERIFY_PIN($code_2$) VERIFY_PIN($code_2$)	VERIFY_PIN($code_1$)
$BG2_2$	empty	VERIFY_PIN($code_1$)
$BG3_1$	VERIFY_PIN($code_2$) VERIFY_PIN($code_2$)	VERIFY_PIN($code_2$)
$BG3_2$	empty	VERIFY_PIN($code_2$)

manual design process. Also, using formal modeling for test generation will help to reach the EAL-5 level of the common criteria. Model-based testing is not an intrusive process and can be easily introduced in the current smart card development process. Model-based test generators produce executable test scripts that can directly be introduced in the configuration management system and run nightly for non-regression testing. The main change concerns the role of the test engineer: instead of spending a large amount of time writing test scripts (which is a tedious and error prone task), he/she can concentrate on more high value work including specification modeling and test verdict analysis.

In this paper, we have described a toolbox for reducing and controlling test case explosion, which is a crucial issue for the scalability of test generation. This is mainly based on coverage and selection criteria. It allows the validation engineer to cleverly steer the generation process.

References

1. European Telecommunications Standards Institute, F-06921 Sophia Antipolis cedex - France. *GSM 11-11 V7.2.0 Technical Specifications*, 1999.
2. European Telecommunications Standards Institute, F-06921 Sophia Antipolis cedex - France. *ETSI TS 102 221 V4.4.0 Technical Specifications*, 2001.
3. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the International Conference on Formal Methods Europe (FME'93)*, volume 670 of *LNCS*, pages 268–284. Springer Verlag, April 1993.
4. P. Stocks and D.A. Carrington. Test templates: a specification-based testing framework. In *Proceedings of the 15^{th} International Conference on Software Engineering (ICSE'93)*, pages 405–414, Baltimore, Maryland, May 1993. IEEE Computer Society Press.
5. R. Hierons. Testing from a Z specification. *The Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.
6. L. Van Aertryck, M. Benveniste, and D. Le Metayer. CASTING: a formally based software test generation method. *In 1^{st} IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 99–112, 1997.
7. S. Behnia and H. Waeselynck. Test criteria definition for B models. In *Proceedings of the World Congress on Formal Methods (FM'99)*, volume 1708 of *LNCS*, pages 509–529, Toulouse, France, 1999. Springer Verlag.

8. Object Management Group. UML Standard. `http://www.omg.org`, 2003.
9. J-R. Abrial. *The B-BOOK: Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0 521 49619 5.
10. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. Automated test and oracle generation for smart-card applications. In *Proceedings of the International Conference on Research in Smart Cards (e-Smart'01)*, volume 2140 of *LNCS*, pages 58–70, Cannes, France, September 2001. Springer Verlag.
11. E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications: GSM 11.11 standard case-study. *The Journal of Software Practice and Experience*, 34(10):915 – 948, 2004.
12. J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, and K. Scholl. Model-based test case generation for smart cards. In *Proceedings of the 8^{th} International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)*, volume 80 of *ENTCS*, Trondheim, Norway, June 2003. Elsevier.
13. F. Bouquet and B. Legeard. Reification of executable test scripts in formal specification-based test generation: the Java Card transaction mechanism case study. In *Proceedings of the International Conference on Formal Methods Europe (FME'03)*, volume 2805 of *LNCS*, pages 778–795, Pisa, Italy, September 2003. Springer Verlag.
14. B. Legeard and F. Peureux. Generation of functional test sequences from B formal specifications - Presentation and industrial case-study. In *Proceedings of the 16^{th} International Conference on Automated Software Engineering (ASE'01)*, pages 377–381, San Diego, USA, November 2001. IEEE Computer Society Press.
15. B. Legeard, F. Peureux, and M. Utting. A comparison of the BTT and TTF test-generation methods. In *Proceedings of the International Conference on Formal Specification and Development in Z and B (ZB'02)*, volume 2272 of *LNCS*, pages 309–329, Grenoble, France, January 2002. Springer Verlag.
16. B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.
17. F. Ambert, F. Bouquet, B. Legeard, and F. Peureux. Automated boundary-value test generation from specifications - method and tools. In *Proceedings of the 4^{th} International Conference on Software Testing (ICS-Test'03)*, pages 52–68, Köln, Germany, April 2003. Software and Systems Quality Conferences.
18. Statemate Tool. `http://www.ilogix.com`, 2003.
19. F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. BZ-TT: A tool-set for test generation from Z and B using constraint logic programming. In *Proceedings of the CONCUR'02 Workshop on Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic, August 2002. INRIA Technical Report.
20. H. Zhu, P.A.V. Hall, and J.H.R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.
21. D.J. Richardson and S.L. Aha T.O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14^{th} International Conference on Software Engineering (ICSE'92)*, pages 105–118, Melbourne, Australia, May 1992. ACM Press.
22. B. Legeard, F. Peureux, and M. Utting. Controlling Test Case Explosion in Test Generation from B Formal Models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81–103, 2004.
23. ISO. Information Processing Systems, Open Systems Interconnection. *OSI Conformance Testing Methodology and Framework – ISO 9646*, 1998.

A Mechanism for Secure, Fine-Grained Dynamic Provisioning of Applications on Small Devices

William R. Bush[1], Antony Ng[2], Doug Simon[1], and Bernd Mathiske[1]

[1] Sun Microsystems Laboratories, Mountain View CA 94043, USA
bill.bush@sun.com
http://research.sun.com/
[2] D'Crypt Pte. Ltd

Abstract. As small, secure devices become more powerful and more wide spread, it has become desirable to support the dynamic provisioning and updating of multiple applications on such devices. This paper presents a simple mechanism for performing such provisioning and updating, even if the applications are mutually distrustful. The mechanism extends CLDC Java™technology with a classfile attribute that carries the certificates necessary to enable the added security.

1 Background

The work described here was motivated by a number of developments and considerations:

- Small, secure devices, such as smart cards and cryptographic modules, are becoming more capable.
- Such devices are being used in more complex situations running multiple applications.
- Updating the software on such devices once deployed is highly desirable, to provide both new functionality and software fixes, but poses various security issues.
- A dynamic provisioning mechanism supporting such activity should be small and simple, because of device limitations and to aid in verification and certification.
- The Java platform has appeared on small devices and provides dynamic class loading and some basic security features.

The resulting solution presented here:

- supports the secure incremental replacement and extension of software on small devices,
- enables distinct trust communities developing distinct applications,
- accomplishes this by extending a common version of the Java platform, and
- enables additional support for capabilities and running untrusted code.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 86–107, 2005.

2 The Java Context

The Java platform was the starting point for this investigation because it already provides various features supporting security and dynamic provisioning. Unlike C and C++, for example, it guarantees type and pointer safety. Various Java versions provide different forms of application isolation. And all versions support some mechanism for dynamically installing classes.

2.1 The Connected Limited Device Configuration

The work described here is specifically aimed at the next generation of smart cards and other similar small devices. Thus the technology base used is the Connected Limited Device Configuration (CLDC) version of Java 2 Platform, Micro Edition (J2ME^{TM}technology) [1]. This version of the Java platform is the smallest one that supports most standard features of the Java language (in contrast to the much more restrictive Java Card^{TM}specification [2]). It outlines a basic set of libraries and Java virtual machine features. Compliance with the CLDC specification is demonstrated by passing the CLDC Technology Compatibility Kit (TCK) tests [3].

The heart of the configuration is the K Virtual Machine (KVM) [4]. The KVM is a virtual machine designed with the constraints of small devices in mind. Named to indicate that its size is measured in tens of kilobytes, the KVM is simple, in order to minimize memory footprint. This simplicity makes the KVM easy to understand and modify, important characteristics in the context of security.

2.2 CLDC Security

The CLDC security model [1, section 3.4] defines three types of security, low-level VM security, application-level security, and end-to-end security.

Low-level VM security is defined as the characteristic that "an application must not be able to harm the device in which it is running, or crash the virtual machine itself." In the context of the KVM this means that CLDC verification must be done on all classfiles.

End-to-end security refers to network-based solution-oriented security, which is outside the scope of the CLDC specification.

Application level security is defined as controlling access to external resources, which is done on the larger J2SE ^{TM}platform by the security manager [5]. The security manager was deemed to have too large a memory footprint for the CLDC, so a sandbox model is used instead. Specifically:

- Only a limited set of APIs is available (the CLDC libraries, profiles, and manufacturer-specific classes).
- Such system classes cannot be overridden.
- No user-defined class loaders are allowed.
- No native functions can be dynamically loaded onto the device.

- The class lookup order may not be manipulated.
- By default, an application may only load classes from its own JAR file.
- In addition, a CLDC implementation need not support multiple concurrent applications.

The CLDC security model is a good starting point for a more secure platform. It is small, simple, and relatively static, which is good both for small devices and for increased security. It has a static set of APIs, system classes, and native functions, and a single system class loader. It has a simple application model. It is possible to be compliant with the CLDC and provide greater security than the CLDC mandates.

2.3 MIDP Security

The Mobile Information Device Profile (MIDP) [6] is a set of additions to the base CLDC platform that supports mobile phones. Among the additions is a security mechanism [7], [8]. The MIDP security mechanism is based on two concepts: protection domains and JAR file signing.

A protection domain is a set of permissions granted an application, and defines the application's sandbox (see [9] and [10] for descriptions of nuanced sandboxes). An application runs in a single protection domain. A MIDP platform may define various domains, but required domains include Manufacturer, Operator, Third-Party, and Untrusted. Some permissions may only be granted through interaction with the user of the device (confirming use of the permission).

JAR file signing is used to verify the authenticity of an application. MIDP requires that an application reside in a single JAR file, which is typically signed using X.509 PKI infrastructure, support for which is required by the MIDP standard. The MIDP device uses the certificates it possesses to authenticate the application.

This security model has limitations with respect to high security devices. Specifically:

- Permissions are coarse grained and set when the device is manufactured. A set of permissions is *a priori* bound to a domain, and an entire application then executes in one of those predefined domains.
- Support for X.509 PKI is required, which may not be appropriate and can be cumbersome.
- User interaction may be required to grant some permissions.

In sum, the MIDP platform has been carefully tuned for mobile phones. High security devices are different.

2.4 Compatibility and Security Goals

The broad goal of this work is to develop a more secure version of the KVM (the Secure KVM, or SKVM), with a particular focus on dynamic provisioning. More specifically:

- Correct CLDC/KVM applications should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
- Correctly implemented secure applications operating normally should not be able to distinguish the KVM from the SKVM on the basis of observed behavior.
- Only malicious classes should elicit different behavior from the SKVM than they would from the KVM.
- The SKVM must pass the proper compatibility tests (the CLDC TCK).

Additionally, the SKVM should be capable of being validated as secure, specifically achieving FIPS 140-2 certification [11]. FIPS 140-2 is a specification drawn up by the National Institute of Standards and Technology, defining security requirements for cryptographic modules. In addition to being the stipulated requirement for any cryptographic module acquired by the US government, FIPS140-2 has become a de facto standard for cryptographic equipment and provides a level of assurance that the equipment was designed with adequate consideration of security. The standard spells out requirements in 11 different areas including physical security, hardware security, software security, and key management. A cryptographic module can be certified to any of four increasing levels of assurance. For example, the IBM 4758 has been certified at the highest level (of the predecessor FIPS 140-1 standard) [12].

3 Key Precepts

A number of key precepts guided the design of the SKVM architecture (see [13], [14], [15]).

3.1 Simplicity

The overriding precept is simplicity. It has several important benefits:

- It minimizes bugs and possible points of compromise.
- It keeps the system's memory footprint small.
- It makes the system easier for application developers to understand, which in turn makes it easier for them to implement secure applications.
- It makes the system easier to validate for security. Such validation typically involves modeling security state transitions with a state machine (this is the paradigm required for FIPS 140-2 certification).

3.2 Fail-Safe Design

When a fail-safe system encounters an unanticipated condition, it always lapses into a conservative, secure state. Such conditions can be genuinely unanticipated or can be a result of a partial malfunction of the system. No matter how comprehensively a system is analyzed, it is unlikely that all possible combinations of conditions have been anticipated. Fail-safe design ensures that the system makes

conservative assumptions and lapses into a secure state when an unanticipated condition is encountered.

A watchdog signal in a battery-powered system is an example of fail-safe design. As long as periodic signals arrive from the watchdog, the CPU continues normal operation. If the battery runs low, the periodic watchdog signal is terminated, and the CPU shuts down. Note that if the watchdog signal is interrupted in any other way (for example, due to physical tampering), the CPU also shuts down – it lapses into a secure state.

3.3 Static Specification of Security Policy

The ability to modify policy dynamically is usually considered a desirable feature. For a secure device, it is also a major source of weaknesses. Dynamic modification of security policy is almost always a cause of subtle bugs. In contrast, a static security model forces the application developer to consider the security aspects of the software architecture earlier and more completely in the design process. A static model is also in general easier to analyze and vet because it is simpler and usually has fewer states and state transitions than a dynamic model.

3.4 Explicit Specification of Security

The combination of static security specification and fail-safe design dictates that security issues – specifically the granting of privileges – involve an explicit act on the part of an object, and that any privilege not explicitly granted is automatically denied. The SKVM implements this precept within the confines of the semantics of the Java programming language (hereinafter referred to as the Language).

As an example of how the Language semantics affects this precept, note that a class has the right to manipulate those parts of it (including protected data) that are inherited from its ancestors. Thus when a class grants privileges to another class, through the Language semantics it automatically and implicitly grants (some) access to all its ancestors in the inheritance tree.

3.5 Security at the VM Level

Implementing security policy using only classes is attractive for several reasons, including extensibility and uniformity. However, good security engineering suggests that the core of the security framework be implemented at the VM level. The challenge is to keep the bare minimum in the VM level and leave as much as possible at the language level so that a high level of security assurance can be established without compromising system flexibility.

3.6 Reliance on Data Authenticity, Not Secrecy

Secrets stored on a device, such as symmetric or private keys, introduce potential vulnerabilities and complicate responses to security compromises. They also

complicate the device and increase its cost since the device must now defend the secret against disclosure. In contrast, use of verifiably authentic data, such as certificates verifiable with public keys, does not create such weaknesses.

The SKVM is designed so that security assurance relies only on the ability of the device to keep data authentic. The SKVM does not require the device to keep a secret.

4 The SKVM Security Architecture

This and following sections present the components of the SKVM security architecture:

- The notion of trust;
- The implementation of trust;
- The characteristics of other necessary VM features; and
- The application model.

4.1 Owners, Trust Relationships, and the Trust Community

In a well-designed, secure, closed system every class, interface, and package – every component – has an owner, an entity (nominally a human) with ultimate responsibility for it.

Systems are often assembled from components developed by different owners. One owner may or may not trust another, based on their relationship. Owners that do trust one another form a community, however informally, and grant each other privileges.

These basic observations are the foundation of the SKVM security architecture. It implements these notions of *ownership* and *trust* in the context of the CLDC.

4.2 Trusted Classes

The SKVM supports trust relationships and trust communities by providing the framework and features necessary to request and grant privileges. The SKVM itself maintains no explicit information on trust relationships and trust communities other than what each class brings in.

Intuitively, a trusted class provides functionality for some trusted community of owners. Trusted classes are the means by which sensitive information is encapsulated. Trusted classes also have privileges and can in turn grant privileges to other trusted classes.

As mentioned earlier, a trusted class has an owner. It is the responsibility of the owner to request and obtain the necessary privileges for the trusted class. If class X needs a particular privilege from class Y, the owner of class X will have to acquire this privilege from the owner of the class Y. These privileges come in the form of certificates signed by Y's owner and held by class X. They are verified by the SKVM when class X is loaded.

All the certificates of a trusted class are bundled into a new class attribute called a trust attribute. (Class attributes are the classfile mechanism used to

store extra information about a class, and are described in the Java Virtual Machine Specification [16, section 4.7].)

Classes that do not have a (valid) trust attribute have no privileges and are untrusted. Untrusted classes are not allowed to load or execute (subject to an optional feature discussed in Section 7).

4.3 Trusted Classes and Subclassing

The fundamental privilege enforced by the SKVM is a subclassing privilege that enables one class to install itself as a subclass of another. In some sense this privilege is the privilege to modify, in a controlled way, the code on the device. It is referred to as the S privilege. Subclassing also includes the power to access the protected fields, methods, and constructors of all the superclasses of the subclassing class, regardless of package.

The owner of the root class `java.lang.Object` (typically the owner of the device running the SKVM) grants the S privilege to the trusted owner of each of Object's direct subclasses. These owners in turn grant S privileges to owners they trust. Thus, for the S privilege, the owner of Object can trace a chain of grants of the S privilege to the owner of every class on the device.

Note that accesses to static and instance methods and fields are controlled through the Language's private, protected, and public tagging [17, section 6.6]. Note that the Java Virtual Machine Specification stipulates that accesses have to be checked at run time [16, sections 5.4.3 and 5.4.4]. Information about private, protected, and public access permissions are stored in the classfile [16, sections 4.5 and 4.6] to enable such run time checking. Also note that the ability to subclass a class does not imply the ability to subclass a parent of the class in the class hierarchy independently.

4.4 Trusted Classes and Packages

The Language employs the package construct to bundle groups of classfiles, not necessarily related in the class hierarchy, into a single name space [17, chapter 7]. Packages provide a natural way of organizing and referring to classes and methods. Significantly, classes within a package have access rights to each other's protected fields and methods. Each class is contained in exactly one package (possibly the unnamed package).

In the SKVM, package access must be controlled in order to control access to protected fields and methods (see [18], page 189).

The mechanism used in the SKVM for controlling package access has three elements:

- A package has an owner and an owner-managed key-pair, analogous to that used for subclassing.
- A package's public key is part of the package's name. If someone tries to put a class in a package without the right public key, the class will be put in a different package. The SKVM simply uses the public key as part of the name space reference.

- Every class is in a package. If a class does not specify a package the SKVM puts it in the unnamed package.

Note that the VM does not have an *a priori* list of packages. The VM first knows about a package when a class belonging to the package is loaded (or, optionally, when installed on the device; see below). The first loaded class defines the package to the SKVM, and any subsequent classes belonging to the same package are checked for consistency. It is the responsibility of all classes in a package to identify the package identically, by name as well as public key.

There may appear to be a security weakness because classes bring in both the package signature as well as the key with which the signature is verified. In fact, while a malicious class could generate a fake package key and a signature consistent with this fake key, it would not be able to join an existing package because it would not be able to replicate the signature associated with the package's private key; it would instead be put in a distinct package.

The Language defines an unnamed package and assigns all classes that do not specify a package name to this unnamed package. A class that does not specify a package is automatically put into the unnamed package. Any package public key specified in the trust attribute of a class that does not specify a package name is ignored.

While the unnamed package is a convenience during code development, a secure application built for the SKVM should specify packages for all its classes. To encourage such a practice and to provide higher levels of security assurance, the SKVM has a flag that, when set, prevents the loading of any class that does not specify a package. The SKVM by default runs with this flag cleared. Once set, the flag cannot be cleared without restarting the SKVM.

4.5 Interfaces

With one notable exception, an interface specifies functionality without providing an implementation [17, chapter 9]. The exception is for static initialized fields. In such cases, the initializing expression may contain requests to instantiate objects, which may require SKVM privileges. It is therefore necessary to associate privileges with an interface.

As with a class, an interface has a nominal owner. The owner is responsible for securing all required privileges for the interface. A trusted interface X demonstrates that it can extend a trusted interface Y by presenting a certificate signed by Y. This certificate is analogous to the subclassing certificate and employs the same data structure and mechanisms.

Note that a class that implements an interface can be independently accessed and manipulated as a class in its own right. In such cases, normal Language semantics dictate what can be accessed.

4.6 Inner Classes

The Language allows the definition of inner classes as members of other classes [17, section 8.1.2]. These inner classes are implemented through compiler introduced source code transformations and appear to the VM as distinct classfiles.

The SKVM requires a trusted inner class to present a trust attribute, as any other trusted class would. It is the responsibility of the owner of the trusted inner class to generate this trust attribute. Of course, any tool that supports generation of trust attributes may wish to facilitate the construction of attributes for inner classes. For instance, the tool might handle all name transformations transparently, and employ the same key-pairs for the inner class as it employs for the outer, enclosing class.

The source code transformations introduced by the compiler to support inner classes implement a weakening of access permissions. This is necessary because there is no support in Java virtual machines for direct access to a private member of a class from another class. The specific instances of access permission weakening are:

- Private inner classes are implemented as package level classes.
- Protected inner classes are implemented as public level classes.
- Private class members (fields or methods) that are visible between classes (due to the shared scoping relationship between inner and enclosing classes) are indirectly implemented with package level access. Note that sharing of private members between classes participating in an inner class relationship is achieved by a local protocol of access methods that reflect the mode of access expressed in the source. These methods have package scope and as such are open to any class within the same package.

Like other VMs, the SKVM cannot reliably identify inner classes and therefore cannot determine when such access permissions have been weakened. Therefore, developers for the SKVM platform must be aware of these issues. The problems due to weakened access permissions can be avoided by adopting the following guidelines:

- Classes in an enclosing/inner class relationship should never rely on the shared scoping of their private members.
- Inner classes should never be declared private or protected.

Following such guidelines will ensure that there is always a one-to-one correspondence between the source level access permissions of a class and its classfile implementation.

The use of anonymous inner classes should be avoided due to the difficulties of managing their trust relationships.

The use of non-static inner classes should also be avoided since they are in a sense syntactic sugar for static inner classes and thus hide detail that makes security analysis harder.

4.7 Exceptions and Trust Relationships

When an exception occurs in a running program, the VM unwinds the call stack until the most recently installed relevant exception handler is encountered, which then catches the exception [17, section 11.3] [16, sections 2.16.2 and 3.10]. The code that throws the exception is never resumed.

There are security issues in adopting such a model directly in the SKVM. For instance, a class could install an exception handler and, at a later stage, a different class could throw an exception. This second class may not enjoy any trust relationship with the first class. As a result there is an unanticipated transfer of control that complicates security analysis and becomes a potential vulnerability.

Note that exceptions are simply standard objects with the additional property that they are derived (indirectly or directly) from `java.lang.Throwable`. As such, package access semantics can be leveraged to prevent classes external to a package from catching exceptions thrown from within the package. A non-public exception (one whose class definition does not include the public access modifier) is invisible to all classes outside its package and therefore no handler in these external classes can explicitly declare to catch exceptions.

Unfortunately, package access semantics do not completely control exception handlers. It is legal to hold a reference to an object even though the static type of the reference may preclude any knowledge of the object's complete type. This can be achieved with a reference to publicly accessible base type (such as `java.lang.Object`). This means that exception handlers can catch exceptions via base class declarations. The lowest common base class for every exception is `java.lang.Throwable`. A handler declared to catch such an exception would catch package-restricted exceptions. While Language semantics prevent the handler's scope from using the exception as an instance of its complete type, the mere fact that it can be caught presents a means to mask out or alter secure control flow transfer. Thus, a mechanism for preventing this interference is built into the SKVM.

Each class includes a flag within its trust attribute. If the flag is cleared, then any package-restricted exception thrown by X can only be caught by exception handlers within the same package as the throwing class. If this flag is set in class X, then standard exception semantics are applied when an exception is thrown by any method in X.

Note that if all classes that potentially throw exceptions set this flag, exception handling in the SKVM will be identical to, and compatible with, the KVM.

5 The Trust Attribute

The trust attribute is a collection of data that is attached to each trusted class and that determines its privileges. The trust attribute is primarily composed of a number of public keys. The use of these keys to sign and verify privileges constitutes the crux of the SKVM. The trust attribute is understood by the SKVM and ignored by other VMs.

With each package P there is an associated key pair (PK_P, pk_P), generated by the owner of the package. With each class X there is an associated key pair, the subclassing key pair (SK_X, sk_X). The SKVM does not require that all these key-pairs be distinct. Indeed, a key pair can be employed in multiple

roles depending on the underlying security policy that the SKVM is enforcing (subclass and package being the same, for example).

In the following, class X belongs in package P and wishes to subclass Y. The trust attribute for X is described in Table 1.

Table 1. Components of a Trust Attribute

T_X	A non-negative integral timestamp indicating the time of creation of X. This timestamp is used for version control on installation and loading, and it is assumed that newer versions have larger timestamps than older versions.
SK_X	The public key used for verifying attempts to subclass X.
$\mathrm{SH}_Y(X\|T\|Cert)$	A hash of class X, its timestamp T_X, and all the fields in the trust attribute *minus this field*, signed by the subclassing private key of parent class Y. This hash is the signature that is used to validate the subclassing privilege, as well as the authenticity of the classfile and the trust attribute.
PK_P	The public key of the package P (if any) that X belongs in. This is needed for identification of packages and exception processing.
$\mathrm{PH}_P(X\|T)$	The hash of class X and its timestamp T_X signed by the private key pk_P of package P. This signature is verified with the public key PK_P and is the means by which the SKVM knows that class X belongs in package P.
EF_X	A constant specifying how package-private exceptions are handled in the face of handlers declared to catch them via publicly accessible base classes. The constant takes on values **yes** (all classes can catch package-private exceptions thrown by this class) and **no** (only trusted classes in the same package as the class throwing the exception can catch it).

The classfile described here refers to the CLDC classfile, which includes the traditional J2SE classfile and the stack-maps generated by the CLDC preverifier.

The timestamp above is part of the hash in order to validate the time the class was hashed. This guarantees the integrity of versioning, which is based on the timestamp.

A rogue class cannot use the public key of another package because it will not be able to generate the necessary signed hash, since it does not have access to the private key. The rogue class could generate a separate key pair (in which case it would have the private key of that pair), but the public keys would not match the package keys of other classes and the class will end up in its own package. Since packages are determined by equivalence classes defined on the relation of "equality of public keys", it is not possible for a rogue class to forge admission to a package.

The EF_X flag in the trust attribute is designed to be fail-safe. Specifically, the false or **no** setting is the secure setting. It is assumed that the false or **no** setting is associated with the zeroed state in the platform (typically integer 0 or Boolean False).

Now consider what happens when the SKVM receives class X and wishes to install it. As a trusted class, X subclasses Y and should be installed as its child. Class X demonstrates that it has this privilege by presenting $\mathrm{SH}_Y(X|T|Cert)$. This same signature also establishes that the owner of Y has vouched for the integrity of the contents of X. This can be verified with Y's subclassing public key, which can be found in Y's certificate. (Since Y is already installed, its certificate must have been previously validated.)

Class X proves that it has the privilege of belonging in package P by subjecting the signed hash $\mathrm{PH}_P(X|T)$ to verification using the public key stored with the package. Membership in an existing package is demonstrated by using the same package public key as an existing class.

6 Contextual Issues

The SKVM as described above requires certain platform support to operate properly. In particular, classfiles stored on device must be handled correctly, and specific cryptographic functions must be available.

6.1 When Trust Attributes Are Checked

In general, the arrival and storage of classfiles on a device will occur before the SKVM needs to load them (as is generally the case with CLDC platforms). In addition, some of the new classfiles arriving on a device may replace existing ones. These circumstances make it potentially desirable to check trust attributes at times other than class loading.

The CLDC specification [1, section 5.3] gives the platform implementor considerable freedom. The classfile lookup order is implementation dependent and a classpath is not required. It is required that the lookup order cannot be manipulated by the application programmer in any way. (Note that the platform must read classfiles and JAR files [1, section 5.3.1]. Applications that are "distributed publicly" on a network open to the public must be in JAR format. Also note that the JAR file loading boundary required by the CLDC specification is not necessary in the SKVM, since the SKVM uses a stronger security mechanism. Nonetheless, the SKVM supports this boundary in strict KVM mode.)

The SKVM follows the CLDC specification and thus does not impose an ordering or, therefore a particular time for checking trust attributes. The earliest attributes can be checked is when classfiles arrive on a device; the latest is when they are executed for the first time (per application), the traditional load time).

Various issues arise that affect order considerations. First, for devices that may install classfiles from a potentially malicious source, buffer overflow attacks via classfiles with bogus attributes are possible. A device could be flooded with apparently proper classfiles that can only be flushed when their attributes are checked. Second, newly arrived classfiles may dynamically replace a subset of

the classes in an application, which requires versioning and the rechecking of attributes.

The way to deal with the buffer overflow problem is to check trust attributes when classfiles arrive on the platform. Otherwise, classfiles would have to be kept around indefinitely. Applications must therefore be engineered to install their classes in an order that allows them to be verified as trusted when they are installed. The burden of meeting this constraint is on the application designer: the superclass must be either already installed or immediately available.

One implementation of this checking is to use an arrival buffer. Classes arriving on the device are initially put in this buffer and are moved into classfile storage when their trust attributes are verified. JAR files are also unpacked in this buffer. When a classfile arrives its subclassing certificate is checked against its superclass, if extant. If the trust attribute or superclass is absent, the classfile is flushed. If the subclassing certificate is valid the classfile is moved to classfile storage; otherwise it is flushed.

With respect to the versioning issue, the SKVM requires the trust attribute of each classfile to include a time stamp. The trust attribute checker (whenever run) inspects the time stamps, validates the newest class, and discards the old class (or marks it as to be deleted if the old classfile is being used by a running application).

Rechecking of attributes can be done in one of two ways. If attribute checking is done at installation then the installation of a replacement class in turn requires that classes granted privileges by it must be rechecked, if anything in its attribute has changed. If checking is done dynamically, then no extra processing is necessary. (Note that binary compatibility is important but is not an SKVM issue; it is rather handled by the VM in due course.)

6.2 Cryptographic Support

The SKVM requires cryptographic support to enforce security. Specifically, it requires two functions, one for digital signature support and one for cryptographic hash computation. The SKVM provides basic implementations of these functions, but allows deployment of custom versions.

The basic functions are RSA with a 1024-modulus key as the signature algorithm, and MD5 as the cryptographic hash function. RSA is a public key algorithm. It operates by creating two keys, a public key and a private key. As the names suggest, the private key is kept private and used for signing certificates while the public key is made known to everyone for verifying certificate signatures. MD5 is a collision-free message digest, or hash function. It computes a 128-bit hash value of an array of data.

Note that RSA is a *de facto* standard and has the advantage of very rapid signature verification times. However, this is at the expense of rather large signatures. This is not an issue for the subclass privilege, which is verified only on class loading and can be discarded afterwards. However, instantiation certificates (described in Section 7) have to be maintained in the VM, and each such certificate requires 1024 bits (128 bytes) of storage.

If standard RSA certificate size is found to be unacceptable, either elliptic-curve RSA, which is secure with approximately 155bits ($\approx$ 20 bytes), or DSA, for which the signatures are $2 \times 160 = 320 bits$ (40 bytes), can be employed. However, there is a (running time) performance penalty in the use of either ECC or DSA. In addition, DSA optimizes signing at the expense of verification and can be up to 100 times slower than RSA.

Message Authentication Codes (MAC) are an alternative to public key based hash-and-sign signatures. While MACs are considerably less demanding in terms of storage and computation time, a MAC requires a symmetric secret key. The use of MACs should be avoided because they make SKVM integrity depend on secrets internal to the SKVM, violating the precept against secrecy. Additionally, they require infrastructure for the storage and management of secret keys, and require the secure transmission of classfiles (since keys appear in the clear in the trust attribute). Nevertheless, the choice remains with the platform owner.

Platform owners can integrate custom signature and hash algorithms into the SKVM. However, for security reasons, these must be integrated at the VM level, and not at the class level, in keeping with the precept of static specification.

Note that the SKVM architecture does not specify how subclassing and instantiation verification keys are to be managed. Nor does it stipulate how the system protects itself against other forms of attack on public key systems such as spoofing and man-in-the-middle. Such issues depend on the requirements of the application and the trusted community. Note however that since all keys are public keys (unless MACs are used), confidentiality is not required and there is therefore no need to store secrets. All that is required is that the keys be authentic, and the mechanism by which classfiles are loaded ensures this.

6.3 Security-Related Exceptions

The SKVM throws an exception when a privilege verification fails. Depending on the circumstances, `IllegalSubclassException` or `IllegalPackageException` is thrown by the VM when a privilege certificate could not be verified successfully. When the VM throws one of these exceptions it uses an instance created at VM startup. Establishing whether or not an exception being propagated resulted from a security violation is thus reduced to a simple object pointer comparison.

6.4 SKVM Applications and Their Development

SKVM applications are CLDC applications: programs with a main method [1, section 3.2]. An application's component classes are loaded when necessary. When the class containing the main method is loaded, the application is registered and is then run. Class loading is controlled by the security mechanisms described above.

The SKVM employs a Java Application Manager, or JAM, similar to JAMs used with the KVM. The JAM assumes that there is local storage (typically a file system or a local database) that stores installed classes and from which

classes can be loaded. The SKVM architecture does not require that the complete application be resident – components may be loaded dynamically over a communication channel – although an implementation may impose this restriction. The JAM starts the SKVM and indicates to it which application (which classfile with a main method) should be run.

For the purposes of defining trusted communities and establishing the initial trust relationships, the CLDC library can be thought of as owned by the platform. Applications wishing to execute on the platform will have to request and obtain privileges to subclass the CLDC library.

If the platform is one of many being issued by an authority (such as a payment token being issued by a credit card company) then all platforms may share a common CLDC owner and hence have identical subclassing and instance-creation public keys.

For development purposes, a platform authority can release a version of the platform with a different CLDC signing key-pair created purely for application development, with the private key released to developers. This allows the developers to sign their classes each time they are changed, without having to request the authority to do so. When the application is complete, it is installed on the production platform with a different set of keys, with the CLDC signing key kept private.

The SKVM architecture does not support multiple applications running in the same virtual machine (but rather multiple suppliers of code). The goal of such support is to protect applications from each other to a degree comparable with process isolation on a standard operating system, less protection than the SKVM aims to provide.

The KVM supports the KVM Debug Wire Protocol [19], a debugging protocol that is a subset of the JDWP standard. The SKVM implements the KDWP, but only during debugging. It obviously must be removed for deployment. The KDWP may be enhanced to display the additional information present in the SKVM, such as that pertaining to trusted and untrusted space.

The security support in the SKVM is enabled by configuration settings. Given fail-safe design principles, these settings default to secure modes. However, they can be set so that all security features in the SKVM visible to developers are disabled, with the result that the SKVM is identical in function to the KVM.

For backward compatibility, the SKVM can be initiated in strict KVM mode. In this mode, SKVM security features are disabled. Trust attributes are not required for any of the classes. Strict KVM mode is enabled if the class containing the main method has no security attribute. The KVM default restriction that all classes in an application be in a single JAR file is enforced in strict KVM mode [1, section 3.4.2.3].

7 Additional Optional Functionality in the Trust Attribute

The trust attribute mechanism presented above can be extended with additional information to support other, optional, security-related features besides

dynamic provisioning. The specific extensions explored in the SKVM involve implementing a form of capability-based control and enabling limited execution of untrusted classes.

7.1 Support for Capability-Oriented Design

Good object-oriented and secure programming practice mandates the factoring and encapsulation of data. This factoring enables a capability-based style of programming [20], in which a capability is represented by an object, and references to that object are controlled. Ideally, all references to a capability object would be monitored, and unauthorized uses prevented, but the overhead of checking all references is too great.

A simpler (less powerful, and less secure) mechanism can be constructed to control the creation of objects (as capabilities) and references to static fields and methods. With this compromise scheme, the creation of capabilities is monitored but their use is not. Class-based references are also monitored.

This mechanism has been implemented in the SKVM via another trust attribute privilege: the privilege to create a new instance of (that is, instantiate) an object and reference its static methods and fields. This privilege is referred to as the **A** ("access") privilege, with the sense of accessing the resources of a class.

7.2 Domains

In practice the access privilege can be burdensome to administer. A collection of classes may want to share the privilege to access one another's class resources (instantiation and access to static methods and fields), especially if the classes have been developed together, provide a coherent module of functionality, or are within a shared security perimeter. To reduce the burden of maintaining individual class access privileges, the SKVM supports domains.

A domain privilege is shared among a group of classes in a domain and allows each class to instantiate all other classes in the domain and reference their static methods and fields. With domains, individual access privileges for each class are no longer required. A class may be in only one domain. Domains, reflecting security concerns only (as opposed to name space issues), are distinct from packages, but may be made coincident with them.

Typically, domains allow groups of classes that reference each other frequently (whether through instantiation or static method or field access) to do so without needing to verify access privileges each time. Such privilege verification involves verifying a signature against a public key and can be costly in execution time as well as memory (to store the signature). With domains, an inter-class resource access is permitted as long as the domain membership keys of the accessing and accessed class are equal. In this way, domains simplify application design, application development, and SKVM implementation.

7.3 Additions to the Trust Attribute to Support Capabilities

With each class X there is the associated class resource access key pair (AK_X, ak_X).

With each domain D there is an associated key pair (DK_D, dk_D) generated by the owner of the domain.

Consider a class X that belongs in package P and domain D and wishes to access the resources in class Z (not in domain D). The additional components of the trust attribute for X relating to capabilities for this type of access are described in Table 2.

Table 2. Components of a Trust Attribute for supporting Capabilities

AK_X	A public key used to verify access requests to the resources of class X.
$\mathrm{AH}_Z(X\|T)$	A hash of class X and its timestamp T_X signed by the class resource access private key of class Z. This signature is verified with the public key AK_Z. Note that there are as many hashes of the form $\mathrm{AH}_Z(X\|T)$ as there are resources that X needs to access from different classes.
DK_D	The public key of the domain D that X belongs in. This key determines the identity of the domain.
$\mathrm{DH}_D(X\|T)$	The hash of class X and its timestamp T_X signed by the private key dk_D of domain D. This signature is verified with the public key DK_D and is the means by which the SKVM knows that class X belongs in domain D.

If class X now wishes to create an instance of class Z, it demonstrates that it has this privilege by having the signed hash $\mathrm{AH}_Z(X|T)$, which can be verified with AK_Z, the public key found in Z's certificate.

Class X proves that it has the privilege of belonging in domain D by subjecting the signed hash $\mathrm{DH}_D(X|T)$ to verification using the public key stored with the domain. Membership in an existing domain is demonstrated by using the same domain public key as an existing class.

The ability to subclass or to create an object implies the ability to subclass or initialize any parent as part of the act of subclassing or creating the object. These are standard Language rules. Note however that the ability to subclass or create an object does not imply the ability to subclass or create an object of a parent class in the class hierarchy independently. The operation on the parent class can only happen as a direct and automatic result of the operation on the class itself. For example, if class X has permission to create an object of class B, which subclasses A, then it does not follow that X can explicitly create an object of class A. To do so requires that X have explicit permission from A.

As with packages, a rogue class cannot use the public key of another domain because it will not be able to generate the necessary signed hash, since it does not have access to the private key. The rogue class could generate a separate key pair (in which case it would have the private key of that pair), but the public keys would not match the domain keys of other classes and the class will end up

in its own domain. Since domains are determined by equivalence classes defined on the relation of "equality of public keys", it is not possible for a rogue class to forge admission to a domain.

7.4 Loading and Executing Untrusted Classes

Untrusted classes are ones without any trust attributes (as distinguished from classes with invalid trust attributes, which are mistrusted). Such classes can provide useful, CLDC-standard functionality if their execution is strictly controlled.

In the SKVM this is done by keeping untrusted classes in a sandbox and allowing trusted classes to grant privileges to untrusted ones. The degree to which trusted classes are prepared to grant privileges to untrusted classes defines the extent of the sandbox; the sandbox is not fixed or defined *a priori*.

When untrusted classes arrive on the SKVM platform they can be placed in their own arrival buffer, to preclude a buffer overflow attack on the trusted class arrival buffer.

7.5 Untrusted Classes and Privileges

Untrusted classes rely on trusted classes for all their privileges. Since untrusted classes have no certificates, from a security standpoint they are indistinguishable from one another. Privileges are granted uniformly to all untrusted classes. These privileges take three forms:

- An untrusted class may be allowed to subclass a trusted class. Like all other privileges granted to untrusted classes, this is a privilege that the trusted class in question must grant explicitly.
- In the capability-based style discussed above, and similar to a trusted class, an untrusted class may be granted the privilege to create a new instance of a trusted class. Again, this is a privilege that is explicitly granted by a trusted class to all untrusted classes uniformly.
- An untrusted class may be granted or denied the power to call a trusted method (usually static) or access a trusted field. This power is in addition to the Language's standard access control mechanisms, and is necessary for historical reasons.

Privileges granted to untrusted classes are specified with flags in the trust attribute of the trusted class. In addition, flags associated with each method and field in the trusted class indicate whether the method can be called from, or the field accessed from, an untrusted class. This enables an application to run untrusted classes written to the standard CLDC API while retaining some measure of control.

It would be simpler to have a flag that indicated whether all methods and fields in a trusted class could be accessed from untrusted classes. This would be sufficient if the trusted aspects of an application were well factored into specific trusted classes. Although most of the CLDC library can be handled with such a flag, there are cases in CLDC that break this principle. It is in general desirable

from a security factorization standpoint that SKVM applications be designed to use class-level security rather than relying on method-level or field-level control.

There have been attempts in the various releases of the Language to enumerate which functions in the core libraries are exposed to sandboxed classes. The flag mechanism provides a means by which such selective exposure can be accomplished. The flag mechanism also provides control over static methods and fields in classes like `java.lang.System` that cannot be instantiated. This is important since untrusted classes may need access to some fields and methods (such as `java.lang.System.out`) while other fields and methods (such as `java.lang.System.exit`) should not be accessible.

7.6 Additions to the Trust Attribute to Support Untrusted Classes

For a class X, Table 3 describes the additional components of the trust attribute that relate to untrusted classes.

Table 3. Components of a Trust Attribute for supporting Untrusted Classes

SF_X	A binary flag indicating if objects can subclass X without privileges. If the flag is false, then untrusted classes cannot subclass X and trusted classes need to present the appropriate certificate (signed hash) in order to subclass X successfully. If the flag is true, then all classes can subclass X as long as Language semantics are obeyed.
NF_X	A flag specifying if objects can instantiate X without privilege. If the flag is false, then an object can only instantiate X by presenting the appropriate certificate. If the flag is true, then all classes can instantiate X as long as Language semantics are obeyed.
MF_X	A constant specifying if all objects can invoke static methods in X. The constant takes on values **yes** (all static methods in X can be invoked by any object, subject to standard Language semantics), **no** (static methods in X can only be invoked by an object that presents the appropriate certificate), and **byMethod** (flags associated with each static method determine if the method can be invoked without privilege).
FF_X	A constant specifying if all objects can access static fields in X. The constant takes on values **yes** (all static fields in X can be accessed by any object, subject to standard Language semantics), **no** (static fields in X can only be accessed by an object that presents the appropriate certificate), and **byField** (flags associated with each static field determine if the field can be accessed without privilege).

These flags are designed to be fail-safe. Specifically, the false or **no** setting of the SF_X, NF_X, MF_X, and FF_X flags are secure settings. It is assumed that the

false or no setting is associated with the zeroed state in the platform (typically integer 0 or Boolean False).

8 Implementation

A version of the SKVM has been implemented using the KVM code base version 1.03. The implementation includes basic support for secure dynamic provisioning, support for capabilities, and support for untrusted classes.

In general, changes to the KVM were small and localized.

8.1 Secure Dynamic Provisioning

Seven files (out of 24) required modification, and one small file was added. The details of these changes are shown in Table 4. The total increase in size, in lines of code, is 4%.

Table 4. Code size overhead of Secure Dynamic Provisioning

File	LoC in base KVM	Additional LoC for SKVM	% Increase
class.c	1985	216	11%
collector.c	2096	54	3%
crypto.c	0	124	100%
frame.c	1149	6	1%
hashtable.c	718	76	11%
loader.c	2957	381	13%
nativeCore.c	1287	12	1%
pool.c	437	46	11%
total	*23759*	*915*	*4%*

This increase consists of enhancements to: identify and process trusted classes (loader); manage certificates (hashtable), privileges (collector), and other runtime structures (crypto); perform privilege checks (class, nativeCore, and pool); and handle exceptions (frame).

Additionally, a stand alone tool was written to annotate classfiles with properly constructed trust attributes.

8.2 Capabilities

Support for capabilities pushed the total code size increase to 6%, to a total of 24272 lines. The details of these changes are shown in Table 5.

Table 5. Code size overhead of Capabilites

File	Additional LoC for Capabilities
class.c	243
frame.c	4
loader.c	266

This further increase consists of enhancements to: process the capability-based privileges (loader); handle domain intersection and resource access checks (class); and handle exceptions (frame).

8.3 Untrusted Classes

Support for untrusted classes bumped the total code size increase to 7%, to a total of 24538 lines. The details of these changes are shown in Table 6.

Table 6. Code size overhead of Untrusted Classes

File	Additional LoC for Untrusted Classes
loader.c	171
pool.c	95

This final increase consists of enhancements to: process the untrusted class privileges (loader); and handle access checks (pool). The details of these changes are shown in Table 6.

9 Status and Further Work

As described above, a prototype version of the SKVM has been implemented based on the standard KVM. There are other, more recent CLDC implementations that are potentially better platforms on which to base the SKVM (such as [21]). Further work with the SKVM will be done using one of these implementations. Such work will include various size and performance measurements (such as the space overhead for certificates and the runtime overhead for capability support).

The next major step in demonstrating the feasibility and value of the SKVM will be porting it to a cryptographic module. This task may expose platform and deployment issues. It will also enable real-world testing of SKVM applications.

The subsequent step will be the FIPS certification of the SKVM. This effort will require precise definition of the operation and implementation of the SKVM.

A possible enhancement involves untrusted classes. If it is determined that they are truly useful on a secure platform, they can be completely isolated in their

own execution environment [22]. They could be given their own heap, execution stack, and resource limits.

References

1. *Connected, Limited Device Configuration, Specification Version 1.1*; Sun Microsystems, May 2002; `http://java.sun.com/products/cldc`.
2. *Java Card Technology for Smart Cards*; Zhiqun Chen; Addison-Wesley; June 2000.
3. *CLDC Technology Compatibility Kit version 1.0a User's Guide*; Sun Microsystems; February 2001.
4. Information on the KVM can be found at `http://java.sun.com/products/cldc`.
5. *Inside Java 2 Platform Security*; Li Gong; Addison-Wesley; October 1999.
6. *Mobile Information Device Profile for Java 2 Micro Edition, Version 2.0*; Java Community Process, November 2002; `http://java.sun.com/products/midp`.
7. "MIDP 2.0 Security Enhancements"; Otto Kolsi, Teemupekka Virtanen; *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*; January 2004.
8. "Understanding MIDP 2.0's Security Architecture"; Jonathan Knudsen; February 2003; `http://developers.sun.com/techtopics/mobility/midp/articles/permissions/`
9. "MAPbox: Using Parameterized Behavior CLasses to Confine Untrusted Applications"; Anurag Acharya, Mandar Raje; *Proceedings of the 9th USENIX Security Symposium*; August 2000.
10. "A Flexible Containment Mechanism for Executing Untrusted Code"; David S. Peterson, Matt Bishop, Raju Pandey; *Proceedings of the 11th USENIX Security Symposium*; August 2002.
11. *Security Requirements for Cryptographic Modules*; NIST FIPS PUB 140-2, 25 May 2001.
12. *Building the IBM 4758 Secure Coprocessor*; Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, Sean W. Smith, Steve Weingart; IEEE Computer; October 2001; pp. **57-66**.
13. *Secrets and Lies*; Bruce Schneier; John Wiley and Sons, 2000.
14. *Applied Cryptography, Second Edition*; Bruce Schneier; John Wiley and Sons; 1996.
15. *Security Engineering: A Guide to Building Dependable Distributed Systems*; Ross Anderson; John Wiley and Sons; 2001.
16. *The Java Virtual Machine Specification, Second Edition*; Tim Lindholm, Frank Yellin; Addison-Wesley, April 1999.
17. *The Java Language Specification, Second Edition*; James Gosling, Bill Joy, Guy Steele, Gilad Bracha; Addison-Wesley, June 2000.
18. *Securing Java*; Gary McGraw, Edward W. Felten; John Wiley and Sons, 1999.
19. *KVM Debug Wire Protocol (KDWP), Version 1.0*; Sun Microsystems; 26 February 2001.
20. "Programming Semantics for Multiprogrammed Computations"; Jack Dennis, Earl Van Horn; *Communications of the ACM*; March 1966; pp. **143-155**.
21. "A Java Virtual Machine Architecture for Very Small Devices"; Nik Shaylor, Doug Simon, Bill Bush; *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, June 2003, pp. **34-41**.
22. "A Secure Java Virtual Machine"; Leendert van Doorn; *Proceedings of the 9the USENIX Security Symposium*; August 2000.

ESC/Java2: Uniting ESC/Java and JML

Progress and Issues in Building and Using ESC/Java2, Including a Case Study Involving the Use of the Tool to Verify Portions of an Internet Voting Tally System

David R. Cok[1] and Joseph R. Kiniry[2]

[1] B65 MC01816
Eastman Kodak R & D Laboratories
Rochester, NY 14650-1816, USA
cok@frontiernet.net
[2] Department of Computer Science, University College Dublin,
Belfield, Dublin 4, Ireland*
kiniry@acm.org

Abstract. The ESC/Java tool was a lauded advance in effective static checking of realistic Java programs, but has become out-of-date with respect to Java and the Java Modeling Language (JML). The ESC/Java2 project, whose progress is described in this paper, builds on the final release of ESC/Java from DEC/SRC in several ways. It parses all of JML, thus can be used with the growing body of JML-annotated Java code; it has additional static checking capabilities; and it has been designed, constructed, and documented in such a way as to improve the tool's usability to both users and researchers. It is intended that ESC/Java2 be used for further research in, and larger-scale case studies of, annotation and verification, and for studies in programmer productivity that may result from its integration with other tools that work with JML and Java. The initial results of the first major use of ESC/Java2, that of the verification of parts of the tally subsystem of the Dutch Internet voting system are presented as well.

1 Introduction

The ESC/Java tool developed at DEC/SRC was a pioneering tool in the application of static program analysis and verification technology to annotated Java programs [13]. It was a successor to the ESC/Modula-3 tool [22], using many of the same ideas, but targeting a "mainstream" programming language. ESC/Java operates on full Java programs, not on special-purpose languages. It acts modularly on each method (as opposed to whole-program analysis), keeping the complexity low for industrial-sized programs, but requiring annotations on methods that are used by other methods. The program source and its specifications are translated into verification conditions; these are passed to a theorem

* Formerly with the Security of Systems Group at the University of Nijmegen.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 108–128, 2005.

prover, which in turn either verifies that no problems are found or generates a counterexample indicating a potential bug. The tool and its built-in prover operate automatically with reasonable performance and need only program annotations against which to check a program's source code. The annotations needed are easily read, written and understood by those familiar with Java and are partially consistent with the syntax and semantics of the separate Java Modeling Language (JML) project [1, 19]. Consequently, the original ESC/Java (hereafter called ESC/Java) was a research success and was also successfully used by other groups for a variety of case studies (e.g., [16, 17]).

Its long-term utility, however, was lessened by a number of factors. First, as companies were bought and sold and research groups disbanded, there was no continuing development or support of ESC/Java, making it less useful as time went by. As a result of these marketplace changes, the tool was untouched for over two years and its source code was not available.

The problem of lack of support was further compounded because its match to JML was not complete, and JML continued to evolve as research on the needs of annotations for program checking advanced. This unavoidable divergence of specification languages made writing, verifying, and maintaining specifications of non-trivial APIs troublesome (as discussed in Section 5).

Additionally, JML has grown significantly in popularity. The activities of several groups [1, 3, 19, 27, 28] generated a number of tools that work with JML. Thus, many new research tools worked well with "modern" JML, but ESC/Java did not.

Finally, some of the deficiencies of the annotation language used by ESC/Java reduced the overall usability of the tool. For example, frame conditions were not checked, but errors in frame conditions could cause the prover to reach incorrect conclusions. Also, the annotation language lacked the ability to use methods in annotations, limiting the annotations to statements only about low-level representations.

The initial positive experience of ESC/Java inspired a vision for an industrial-strength tool that would also be useful for ongoing research in annotation and verification. Thus, when the source code for ESC/Java was made available, the authors of this paper began the ESC/Java2 project.

This effort has the following goals: (1) to make the source consistent with the current version of Java; (2) to fully parse the current version of JML and Java; (3) to check as much of the JML annotation language as feasible, consistent with the original engineering goals of ESC/Java (usability at the expense of full completeness and soundness); (4) to package the tool in a way that enables easy application in a variety of environments, consistent with the licensing provisions of the source code release; and (5) as a long-term goal, and if appropriate, to update the related tools that use the same code base (Calvin, RCC, and Houdini [12]) and to integrate with other JML-based tools. This integration will enable testing the tool's utility in improving programmer productivity on significant bodies of Java source; the tool will also serve as a basis for research in unexplored aspects of annotation and static program analysis.

We have released over seven alpha versions of ESC/Java2. The latest version is available on the web[1] and we encourage experimentation and feedback. The source code is available (and additional contributors are welcome) and is subject to fairly open licensing provisions. The discussion below of various features of JML and ESC/Java2 is necessarily brief; more detail is available in the implementation notes that are part of an ESC/Java2 release.

The subsequent sections will discuss the most significant changes in creating ESC/Java2, the extensions to static checking, the backwards incompatibilities introduced, unresolved semantic issues in JML, and the direction of the ongoing work in this project. Also discussed is ESC/Java2's first serious use: the verification of portions of the tally subsystem of the Dutch Internet voting system.

Appendices list the details of the enhancements to ESC/Java and those features of JML that are not yet implemented in ESC/Java2. We fully acknowledge that the on-going work described here builds on two substantial prior efforts: the definition of the Java Modeling Language and the production of ESC/Java and the Simplify prover in the first place.

2 JML Example

The Java Modeling language is described in detail in several other publications ([19] and various papers listed at [1]), so here we will give just one example showing some of the syntax. The class in Fig. 1 uses an array to implement a List. A few methods with partial specifications are shown. They demonstrate the following features of JML:

- JML annotations are contained in comments beginning with `//@` or `/*@`.
- `model import` statements declare classes imported for use in annotations.
- `spec_public` indicates that the field named *seq* has public visibility in specifications.
- The `invariant` states that after construction *seq* is never null and is an array with `Object`s as elements.
- The `requires` keyword states a precondition for the *reverse* method, namely its argument is presumed to be non-null.
- The `modifies` keyword states a frame condition for the *reverse* method, namely that the only fields that are assigned during its execution are the elements of the `out` argument.
- The `signals` keyword states a postcondition for the *reverse* method that must hold if an exception is thrown, in this case that it never throws a `NullPointerException`.
- The `ensures` keyword states a postcondition for the *reverse* method that must hold if the method terminates normally.
- The `model` declaration declares a public field used in specifications, typically as an abstract representation of the class. In this case, the class represents a `List`.

[1] http://www.niii.kun.nl/ita/sos/projects/escframe.html

```
//@ model import java.util.List;
//@ model import java.util.ArrayList;
public class Example {
  /*@ spec_public */ private Object[] seq;
                       //@ in list;
                       //@ maps seq[*] \into list;
  //@ invariant seq != null && \elemtype(\typeof(seq)) == \type(Object);

  //@ requires out != null;
  //@ modifies out[*];
  //@ signals (NullPointerException) false;
  //@ ensures seq.length > 0 ==> out[0] == seq[seq.length-1];
  public void reverse(Object[] out) {
    int i = 0;
    int j = seq.length;
    while (i < seq.length) out[i++] = seq[--j];
  }

  //@ public model List list;
  //@ private represents list <- toList(seq);

  /*@ requires input != null;
    @ ensures \result != null;
    @ pure
    @ private model List toList(Object[] input) {
    @   List list = new ArrayList(input.length);
    @   for (int i=0; i<input.length; ++i) list.add(input[i]);
    @   return list;
    @ }
    @*/

  //@ requires i >= 0 && i < length();
  //@ modifies list;
  public void insert(int i, Object o) { seq[i] = o; }

  //@ private normal_behavior
  //@   ensures \result == seq.length;
  //@ pure
  public int length() { return seq.length; }
}
```

Fig. 1. A List class with a partial specification

- The `represents` statement indicates the relationship between the value of the model field and the implementation.
- The next set of declarations constitute a model method declaration and its specifications; a model method is only used in annotations and need not have an implementation.

- The `modifies` clause on the *insert* method indicates that it may modify the value of the model field *list* or any field in its *datagroup*; the `in` and `map` annotations on the declaration of *seq* stipulate that the *seq* field and its array elements are in the *list* datagroup.
- The `pure` modifier on the *length* method indicates that that method has no side effects (it does not assign to any fields).

This class will compile with javac and will pass all the checks of the jml checker. If it is subjected to the ESC/Java2 tool described in this paper, three warnings are produced, correctly pointing out three potential problems with this code:

- The default constructor does not set the value of `seq` to a non null array as required by the invariant.
- The assignment to `out[i++]` on line 16 is problematic because the index may be too large for the array; this is fixed by stating that the length of `out` must be equal to the length of `seq`.
- An additional warning on that line indicates that the type of the `out` array may possibly not allow assignments of Object references to its elements.

3 Changes to DEC/SRC ESC/Java

Creating ESC/Java2 required a number of changes to the ESC/Java tool. Here we present the most significant of these.

3.1 Java 1.4

The original work was performed from 1998 to 2000, and Java has evolved since then[2]. The addition required by Java 1.4 is support for the Java `assert` statement.

JML itself contains a similar assert statement. Hence, the user may make a choice between two behaviors. A Java assert statement may be interpreted simply as another language feature whose behavior is to be modeled. The corresponding behavior is to raise an `AssertionError` exception under appropriate circumstances. Alternatively, a Java assert statement may be interpreted as a JML assert statement. In this case, the static checker will report a warning if the assertion predicate cannot be established. Both alternatives are available through user-specified options.

3.2 Current JML

The Java Modeling Language is a research project in itself; hence the JML syntax and semantics are evolving and are somewhat of a moving target (and

[2] In fact, Java 1.5 went beta recently. No work has begun on parsing or statically checking Java 1.5 code. Interested parties are welcome to contact the authors with regard to this topic.

there is as yet no complete reference manual). However, the core language is reasonably stable. The following are key additions that have been implemented; other changes that relate primarily to parsing and JML updates are listed in the Appendix:

- inheritance of annotations and of `non_null` modifiers that is consistent with the behavioral inheritance of JML;
- support for datagroups and `in` and `maps` clauses, which provides a sound framework for reasoning about the combination of frame conditions and subtyping;
- model import statements and model fields, routines, and types, which allow abstraction and modularity in writing specifications;
- enlarging the use and correcting the handling of scope of ghost fields, so that the syntactic behavior of annotation fields matches that of Java and other JML tools.

In addition, all of the differences between JML and ESC/Java noted in the JML Reference Manual have been resolved[3].

3.3 New Verification Checks

Though all of JML is parsed, not all of it is currently checked. ESC/Java concentrated on checking for possible unexpected exceptions arising from conditions such as null pointers or out-of-bounds array indices, since these did not need annotations to be found; annotations were used, however, to state conditions on method arguments or class fields that would preclude such errors. Thus ESC/Java was capable of checking the pre- and post-conditions of methods as well. However, these could only be expressed at a low-level given the limitations of the ESC/Java input language.

The expanded capabilities of ESC/Java2 allow more thorough checking at a higher level of abstraction. This has required only minor changes in the background axioms used by the theorem prover (mostly regarding primitive types, though additions to handle the semantics of String objects are needed). Most of the changes are implemented by the appropriate translation of JML features into the theorem prover's input logic. The space available in this paper permits only a summary of the embedding of the above into the underlying ESC/Java logic[4].

Static checking of the following features has been added to that performed by ESC/Java.

The constraint and initially clauses. These two clauses are variations on the more common `invariant` clause. They apply to the whole class. A constraint states a condition that must hold between the pre-state and the post-state of every method of a class. For example,

[3] The tools still differ in (a) the search order for refinement files on the classpath and (b) which methods may be declared as helper methods.

[4] Subsequent papers are planned that will describe these embeddings in more detail.

```
constraint maxSize == \old(maxSize);
```

states that `maxSize` is not changed by any method of the class. It is implemented by adding the predicate as a postcondition of every (non-helper) method in the type (and its derived types).

Similarly, `initially` states a condition that must hold of every object after construction. It is implemented by adding its predicate as a postcondition of every (non-helper) constructor of the type (but not of its derived types).

The `\not_modified` expression. The `not_modified` construct is a way of saying, within a postcondition, that a particular expression has the same value in the pre-state and the post-state. That is,

$$\texttt{\textbackslash not\_modified(x+y)} \equiv \texttt{((x+y) == \textbackslash old(x+y))} .$$

Uses of the expression in postconditions are expanded inline according to this definition.

Checking of datagroups and frame conditions. JML contains syntax to define datagroups [24]. With datagroups, the items in an `assignable` clause may represent sets of program locations, and those sets may be extended by subtypes. Each specification case of a routine may be guarded by a precondition and may specify the set of store locations that may be assigned to.

There are a number of cases to be considered in a full implementation. We will discuss just one here: an assignment statement that has a left-hand side of *expr.field*. For this to be a legal assignment with respect to the specifications, either (a) the *expr* must evaluate to an object that has been allocated since the beginning of the execution of the method, or (b) it must be the case that for every specification case of the method containing the assignment for which the precondition is true (in the pre-state) there is at least one store location in the list of assignable locations that matches *expr.field*. To match, the field names must be the same and the *expr* values must evaluate to the same object. The matching is complicated by the variety of syntax (e.g. *expr*`.*` matches any field of *expr*) and by the fact that a given field designation may have an accompanying datagroup and the match may be to any element of the datagroup. All of this syntax is parsed, and checks are implemented in the logic except where induction is needed to handle recursive definitions.

Recursive definitions of frame conditions (arising from recursive structures such as linked lists and trees) are indeed the most substantial complication in checking datagroups. As an example, consider the datagroup of all of the 'next' fields of a linked list. ESC/Java2 currently deals with this by unrolling the recursion to a fixed depth; since in ESC/Java loops are also unrolled to a fixed number of iterations, this solution handles common cases of iterating over recursive structures.

Annotations containing method and constructor calls. JML, but not ESC/Java, allows pure method and constructor calls (that is, methods and constructors without side-effects) to be used in annotations. This allows both a degree of abstraction and more readable and writable specifications.

ESC/Java2 supports the JML syntax and also performs some static checking. The underlying prover, Simplify, does support function definitions and reasoning with functions. But, as is the rule in first-order logic, the result of a function in Simplify depends only on its arguments and not on hidden arguments or on global structures referenced by the arguments. Consequently there is a mismatch between the concept of a pure method in Java and the concept of a function in the prover. However, a moderate degree of checking can be performed without resorting to a full state-based translation and logic if we (a) identify some methods as functions, where possible, (b) include the current state of the heap as an additional uninterpreted parameter, and (c) incorporate the specifications of the called method as additional axioms.

Dynamic allocations of objects using constructors are simply static method calls that return new objects and are treated in the same way as other method calls. The logic includes axioms that ensure that a newly allocated object is distinct (reference values are unequal) from any previously allocated object. Dynamic allocations of arrays are translated into first-order logic as functions without difficulty, as they were in ESC/Java.

model fields and represents clauses. The combination of `represents` clauses and model fields provides a substantial benefit in abstraction, especially since the representations may be provided by a subtype [8]. Simple representations can be implemented in ESC/Java2 by inlining the representation wherever the model field is used in an annotation. However, that proves not to be workable in larger systems. Instead, we translate instances of model fields as functions of the object that owns them and the global state (because model fields can depend upon fields in other, non-owner, objects). This allows a useful degree of reasoning when combined with the class invariants that describe the behavior of the model fields.

The Simplify theorem prover used by both ESC/Java and ESC/Java2 remains unchanged, except for being compiled for a new platform (Apple's OS X). It is written in Modula-3 and consequently requires compilation for each supported platform. Although the prover has definite limitations, as pointed out below, revising it would be a significant project in its own right.

3.4 Backwards Incompatibilities

The ESC/Java specification language and JML arose separately; there was some initial but incomplete work to unify the two [20]. The ESC/Java2 project intends to have the tool reflect JML as precisely as reasonable. In some cases, discussion about differences resulted in changes to JML. In a few cases, some backwards incompatibilities in ESC/Java were introduced. The principal incompatibilities are these:

- The semantics of inheritance of specification clauses and of `non_null` modifiers was modified to match that defined by JML, since the work on JML resulted in an interpretation consistent with behavioral subtyping. JML has a standalone `also` keyword that indicates there are inherited explicit or

implicit specifications; its interpretation of specification inheritance is consistent with behavioral subtyping. By contrast, ESC/Java's use of inherited specifications had limitations and was a known source of soundness problems [23]. See the section titled "Inheritance and `non_null`" in the ESC/Java2 Implementation Notes for more details [10].
- The specification `modifies \everything` is now the default frame axiom.
- The syntax and semantics of `initially`, `readable_if` and `monitored_by` have changed.
- ESC/Java2 forbids bodies of (non-model) routines to be present in non-Java specification files.

4 Unresolved Semantic Issues

The work on ESC/Java2 has been useful in exposing and resolving semantic issues in JML. Since ESC/Java2 is built on a different source code base than other JML tools, differences of interpretation in both syntax and semantics arise on occasion. These are generally resolved and documented via mailing list discussions[5] by interested parties. There are, however, still unresolved issues, most of which are the subject of ongoing research.

- *pure routines*: It is convenient and modular to use model and Java methods within specifications (model methods are methods defined for annotations only and not part of the Java source, such as the *toList* method in Fig. 1). The semantics of such use is clearer and simpler if such routines are *pure*, that is, they do not have side-effects[6]. This is important when evaluating annotations during execution, since the checking of specifications should not affect the operation of the program being checked. Side-effects also complicate static reasoning. However, some side-effects are always present, such as changes to the stack or heap or external effects such as the passage of time. Some are often overlooked but can be consequential, such as locking a monitor. Others the programmer may see as innocuous, benevolent side-effects, such as maintaining a private cached value or logging debugging information in an output file. An interpretation of the combination of purity and benevolent (or ignorable) side-effects that is suitable for both static and run-time checking and is convenient and intuitive for users is not yet available. (See also the discussion of purity checking in [18].)
- *exceptions in pure expressions*: The expressions used in annotations must not have side-effects, but they may still throw exceptions. In that case the result is ill-defined. A semantics that is suitable for both run-time checking and static verification needs to be established.

[5] See `jmlspecs-interest@lists.sourceforge.net`, `jmlspecs-developers@lists.sourceforge.net`, and `jmlspecs-escjava@lists.sourceforge.net` or the corresponding archives at `http://sourceforge.net/projects/jmlspecs`

[6] Non-pure methods may be used within annotations in *model programs*, which are not discussed in this paper.

- *initialization*: The authors are not aware of any published work on specifying the initialization of classes and objects in the context of JML; initial work formalizing `\not_initialized` was only recently completed for the Loop tool. This task includes providing syntax and semantics for Java initialization blocks, JML's `initializer` and `static_initializer` keywords, and formalizing the rules about order of initialization of classes and object fields in Java.
- *datagroups*: The `in` and `maps` clauses and the datagroup syntax are designed to allow the specification of frame conditions in a sound way that is extensible by derived classes. We do not yet have experience with the interaction among datagroups, the syntax for designating store locations, and either reasoning about recursive data structures or checking them at run-time.
- *unbounded arithmetic*: Chalin [7] has proposed syntax and semantics to enable specifiers to utilize unbounded arithmetic in a safe way within annotations. Tool support and experience with these concepts is in progress. Axioms and proof procedures will be needed to support this work in static checkers.

There are other outstanding but less significant issues concerning helper annotations, model programs and the `weakly`, `hence_by`, `measured_by`, `accessible` and `callable` clauses.

5 Usage Experience to Date

The SoS group at the University of Nijmegen, along with other members of the European VerifiCard Project[7], has used ESC/Java for several projects. For example, Hubbers, Oostdijk, and Poll have performed verifications of Smart Card applets using several tools, including ESC/Java [17]. Hubbers has also taken initial steps integrating several JML-based tools [16].

These and other VerifiCard projects relied upon the specifications of the Java Card 2.1.1 API written and verified by Poll, Meijer, and others [26]. This specification originally came in two forms: one "heavyweight" specification that used JML models, heavyweight contract specifications, and refinements, and another "lightweight" specification that was meant to be used with ESC/Java and other verification tools like Jack, Krakatoa, and the Loop tool [2, 4, 25].

Writing, verifying, and maintaining these two specifications was a troublesome experience. Because of limitations of various tools which depended upon the specifications, several alternate forms of specifications were required. Additionally, it was sometimes the case that the alternate forms were neither equivalent nor had obvious logical relationships among them.

This experience was one of the motivators for the SoS group's support of this work on ESC/Java2. Now that multiple tools are available that fully cover the JML language, the incidence of specification reuse is rising and painful maintenance issues are becoming a thing of the past. As a result, early evidence for the success of this transition is beginning to appear.

[7] http://www.verificard.org/

5.1 Transitional Verifications

First, the specifications of a small case study [5] were updated and re-verified by one of this paper's authors (Kiniry) using ESC/Java2. The original work depended upon "lightweight" JML specifications of core Java Card classes and the verification was performed with ESC/Java and the Loop tool. The re-verification effort used the full "heavyweight" Java and Java Card specifications and was accomplished in a single afternoon by an ESC/Java expert.

Second, several members of the SoS group are contributing to updating the "heavyweight" JML specifications of the Java Card API. As a part of this work, the Gemplus Electronic Purse case study, which has been verified partially with ESC/Java [6] and partially with the Loop tool [5], is being re-verified completely with ESC/Java2 using "heavyweight" specifications.

Finally, recent attempts at verifying highly complex Java code examples written by Jan Bergstra and originally used as stress-tests for the Loop tool have been encouraging. Methods that originally took a significant amount of interactive effort to verify in PVS are now automatically verified in ESC/Java2, much to the surprise of some of the Loop tool authors. This work has caused some re-evaluation of the balance between interactive and automated theorem proving in the SoS group.

5.2 Verification of an Electronic Voting Subsystem

The first major partial verification using ESC/Java2 took place in early 2004.

The Dutch Parliament decided in 2003 to construct an Internet-based remote voting system for use by Dutch expatriates. The SoS group was part of an expert review panel for the system and also performed a black-box network and system security evaluation of this system in late 2003. A recommendation of the panel was that a third party should construct a redundant tally system. Such a second system would ensure a double-check of the election count with an independent system. It was also thought that such an external implementation would provide some third-party review of the original work, as the new implementation would depend entirely upon system design documentation and data artifacts (e.g. candidate and vote files); no source code would be shared, or even seen, by the team implementing the redundant system.

The SoS group bid on the construction of this new system, emphasizing the fact that they would use formal methods (specifically, JML and ESC/Java2) to specify, test, and verify the tally system. The bid was successful; as a result the SoS group was contracted to write the tally system.

The most challenging aspect of the contract was *not* the use of formal methods. Instead, it was the strict *time requirements* of the contract, as the system was to be used in the upcoming European elections. In particular, the SoS group was asked to construct the vote counting system (henceforth called the KOA system) in approximately *four weeks*, with only three developers.

Development Methodology. To approach this problem, the three developers (Dr. Engelbert Hubbers, Dr. Martijn Oostdijk, and the second author) parti-

tioned the system into three subcomponents: file I/O, graphical I/O, and core data structures and algorithms. It was decided that, due to the challenges inherent in full system verification and the restricted time allotted to the project, while all subsystems would be annotated with JML, only the third "core" subsystem (Dr. Kiniry's responsibility) would be fully elaborated in specification.

Additionally, ESC/Java2 would only be used on the core subsystem. In the allotted time a "best-effort" verification would be attempted, in addition to all other standard software engineering practices. This approach is a standard strategy for lightweight use of formal methods [9].

Table 1. KOA System Summary

	File I/O	**Graphical I/O**	**Core**
classes	8	13	6
methods	154	200	83
NCSS	837	1,599	395
Specs	446	172	529
Specs:NCSS	1:2	1:10	5:4

Table 1 summarizes the size (in number of classes and methods), complexity (non-comment size of source, or NCSS for short), and specification coverage of the three subsystems, as measured with the JavaNCSS tool version 20.40 during the week of 24 May, 2004. Assertions were counted by simply counting the number of uses of appropriate core specification keywords (requires, ensures, invariant, non_null, in, set, and modifies).

The size of the code and specifications gives a strong indication of the complexity of the verification effort. Longer methods take significantly more time to specify and verify than short ones. Classes with many methods, on the other hand, do not necessarily take much more time to deal with than shorter classes, as effort is coupled to the complexity of the methods, their specs, and the class invariants (e.g., many short, simple methods are trivial to verify, while one long method might take days).

There is very little inheritance in this system. Visual components all inherit from a top-level `Task` class which implements all state changes in response to external input, and the I/O classes inherit from an `AbstractObjectReader`, an Apache licensed helper class. Other than that, all classes have no parent (beside `Object`).

Because there is little inheritance and we adopt a closed-system view on the vote tally system (no classloading is permitted), ESC/Java2's weak strong support for specifying and reasoning about dynamic binding is not an issue.

Specification Coverage and Methodology. Unsurprisingly, the GUI portion of KOA is the largest subsystem with the lightest specification coverage, having approximately 1 annotation for every 10 lines of code. The focus of the GUI

subsystem specification is a finite state machine that represents the state of the GUI. The state of the KOA application is tightly coupled to this GUI state machine as the vote counting process is highly serialized.

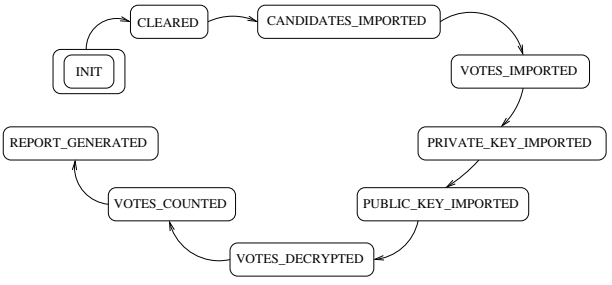


Fig. 2. KOA State Diagram

Figure 2 contains a diagram that summarizes this state machine. The state of the system is represented in a (spec_public) field "`state`" of the main class of the application. The state machine is formally modeled using the standard mechanisms developed in the past by the SoS Group [15].

The file I/O subsystem exhibits better specification coverage, much of which focuses on contracts to ensure that data-structures in the core subsystem are properly constructed according to the contents of input files.

The core subsystem understandably has the highest specification coverage, at over one line of specification for every line of code. This part of the system was designed by contract, and a small-step development process was used throughout (i.e., every time a single line of the specification or the code was changed ESC/Java2 was re-executed). Contractual specifications (e.g., requires/ensures-style and invariants) accounted for the vast majority of the specification; assertions and invariants were only used to assist the verification process.

The verification of the key properties of the system, particularly the property of having a correct tally of votes, are directly tied to the overall state of the system using invariants of the form

```
invariant (state >= <STATE>) ==> (state-field != null);
```

where the states of the system form a total order. Such an invariant says that, if the state of the system is at least `<STATE>`, then the appropriate representation for that state, captured in the `state-field`'s datatype is well-formed. This is a strong claim because if `state-field` is non-null, then not only is it initialized, but all of its invariants hold.

At this time, verification coverage of the core subsystem is good, but not 100%. Approximately 10% of the core methods (8 methods) are unverified due to issues with ESC/Java's Simplify theorem prover (e.g., either the prover does not terminate or terminates abnormally, as discussed below). Another 31% of the core methods (26 methods) have postconditions that cannot be verified, typically due to completeness issues discussed above, and 12% of the methods (10 methods) fail to verify due to invariant issues, most of which are due to

suspected inconsistencies in the specifications of the core Java class libraries or JML model classes. The remaining 47% (39 methods) of the core verifies completely.

Since 100% verification coverage was not possible in the timeframe of the project, and to ensure the KOA application is of the highest quality level possible, a large number unit tests were generated with the *jmlunit* tool for all core classes. A total of nearly 8,000 unit tests were generated, focusing on key values of the various datatypes and their dependent base types. These tests cover 100% of the core code and are 100% successful.

Impressions of ESC/Java2. ESC/Java2 made a very positive impression on the KOA developers. Its increased capabilities as compared to ESC/Java, particularly with regards to handling the full JML language, the ability to reason with models and specifications with pure methods, are very impressive. And, while the tool is still classified as an "alpha" release, we found it to be quite robust (perhaps unsurprising given its history, the use of JML and ESC/Java2 in and on its own source code, and the fact that it is passed through seven alpha releases thus far). But there are still a number of issues with ESC/Java2 and JML that were highlighted by the KOA verification effort.

The primary issues that arose include:

- *String semantics in ESC/Java2 are incomplete.* In particular, one cannot reason effectively about String concatenation or equality. While Java Strings are certainly a non-trivial type, they are effectively a pseudo-base type because of their widespread use. Thus, it is vital that this issue be addressed as soon as possible.
- *Issues with reasoning about "representation-less" model variables.* If a class declares a model variable but provides no statement about how that model relates to the implementation of the class (using a `represents` clause or similar), then ESC/Java2 cannot verify assertions that use the model. We believe that a representation-less model variable is equivalent to a ghost variable since it is being used as a specification-only variable in an API spec. Thus, by replacing problematic model variables with ghost variables in API specifications we can successfully perform verifications using the APIs. This problem indicates either a problem with the design and/or use of model variables in JML, or an implementation issue with ESC/Java2.
- *Inconsistencies and ambiguities in the specification of core APIs, particularly classes in the `java.lang` and `java.util` packages and JML model classes.* This is the first large-scale verification effort using "full" JML specifications of the core JDK. These "full" specifications are much more complete and complex than those that were used with ESC/Java. As these core specifications have never been formally analyzed for consistency or completeness, it is not surprising that the KOA verification effort had problems with their use. It is expected that over time, with more use by a range of JML-compatible tools the core specifications will become more consistent and complete to the benefit of all JML tool users.

- *Completeness issues with first-order predicates.* First-order predicates are expressed with the `forall` and `exists` constructs in JML. Only some of these predicates can be used and/or verified in JML-based tools, including ESC/Java2. Unfortunately, many of the most interesting invariants in non-trivial systems can only be expressed using such constructs. Thus, more focus needs to be put on understanding and reasoning about such assertions.
- *Aliasing issues and specification convenience constructs for such.* As usual, reasoning about reference types and avoiding aliasing was one of the key issues in verifying the KOA application. For example, frequently we wished to say that the elements of a set of references were pairwise unequal. The only way to state this in JML today is quite cumbersome, thus the introduction of a new specification construct for such seems warranted.
- *The Simplify prover backend and alternate provers.* Simplify is a relatively robust automatic first-order prover, especially given its age and the fact that no one has supported it in many years. Unfortunately, Simplify sometimes fails catastrophically in one of two ways: it crashes due to an internal exception or assertion failure, or it (rarely) consumes as much memory as possible and halts. Neither of these situations is reasonable, of course, but as there is no support for Simplify, the problems indicated by these affects are rarely correctable. It is our intention to initially augment, then eventually replace, Simplify with an alternative modern, supported prover.

6 Ongoing Work

The work on ESC/Java2 is continuing on a number of fronts.

Language issues. Two obvious and related ongoing tasks are the completion of additional features of JML, accompanied in some cases by additional research to clarify the semantics and usability of outstanding features of JML. Usage of JML is now broad enough that an accompanying formal reference document would be valuable. As tools such as ESC/Java2 become more widely used, users will also appreciate attention to performance, to the clarity of errors and warnings, and to the overall user experience such tools provide.

Case studies. The current implementation supports the static checking of a stable core of JML. With this initial implementation of frame condition checking, of model fields, represents clauses and use of routine calls in annotations, ESC/Java2 can now be used on complex and abstract specifications of larger bodies of software. Consequently, there is a considerable need for good experimental usage studies that confirm that this core of JML is useful in annotations, and that the operation of ESC/Java2 (and Simplify) on that core is correct and valuable.

Verification logic. The logic into which Java and JML are embedded in both ESC/Java and ESC/Java2, by design and admission of the original authors, neither identifies all potential errors (e.g., because not all aspects of Java are

modeled in the logic) nor avoids all false alarms (e.g., because of limitations in the prover). This was the result of an engineering judgment in favor of performance and usability. Research studying expanded and larger use cases may show whether this design decision is generally useful in practical static checking or whether a fuller and more complicated state-based logic is required for significant results to be obtained.

A related issue is the balance between automated and manual proof construction. Use of verification logics will likely be limited to narrow specialties as long as proof construction is a major component of the overall programming task. Thus, automation is essential, though it is expected that full automation is infeasible. The degree of automation achievable will continue to be a research question. However, we believe that broad adoption of automated tools for program checking will require that users only need interact at high levels of proof construction.

User feedback. The purpose of using theorem provers for static analysis, runtime assertion checking, or model checking is to find errors and thereby improve the correctness of the resulting software. Thus, the orientation of a tool must be towards effectively finding *and interpreting* examples of incorrect behavior. A complaint (e.g., [14]) in using such technologies is that it is difficult to determine a root cause from the counterexample information provided by the tool, whether it is a failed proof or an invalid test or execution history. The ESC/Java project implemented some work towards appropriately pruning and interpreting counterexample and trace information [21], but there remains room for improvement. Machine reading of the Simplify output coupled with other tools is also a means to easier interpretation [11].

Tool integration. Finally, though not part of this specific project, an integration of tools that support JML would be beneficial for programmer productivity. A productive programmer's working environment for a large-scale project that uses these tools would need them to be integrated in a way that they seamlessly communicate with one another. A programmer using the tools would naturally move among the various tasks of designing, writing, testing, annotating, verifying and debugging, all the while reading, writing and checking specifications. Design, specifications and code might all be built up incrementally. Thus, the tools would need to be integrated in a way that allows efficient and iterative behavior.

7 Conclusion

The progress and case studies described above have shown that ESC/Java2 is ready for serious evaluation and use, even in its early "alpha" releases. Our ability to verify large portions of a critical, public system in a very short time frame is a strong statement about the state of the tool and the underlying theory of extended static checking.

Additional evidence comes from several groups around the world that are using ESC/Java2 for instruction and research. We are aware of over a half dozen

groups that are using ESC/Java2 for new research in verification, and nearly ten courses are using it for instruction in software engineering, verification, pedagogical instruction of Java, and grading. We continue to see growing interest in ESC/Java2, verification, and extended static checking in general.

One can observe this work in tool creation and evaluation from a number of perspectives. Certainly such work creates working prototypes that test in practice theories of programming and specification language semantics. It also exercises and validates ideas in automated logical reasoning. We prefer to use the viewpoint of programming productivity, particularly given the industrial working environment of the first author. In that context we observe the existence and general use across multiple research groups of the combination of various tools that support using JML with Java programs; this suggests to us that the syntax and semantics of the core of JML are sufficiently useful and natural to provide a basis for future wider use. With respect to logical reasoning, a useful degree of automation is achievable in at least some aspects of static checking tasks; removing the details of proof construction from a programmer's tasks is essential to larger scale acceptance of such tools.

However, the surrounding issues are as relevant to programmer acceptance and productivity. Tools must have intuitive and unsurprising behavior. They must be efficient in elapsed run-time, but also in the time needed to interpret and act upon the results. They must integrate well with other tools of the same family and with commonly used programmer's working environments.

There is progress on enough of the above vectors that one might well be optimistic about the eventual success of the enterprise as a whole. After all, the goal need not be fully automated verification of an arbitrary computer program. Reflect that a computer-produced proof of a mathematical conjecture that cannot be understood at least in its broad outlines by mathematicians leaves those mathematicians unsatisfied and unsettled with respect to the proof. Similarly, we expect that "verifications" of programs whose overall design is incomprehensible to readers of the program (not to mention its author) will not engender much confidence in the verification. If programming is writing for others and we expect that the authors could explain their programs to their colleagues, we may well have a chance at being able to explain those programs to a computer.

Acknowledgments

The authors would like to acknowledge both the work of the team that developed ESC/Java as well as Gary Leavens and collaborators at Iowa State University who developed JML. In addition, Leavens and students engaged in and helped resolve syntactic and semantic issues in JML raised by the work on ESC/Java2. These teams provided the twin foundations on which the current work is built. Other research groups that use and critique both JML and ESC/Java2 have provided a research environment in which the work described here is useful. Thanks are due also to Leavens for his comments on an early draft of this paper.

Joseph Kiniry is supported by the NWO Pionier research project on Program Security and Correctness and the VerifiCard research project.

References

1. Many references to papers on JML can be found on the JML project website, `http://www.cs.iastate.edu/~leavens/JML/papers.shtml`.
2. J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Software*, number 2031 in Lecture Notes in Computer Science, pages 299–312. Springer–Verlag, 2001.
3. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.
4. L. Burdy and A. Requet. JACK: Java applet correctness kit. In *Proceedings, 4th Gemplus Developer Conference*, Singapore, Nov. 2002.
5. J. v. C.-B. Breunesse, B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In C. R. H. Kirchner, editor, *Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 304–318. Springer–Verlag, 2002.
6. N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study using ESC/Java. In *Proceedings, Formal Methods Europe (FME 2002)*, number 2391 in Lecture Notes in Computer Science, pages 272–289. Springer–Verlag, 2002.
7. P. Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. In *Proceedings, ECOOP'03 Workshop on Formal Techniques for Java-like Programs (FTfJP)*, Darmstadt, Germany, July 2003.
8. Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10a, Department of Computer Science, Iowa State University, Sept. 2003. Available from `http://archives.cs.iastate.edu/`.
9. E. Clarke and J. Wing. Strategic directions in computing research: Tools and partial analysis. *ACM Computing Surveys*, 28A(4), Dec. 1996.
10. D. R. Cok. Esc/java2 implementation notes, 2004. Included with all ESC/Java2 releases.
11. C. Csallner and Y. Smaragdakis. Check 'n Crash: Combining static checking and testing. Submitted for publication, 2005.
12. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021, 2001.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 2002. ACM Press.
14. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In T. Ball and S. Rajamani, editors, *Proceedings of SPIN 2003, Portland, Oregon*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135, Berlin, May 2003. Springer–Verlag.
15. E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct java card applets. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *Proceedings of the 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.

16. E.-M. Hubbers. Integrating Tools for Automatic Program Verification. In M. Broy and A. Zamulin, editors, *Proceedings of the Andrei Ershov Fifth International Conference Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 214–221. Springer–Verlag, 2003. `http://www.iis.nsk.su/psi03`.
17. E.-M. Hubbers, M. Oostdijk, and E. Poll. Implementing a Formally Verifiable Security Protocol in Java Card. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226. Springer–Verlag, 2004. March 12–14, 2003, `http://www.dfki.de/SPC2003/`.
18. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*. Springer–Verlag, Berlin, 2003.
19. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, Oct. 2000.
20. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, Apr. 2003.
21. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 2004. To appear.
22. K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Compiler Construction: 7th Internation Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer-Verlag, 1998.
23. K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, Oct. 2000.
24. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 246–257, New York, June 17–19 2002. ACM Press.
25. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1 & 2):89–106, January–March 2004.
26. H. Meijer and E. Poll. Towards a full formal specification of the Java Card. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, number 2140 in Lecture Notes in Computer Science, pages 165–178. Springer–Verlag, Sept. 2001.
27. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings, First Workshop on Runtime Verification (RV'01)*, Paris, France, July 2001.
28. Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. Technical Report SAnToS-TR2003-10, Department of Computing and Information Sciences, Kansas State University, Oct. 2003.

A Principal Changes to ESC/Java

Language semantics

- inheritance of annotations and of `non_null` modifiers that is consistent with the behavioral inheritance of JML;
- support for datagroups and `in` and `maps` clauses, which provides a sound framework for reasoning about the combination of frame conditions and subtyping;
- model import statements and model fields, routines, and types, which allow abstraction and modularity in writing specifications;
- enlarging the use and correcting the handling of scope of ghost fields, so that the syntactic behavior of annotation fields matches that of Java and other JML tools;

Language parsing

- parsing of all of current JML, even if the constructs are ignored with respect to typechecking or verification;
- support for refinement files, which allow specifications to be supplied in files separate from the source code or in the absence of source code;
- heavyweight annotations, which allow some degree of modularity and nesting;
- auto model import of the `org.jmlspecs.lang` package, similar to Java's auto import of `java.lang`;
- generalizing the use of `\old`, `set` statements and local ghost variables, to provide more flexibility in writing specifications;
- introduction of the `constraint`, `represents`, `field`, `method`, `constructor`, `\not_modified`, `instance`, `old`, `forall`, `pure` keywords as defined in JML;
- consistency in the format of annotations in order to match the language handled by other JML tools;
- equivalence of `\TYPE` and `java.lang.Class`;
- a beginning of a semantics for String objects, namely the freshness of the result of built-in + and equality and inequality of String literals.

In addition, all of the differences between JML and ESC/Java noted in the JML Reference Manual have been resolved.

B Aspects of JML Not Yet Implemented in ESC/Java2

Though the core is well-supported, there are several features of JML which are parsed and ignored, some of them experimental or not yet endowed with a clear semantics, and some in the process of being implemented. For those interested in the details of JML and ESC/Java2, the features that are currently ignored are the following:

- checking of access modifiers on annotations and of the `strictfp`, `volatile`, `transient` and `weakly` modifiers;
- the clauses `diverges`, `hence_by`, `code_contract`, `when`, and `measured_by`;

- the annotations within `implies_that` and `for_example` sections;
- some of the semantics associated with the initialization steps prior to construction;
- multi-threading support beyond that already provided in ESC/Java;
- serialization;
- annotations regarding space and time consumption;
- full support of recursive `maps` declarations;
- model programs;
- some aspects of store-ref expressions;
- verification of anonymous and block-level classes;
- verification of set comprehension and some forms of quantified expressions;
- implementation of `modifies \everything` within the body of routines.

A Type System for Checking Applet Isolation in Java Card

Werner Dietl[1], Peter Müller[1], and Arnd Poetzsch-Heffter[2]

[1] ETH Zürich, Switzerland
{Werner.Dietl,Peter.Mueller}@inf.ethz.ch
[2] Technische Universität Kaiserslautern, Germany
poetzsch@informatik.uni-kl.de

Abstract. A Java Card applet is, in general, not allowed to access fields and methods of other applets on the same smart card. This *applet isolation* property is enforced by dynamic checks in the Java Card Virtual Machine. This paper describes a refined type system for Java Card that enables static checking of applet isolation. With this type system, firewall violations are detected at compile time. Only a special kind of downcast requires dynamic checks.

1 Introduction

The Java Card technology allows applications – so-called Java Card applets – to run on smart cards. Several applets can run on a single card and share a common object store. Since the applets on a card may come from different, possibly untrusted sources, a security policy ensures that an applet, in general, cannot inspect or manipulate data of other applets. To enforce this *applet isolation* property, the Java Card Virtual Machine establishes an *applet firewall*, that is, it performs dynamic checks whenever an object is accessed, for example, by field accesses, method invocations, or casts. If an access would violate applet isolation, a `SecurityException` is thrown.

Dynamically checking applet isolation is unsatisfactory for two reasons: (1) It leads to significant runtime overhead. (2) Accidental attempts to violate the firewall are detected at runtime, that is, after the card with the defective applet has been issued, which could lead to enormous costs. In this paper, we sketch a refined type system for the Java Card language that allows one to detect most firewall violations statically by checks on the source code level. This type system serves three important purposes:

1. It reduces the runtime overhead caused by dynamic checks significantly.
2. Most firewall violations are detected at compile time. At runtime, only certain casts can lead to `SecurityException`s. These casts occur when static fields are accessed and when a reference is passed to another applet and then retrieved again. Programmers and verifiers can focus on these cast expressions when reasoning about applet isolation.
3. The refined type information provides formal documentation of the kinds of objects handled in a program such as entry point objects, global arrays, etc., and complements informal documentation, especially, of the Java Card API.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 129–150, 2005.

In this paper, we are only interested in checking applet code and do not consider the Java Card Runtime Environment (JCRE) implementation.

Overview. In the remainder of this introduction, we describe the applet firewall and our approach. Section 2 presents the refined type system, which tracks context information statically. Type safety is proved in Section 3. Based on the refined type information, context conditions can check applet isolation statically. These static checks are explained in Section 4. We discuss the presented and related work in Sections 5 and 6.

1.1 Applet Firewall

The applet firewall essentially partitions the object store of a smart card into separate protected object spaces called *contexts* [24, Sec. 6]. It allows object access across contexts only in certain cases. In this subsection, we describe contexts, object access across contexts, and the dynamic checks that enforce the firewall.

Contexts. Each applet installed on a smart card belongs to exactly one applet context. This context is determined by the package in which the applet class is declared. It contains the applet objects and all objects created by method executions in that context. The operating system of the card is contained in the Java Card Runtime Environment (JCRE) context. At any execution point, there is exactly one *currently active context* (in instance methods, this context contains `this`). When an object of context C invokes a method m on an object in context D, a *context switch* occurs, that is, D becomes the new currently active context. Upon termination of m, C is restored as the currently active context.

Class objects do not belong to any context. There is no context switch when a static method is invoked. Static fields can be accessed from any context. Objects referenced by static fields belong to an applet context or to the JCRE context.

Firewall Protection. We say that an object is *accessed* if it serves as receiver for a field access, array element access, or method invocation, if its reference is used to evaluate a cast or `instanceof` expression, or if the object is an exception that is thrown. In general, an object can only be accessed if it is in the currently active context (see below for object access across contexts). To enforce this rule, the Java Card Virtual Machine performs dynamic checks. If an object is accessed that is not in the currently active context, a `SecurityException` is thrown.

Object Access Across Contexts. The Java Card applet firewall allows certain forms of object access across contexts:

(1) Applets need access to services provided by the JCRE. These services are provided by *JCRE entry point objects*. These objects belong to the JCRE context but can be accessed by any object. There are *permanent entry point objects* (PEPs for short), *temporary entry point objects* (TEPs for short), and *global arrays*. Global arrays share many properties of TEPs: References to global

arrays and TEPs cannot be stored in fields. An applet can invoke methods on entry point objects, but not access their fields.

(2) Interaction between applets is enabled by *shareable interface objects* (SIOs for short). An object is an SIO if its class transitively implements the `Shareable` interface. An applet can get a reference to an SIO of another applet by invoking a static method of the JCRE. Access to SIOs is severely restricted. An applet can invoke those methods of an SIO which are declared in an interface that extends `Shareable`. However, it can neither access fields of SIOs nor cast an SIO to a type other than a shareable interface [24].

(3) The JCRE has access to objects in any context.

Example. The following faulty implementation of two cooperating applets illustrates the dynamic checks of the applet firewall. Fig. 1 shows the implementation of a client applet. We assume that the client and a server applet are installed on the same card, but are contained in different packages.

```
public class Status {
  private boolean success;
  public Status(boolean b)  { success = b; }
  public boolean isSuccess() { return success; }
}

public interface Service extends Shareable {
  Status doService();
}

public class Client extends Applet {
  public void process(APDU apdu) {
    AID    svr  = ...;                    // server's AID
    Shareable s = JCSystem.getAppletShareableInterfaceObject(svr, (byte)0);
    Service ser = (Service)s;             // legal cast: s refers to a Service object
    Status  sta = ser.doService();        // invocation is legal
    if (sta.isSuccess())                  // leads to SecurityException
      ...
  }
}
```

Fig. 1. Implementation of a client applet. All classes are implemented in the same package. `package` and `import` clauses are omitted for brevity. We assume that a server applet is implemented in a different package.

The following interaction is initiated by method `process`: By invoking the static method `JCSystem.getAppletShareableInterfaceObject`, the client requests an SIO from the server. This call returns an SIO that is cast to the

shareable interface `Service`. The client then invokes `doService` on the SIO. This invocation yields a new `Status` object that is used to check whether the service was rendered successfully.

This interaction leads to a `SecurityException`: The client and server applet reside in different contexts. The `Service` SIO and the `Status` object belong to the context of the server. When the invocation `sta.isSuccess()` is checked, none of the three cases for object access across contexts applies: (1) The `Status` object is not an entry point object. (2) Since `Status` does not implement `Shareable`, the `Status` object is not an SIO. (3) Since method `process` is executed on an `Applet` object, the currently active context is an applet context, not the JCRE context. Therefore, the access is denied and the exception is thrown. To correct this error, one would have to use an interface that extends `Shareable` instead of class `Status`.

1.2 Approach

To detect firewall violations at compile time, we adapt ownership type systems for alias control [11, 17, 19]. Whereas these type systems focus on restricting references between different contexts, we permit references between arbitrary contexts, but restrict the operations that can be performed on a reference across context boundaries.

Our type system augments every reference type of Java with context information that indicates (1) whether the referenced object is in the currently active context, (2) whether it is a PEP, (3) whether it is a TEP or a global array, or (4) whether it can belong to any context. Type rules guarantee that every execution state is well-typed, which means, in particular, that the context information is correct. We use downcasts to turn references of kind (4) into references of more specific types. For such casts, dynamic checks guarantee that the more specific type is legal. Otherwise, a `SecurityException` is thrown.

To check an applet with our type system, its implementation as well as the interfaces of applets it interacts with and of the Java Card API must be enriched by refined type information. This information is used to impose additional context conditions on expressions to guarantee that the firewall is respected.

In the execution of a program that is type correct according to our type system, only the evaluation of downcast expressions requires dynamic firewall checks and might lead to `SecurityException`s. Thus, casts point programmers at the critical spots in a program, which simplifies code reviews and testing. Moreover, they allow standard reasoning techniques to be applied to show that no `SecurityException` occurs [23].

2 The Type System

A type system expresses properties of the values and variables of a programming language that enable static checking of well-definedness of operations and their application conditions, in this case, Java Card's firewall constraints.

2.1 Tagged Types

In order to know whether an operation is legal in Java Card, we need information about the context in which the operation is executed. The basic idea of our approach is to augment reference types with context information.

Since we are interested in checking applet code, we consider statements and expressions that are executed in an applet context. From the point of view of an applet context, C, we can distinguish (a) internal references to objects in C, (b) PEP references, (c) TEP references including global arrays, and (d) references to objects in any context.

In the type system, this distinction is reflected by the *context tags* `i` for internal, `p` for PEP, `t` for TEP and global arrays, and `a` for any. The `a`-tag is used for references to non-TEP objects in contexts that are not known statically. For checking applet isolation, it would be desirable to have more precise information about the context of an object. However, the sharing mechanism through method `JCSystem.getAppletShareableInterfaceObject` does not provide any static information about the context of the returned SIO.

A tagged type is either a simple tagged type for primitive values or class instances, or a tagged array type.

Simple Tagged Types. Let *TypeId* denote the set of declared type identifiers of a given Java Card program; then the tagged type system comprises the following types for primitive values and class instances:

$$\mathit{SimpleTaggedType} = \{\mathit{booleanT}, \mathit{intT}, \ldots, \mathit{nullT}\} \cup (\{\mathtt{i}, \mathtt{p}, \mathtt{t}, \mathtt{a}\} \times \mathit{TypeId})$$

Except for the null-type, which is used to type the `null` literal, all reference types in the tagged type system are denoted as a pair of a tag and a Java type. In actual code examples we will use the keywords `intern`, `pep`, `tep`, and `any` instead of the symbols used for the formalization.

Tagged Types. In general, an array type has two tags: The *array tag* specifies the context that contains the array object, whereas the *element tag* specifies the context of the array elements relatively to the context of the array object. For instance, an array of type `intern any Object[]` belongs to the currently active context and stores objects belonging to any context.

Global arrays serve as temporary entry points to the JCRE context. Therefore, we use the `tep` tag to mark an array as global. For instance, the `APDU` buffer, a global array of bytes, has type `tep byte[]`. Since the element type is a primitive type here, there is no element tag.

Formally, a tagged type is either a simple tagged type or an array type. Since Java Card does not provide multi-dimensional arrays, the array elements have a simple tagged type. Permanent entry point arrays do not exist in Java Card.

$$\mathit{TaggedType} = \mathit{SimpleTaggedType} \cup (\{\mathtt{i}, \mathtt{t}, \mathtt{a}\} \times \mathit{SimpleTaggedType})$$

Notation. In the following, the meta-variables S and T denote Java types; TS and TT range over *TaggedType*. Calligraphic $\mathcal{S}$ and $\mathcal{T}$ can stand for Java types or *TaggedType*s. It is often convenient to use a tuple notation for tagged reference types. (γ, T) is the simple tagged type for objects of Java type T with tag γ. (γ, TT) is a tagged array type with element type TT. $(\gamma, \mathcal{T})$ can be the type of both a class instance or an array.

Subtyping on Tagged Types. The subtype relation $\preceq$ on tagged types follows Java's subtype relation $\preceq_J$ on Java types. It is the smallest reflexive and transitive relation satisfying the following axioms.

(1) $(\gamma, \mathcal{T}) \preceq (\gamma, \texttt{java.lang.Object})$
(2) $nullT \preceq (\gamma, \mathcal{T})$
(3) $(\texttt{i}, \mathcal{T}) \preceq (\texttt{a}, \mathcal{T})$
(4) $(\texttt{p}, \mathcal{T}) \preceq (\texttt{a}, \mathcal{T})$
(5) $(\gamma, S) \preceq (\gamma, T) \Leftrightarrow S \preceq_J T$
(6) $(\gamma, (\delta, S)) \preceq (\gamma, (\delta, T)) \Leftrightarrow S \preceq_J T$

Every reference type is a subtype of the tagged type for `Object`, provided that both types have the same tag (1). The null-type is a subtype of any tagged reference type (2). `intern` and `pep` types are subtypes of the corresponding `any` type (3,4). Note that there is no such axiom for `tep` types. `tep` types must not be subsumed under `any` types to prevent TEPs from being stored in fields or arrays (see Section 4.1). Two tagged types with the same tag are subtypes iff the corresponding Java types are subtypes (5). Covariant array subtyping requires runtime checks for each array update. For tagged types, these checks would involve context information and could throw `SecurityException`s. To avoid such checks, we allow only limited covariant subtyping of tagged array types. Two tagged array types can only be subtypes if they have the same element tag (6). That is, covariant subtyping is only possible in terms of Java types, but not of tags. For instance, if S is a subtype of T then `intern intern S` is a subtype of `intern intern T`, but not of `intern any S`.

Since Java Card imposes weaker restrictions on PEPs than on TEPs, we could allow `pep` types to also be subtypes of the corresponding `tep` types, and forbid downcasts from `tep` to `pep`. We omitted this subtype relation for simplicity.

Casts. Casts on tagged types work analogously to Java. A downcast can be used to specialize the tagged type of an expression, in particular, the context information. For instance, an expression of type $(\texttt{a}, T)$ can be cast to $(\texttt{i}, T)$. A runtime check ensures that the refined context information is correct. If not, a `SecurityException` is thrown.

Example. Fig. 2 shows the `Service` interface and the `Client` class from Fig. 1 with tagged type information. The return type of `Service.doService` is internal since the method creates a new `Status` object in the context in which it is executed (the context of the server applet). When `doService` is invoked from the client context, the returned `Status` object is external to the client context and must, thus, be tagged `any`. The type rules that enforce these tags are discussed in the next subsection. The static checks that detect the firewall violation are presented in Sec. 4.

```
public interface Service extends Shareable {
  intern Status doService();
}

public class Client extends Applet {
  public void process(tep APDU apdu) {
    pep AID      svr  = ...;                    // server's applet id is a PEP
    any Shareable  s =
            JCSystem.getAppletShareableInterfaceObject(svr, (byte)0);
    any Service    ser = (any Service)s;        // ser is in general extern
    any Status     sta = ser.doService();       // sta is also extern
    if (sta.isSuccess())                        // static firewall check fails
      ...
  }
}
```

Fig. 2. `Service` interface and `Client` class with tagged types.

2.2 Tagged Type Rules

In the tagged type system, a type judgment of the form $\vdash e :: TT$ means that expression e is well-typed and has tagged type TT. $\vdash s$ expresses that statement s is well-typed. In the formalization, we omit the declaration environment and all rules that handle the environment. Instead, we use $[f]$, TP, and TR to denote the tagged type of a field f, the (sole) parameter type, and the return type of a method, respectively. A more complete formalization of Java's type system including declaration environments is presented in [20, 22, 12].

Fig. 3 shows the most interesting rules of the tagged type system. Since the type rules for statements are trivial, we focus on expressions here. In the type rules, premises marked by ($\star$) are only needed for static checks of applet isolation. These premises will be discussed in Section 4.1. The function *ShareItf?* yields whether the argument is an interface that extends `Shareable`.

For brevity, we do not present the rules for exceptions. Like all reference types, exceptions are tagged. For `throw` and `try` statements, as well as for the declaration of exceptions in method signatures, the normal Java rules apply based on the subtyping of tagged types. The rules for method invocations treat exceptions analogously to normal return values.

Object Creation, Cast, and `instanceof`. Newly created objects always belong to the currently active context. Therefore, the type of the `new` expression has tag `intern` (T-New and T-NewArray). For simplicity, we assume that a `new` expression directly returns a fresh object without calling a constructor.

The tagged type of a cast expression is the type TT appearing in the cast operator (T-Cast). For simplicity, we do not allow upcasts. That is, TT has to be a subtype of the expression type TS. Upcasts can be simulated by an assignment to a local variable of the desired type.

$$\text{T-New}\ \frac{}{\vdash \texttt{new T()} :: (\texttt{i}, T)} \qquad \text{T-NewArray}\ \frac{\vdash e :: intT}{\vdash \texttt{new TT[e]} :: (\texttt{i}, TT)}$$

$$\text{T-Cast}\ \frac{\vdash e :: TS \qquad TT \preceq TS \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \wedge TT = (\gamma, \mathcal{T}) \Rightarrow (\mathcal{S} \preceq_J \texttt{Shareable}^1 \wedge \mathit{ShareItf?}(\mathcal{T}) \vee \mathcal{S} = \mathcal{T})}{\vdash \texttt{(TT) e} :: TT}$$

$$\text{T-Instanceof}\ \frac{\vdash e :: TS \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \wedge TT = (\gamma, \mathcal{T}) \Rightarrow (\mathcal{S} \preceq_J \texttt{Shareable}^1 \wedge \mathit{ShareItf?}(\mathcal{T}) \vee \mathcal{S} = \mathcal{T})}{\vdash \texttt{e instanceof TT} :: \mathit{booleanT}}$$

$$\text{T-Invoke}\ \frac{\vdash e1 :: TS \quad \vdash e2 :: TT \quad TS * TT \preceq TP \qquad (\star)\ TS = (\texttt{a}, \mathcal{S}) \Rightarrow \mathit{ShareItf?}(\mathcal{S})}{\vdash \texttt{e1.m(e2)} :: TS * TR} \qquad \text{T-SInvoke}\ \frac{\vdash e :: TS \qquad TS \preceq TP}{\vdash \texttt{T.m(e)} :: TR}$$

$$\text{T-FRead}\ \frac{\vdash e :: TS \qquad (\star)\ TS = (\texttt{i}, \mathcal{S})}{\vdash \texttt{e.f} :: [f]} \qquad \text{T-FWrite}\ \frac{\vdash e1 :: TS \quad \vdash e2 :: TT \quad TT \preceq [f] \qquad (\star)\ TS = (\texttt{i}, S) \quad (\star)\ TT \neq (\texttt{t}, \mathcal{T})}{\vdash \texttt{e1.f=e2} :: TT}$$

$$\text{T-SRead}\ \frac{}{\vdash \texttt{T.f} :: [f]} \qquad \text{T-SWrite}\ \frac{\vdash e :: TT \quad TT \preceq [f] \quad (\star)\ TT \neq (\texttt{t}, \mathcal{T})}{\vdash \texttt{T.f=e} :: TT}$$

$$\text{T-ARead}\ \frac{\vdash e1 :: (\gamma, TE) \quad \vdash e2 :: intT \quad (\star)\ \gamma = \texttt{i} \vee \gamma = \texttt{t}}{\vdash \texttt{e1[e2]} :: (\gamma, TE) * TE}$$

$$\text{T-AWrite}\ \frac{\vdash e1 :: (\gamma, TE) \quad \vdash e2 :: intT \quad \vdash e3 :: TT \quad (\gamma, TE) * TT \preceq TE \qquad (\star)\ \gamma = \texttt{i} \vee \gamma = \texttt{t} \qquad (\star)\ TT \neq (\texttt{t}, T)}{\vdash \texttt{e1[e2]=e3} :: TT}$$

Fig. 3. Tagged type rules.

Method Invocation. For simplicity, we assume that methods have exactly one formal parameter.

The rule for the invocation of instance methods (T-Invoke) has to handle context switches. Consider for example the invocation `ser.doService` in Fig. 2. The declared return type of `doService` is `intern` because the result object is intern to the server context in which the method is executed. When `doService` is invoked from the client context, the returned `Status` object is external to the client context and, therefore, must be tagged `any`. This adaption of the tag is described by the $*$-operator, which combines two tagged types. It is defined as follows:

[1] We write $\mathcal{S} \preceq_J \texttt{Shareable}$ to express that $\mathcal{S}$ is a Java type, which is a subtype of `Shareable`.

$$
\begin{aligned}
* : &\ \mathit{TaggedType} \times \mathit{TaggedType} \to \mathit{TaggedType} \\
&(\gamma, \mathcal{T}) * (\mathtt{i}, \mathcal{S}) = (\mathtt{a}, \mathcal{S}), \qquad \text{if } \gamma \neq \mathtt{i} \\
&(\gamma, \mathcal{T}) * \mathit{TS} \quad\ = \mathit{TS}, \qquad\quad \text{in all other cases}
\end{aligned}
$$

The $*$-operator tags the parameter or result as `any` when the invocation could lead to a context switch ($\gamma \neq \mathtt{i}$) and an internal reference is passed to or returned by the method.

Static methods are always executed in the currently active context. Therefore, in rule T-SInvoke, the tags do not need to be adapted by the $*$-operator.

Field and Array Access. The type of a field read is the declared type of the field, $[f]$ (T-FRead, T-SRead). T-FWrite and T-SWrite check that the tagged type of the right-hand side of a field update is a subtype of the declared type of the field.

Besides the rules for accessing static fields, there is also a requirement for their declaration: Since class objects do not belong to any context, static fields must not have an `intern` type.

Tagged element types specify the context of array elements relatively to the context of the array object. Therefore, the $*$-operator is used to combine the tagged array type, (γ, TE), with the tagged element type, TE, to determine the type of an array read access (T-ARead). Similarly, the $*$-combination of the array type and the type of the right-hand side expression of an array update has to be a subtype of the element type (T-AWrite).

2.3 Annotations for the Java Card API

To typecheck applet implementations, tags have to be added to the Java Card API. In particular, these tags determine which objects are entry point objects. Fig. 4 illustrates such API annotations for three methods of class `JCSystem`. According to the API specifications, method `getAID` returns an `AID` object that is a PEP. `getAppletShareableInterfaceObject` takes a `pep AID` and returns a reference to an SIO in any applet context, hence the result type `any Shareable`. Method `makeTransientByteArray` illustrates that exceptions thrown by the JCRE are TEPs. Since the method creates a new array in the context in which it is called, the result type has tag `intern`.

3 Dynamic Semantics

In this section, we formalize and prove type safety based on an operational semantics of a subset of Java Card. Although we have proved type safety for the full language, we omit primitive types, arrays, and exceptions in this formalization for simplicity.

3.1 State Model

We build on the formalization of the state model of Java presented in [23]. In the following, we summarize those aspects that are specific to Java Card such as the treatment of contexts.

```
public final class JCSystem {
  static pep AID getAID() {...}
  static any Shareable getAppletShareableInterfaceObject
                                (pep AID serverAID, byte parameter) {...}
  static intern byte[ ] makeTransientByteArray(short length, byte event)
            throws tep NegativeArraySizeException, tep SystemException {...}
  // other methods omitted
}
```

Fig. 4. Tags for selected methods of the Java Card API.

Contexts, Objects, and Values. A *Context* is either the JCRE context or an applet context, defined by a package name. A key property of the formalization is that each object "knows" the context it belongs to and whether it is a PEP or TEP. Since we do not consider primitive types here, a *Value* is either a reference to an object or *null*. Sorts *PackageId*, *ClassId*, and *ObjId* stand for package names, class names, and object identifiers (addresses), respectively. The function *ctxt* yields the context an object belongs to.

$$
\begin{aligned}
&\mathit{Context} = \mathit{jcreC} \\
&\qquad \mid \mathit{appletC}(\ \mathit{PackageId}\ ) \\
&\mathit{Value} = \mathit{ref}(\ \mathit{Object}\ ) \\
&\qquad \mid \mathit{null} \\
&\mathit{Object} = o(\ \mathit{ClassId}, \mathit{ObjId}, \mathit{Context}\ ) \\
&\qquad \mid \mathit{pepo}(\ \mathit{ClassId}, \mathit{ObjId}\ ) \\
&\qquad \mid \mathit{tepo}(\ \mathit{ClassId}, \mathit{ObjId}\ )
\end{aligned}
\qquad
\begin{aligned}
&\mathit{ctxt} : \mathit{Value} \to \mathit{Context} \cup \{\mathit{undef}\} \\
&\mathit{ctxt}(\mathit{ref}(o(T, O, C))) = C \\
&\mathit{ctxt}(\mathit{ref}(\mathit{pepo}(T, O))) = \mathit{jcreC} \\
&\mathit{ctxt}(\mathit{ref}(\mathit{tepo}(T, O))) = \mathit{jcreC} \\
&\mathit{ctxt}(\mathit{null}\ ) = \mathit{undef}
\end{aligned}
$$

In addition to these definitions, we use the following functions: *typeof* yields the dynamic Java type of a value. *pepo*? and *tepo*? test whether an object is a PEP or a TEP.

Object Stores. The state of an object is given by the values of its instance variables. We assume a sort *Location* for the instance variables of objects and the static fields of classes. The functions

$$
\begin{aligned}
&\mathit{iv} : \mathit{Value} \times \mathit{FieldId} \to \mathit{Location} \cup \{\mathit{undef}\} \\
&\mathit{sv} : \mathit{ClassId} \times \mathit{FieldId} \to \mathit{Location} \cup \{\mathit{undef}\}
\end{aligned}
$$

are used to create a location from a value (or class) and a field name (sort *FieldId*).

The state of all objects in the current execution state is formalized by an abstract data type *Store* with the following functions:

$$
\begin{aligned}
&\_(\_) : \mathit{Store} \times \mathit{Location} \to \mathit{Value} \\
&\_\langle \_ := \_ \rangle : \mathit{Store} \times \mathit{Location} \times \mathit{Value} \to \mathit{Store} \\
&\_\langle \_, \_ \rangle : \mathit{Store} \times \mathit{ClassId} \times \mathit{Context} \to \mathit{Store} \\
&\mathit{new} : \mathit{Store} \times \mathit{ClassId} \times \mathit{Context} \to \mathit{Object}
\end{aligned}
$$

$OS(L)$ yields the value of location L in store OS. $OS\langle L := V\rangle$ yields the object store that is obtained from OS by updating location L with value V. $OS\langle T, C\rangle$ yields the object store that is obtained from OS by allocating a new object of type T in context C. $new(OS, T, C)$ yields a reference to a new object of type T in context C. The functions for object creation, $OS\langle T, C\rangle$ and $new(OS, T, C)$, are connected by appropriate axioms. Since these axioms as well as other properties of the above functions are not needed in this paper, we refer the reader to [23] for their axiomatization.

Program States. Program states are formalized as mappings from identifiers to values. A designated variable $\mathcal{C}$ contains the currently active context. We use \$ as identifier for the current object store. We assume that each method has exactly one formal parameter, `p`. *VarId* is the set of identifiers for local variables.

$$\begin{aligned} State \equiv\ & (VarId \cup \{\texttt{this}, \texttt{p}\} \to Value \cup \{undef\}) \quad \cup \\ & (\{\$\} \to Store) \quad \cup \quad (\{\mathcal{C}\} \to Context) \end{aligned}$$

For $\sigma \in State$, we write $\sigma(x)$ for the application to a variable or parameter identifier x. In static methods, we set $\sigma(\texttt{this}) = null$. By $\sigma[x := V]$, we denote the state that is obtained from σ by updating variable x with value V. An analogous notation is used for the current object store, \$, and the currently active context, $\mathcal{C}$. *initS* denotes the state that is undefined for all variables, \$, and $\mathcal{C}$.

3.2 Operational Semantics

The operational semantics has two kinds of transitions: $\sigma :: e \to V, \sigma'$ expresses that the evaluation of expression e in state σ yields value V and final state σ'. For statements, $\sigma : s \to \sigma'$ expresses that the execution of statement s in state σ leads to state σ'. Since the rules for statements are the usual Java rules, we omit them here and refer the reader to [21].

The rules for expressions are found in Fig. 5. In the rules, we mark the premises for the dynamic firewall checks with "$(\star)$". We refer to the semantics including the dynamic firewall checks as *strong* semantics, whereas the *weak* semantics does not contain these checks. In the following, we use $\to$ and $\to^\star$ to denote transitions in the weak and strong semantics, respectively.

Following Drossopoulou and Eisenbach [12], we assume that all expressions are annotated with their static types. These annotated versions of the expressions are produced by the type rules, although we leave that implicit in Fig. 3. In the semantics rules, the static Java type of an expression e is denoted by $[e]$.

The most complex rule handles invocations of instance methods (S-Invoke). $impl(T, m)$ yields the implementation of method m in type T. This implementation can be inherited from a superclass. First, the receiver and actual parameter expressions are evaluated. Next, the implementation of the dynamically-bound method m is executed in a state that maps the formal parameters to the actual parameters and \$ to the store after evaluating the actual parameter. The new currently active context is the context of the receiver object. That is, a context

$$\text{S-New}\ \frac{}{\sigma :: \texttt{new T()} \rightarrow new(\sigma(\$), T, \sigma(\mathcal{C})), \sigma[\$:= \sigma(\$) < T, \sigma(\mathcal{C}) >]}$$

$$\text{S-Cast}\ \frac{\begin{array}{c}\sigma :: e \rightarrow V, \sigma' \qquad ttype(V, \sigma(\mathcal{C})) \preceq TT \\ (\star)\ ctxt(V) = \sigma(\mathcal{C}) \vee pepo?(V) \vee tepo?(V) \vee \\ (typeof(V) \preceq_J \texttt{Shareable} \wedge ShareItf?(TT))\end{array}}{\sigma :: (TT)e \rightarrow V, \sigma'}$$

$$\text{S-Invoke}\ \frac{\begin{array}{c}\sigma :: e1 \rightarrow V1, \sigma' \qquad \sigma' :: e2 \rightarrow V2, \sigma'' \qquad V1 \neq null, \\ initS[\texttt{this} := V1, \texttt{p} := V2, \$:= \sigma''(\$), \mathcal{C} := ctxt(V1)] : impl(typeof(V1), m) \rightarrow \sigma''' \\ (\star)\ ctxt(V1) = \sigma(\mathcal{C}) \vee pepo?(V1) \vee tepo?(V1) \vee ShareItf?([e1])\end{array}}{\sigma :: e1.m(e2) \rightarrow \sigma'''(\texttt{res}), \sigma''[\$:= \sigma'''(\$)]}$$

$$\text{S-SInvoke}\ \frac{\sigma :: e \rightarrow V, \sigma' \qquad initS[\texttt{p} := V, \$:= \sigma'(\$), \mathcal{C} := \sigma(\mathcal{C})] : impl(T, m) \rightarrow \sigma'''}{\sigma :: T.m(e) \rightarrow \sigma'''(\texttt{res}), \sigma'[\$:= \sigma'''(\$)]}$$

$$\text{S-FRead}\ \frac{\sigma :: e \rightarrow V, \sigma' \qquad V \neq null \qquad (\star)\ ctxt(V) = \sigma(\mathcal{C})}{\sigma :: e.f \rightarrow \sigma'(\$)(iv(V, f)), \sigma'}$$

$$\text{S-FWrite}\ \frac{\begin{array}{c}\sigma :: e1 \rightarrow V1, \sigma' \qquad \sigma' :: e2 \rightarrow V2, \sigma'' \qquad V1 \neq null \\ (\star)\ ctxt(V1) = \sigma(\mathcal{C}) \qquad (\star)\ \neg tepo?(V2)\end{array}}{\sigma :: e1.f = e2 \rightarrow V2, \sigma''[\$:= \sigma''(\$) < iv(V1, f) := V2 >]}$$

$$\text{S-SRead}\ \frac{}{\sigma :: T.f \rightarrow \sigma(\$)(sv(T, f)), \sigma}$$

$$\text{S-SWrite}\ \frac{\sigma :: e \rightarrow V, \sigma' \qquad (\star)\ \neg tepo?(V)}{\sigma :: T.f = e \rightarrow V, \sigma'[\$:= \sigma'(\$) < sv(T, f) := V >]}$$

Fig. 5. Selected rules of the operational semantics.

switch may occur. The return value of the method is stored in the special variable, `res`. The strong semantics requires in addition that the receiver object is in the currently active context, an entry point object, or that the static type of the receiver is a shareable interface[2]. These conditions correspond to the firewall checks described in Section 1.1.

3.3 Type Safety

Type safety w.r.t. tagged types means that the tag of the static type of a program element e correctly reflects the context the object denoted by e belongs to. For

[2] The Java Card documentation [24] formulates these rules for bytecode instructions. Java bytecode provides different instructions for methods declared in classes and interfaces. This distinction is reflected in our uniform invocation rule by referring to the static type of the receiver.

instance, the object held by a local variable of type `intern T` in an execution state σ has to belong to the currently active context of σ.

Most Specific Tagged Types. An object knows its class, whether it is a PEP, a TEP, or an ordinary object, and its context. Tagged types approximate this information statically. The best approximation for an object X relative to a context C is determined by $\mathit{ttype}(X, C)$. In particular, for a non-entry point object X, $\mathit{ttype}(X, C)$ yields an `intern` type if X is in context C and an `any` type if X belongs to a different context. For example, if X is an instance of class T in context C, then the *most specific tagged type* relative to context C is $(\mathtt{i}, T)$ because X is intern to C. $(\mathtt{a}, T)$ would also be a valid tagged type for X, but is not the most specific one. Function *ttype* is defined as follows:

$$
\begin{aligned}
&\mathit{ttype} : \mathit{Value} \times \mathit{Context} \to \mathit{TaggedType}\\
&\quad \mathit{ttype}(\mathit{ref}(o(T, O, C)), C) = (\mathtt{i}, T)\\
&\quad \mathit{ttype}(\mathit{ref}(o(T, O, C)), D) = (\mathtt{a}, T) \text{ for } C \neq D\\
&\quad \mathit{ttype}(\mathit{ref}(\mathit{pepo}(T, O)), D) = (\mathtt{p}, T)\\
&\quad \mathit{ttype}(\mathit{ref}(\mathit{tepo}(T, O)), D) = (\mathtt{t}, T)\\
&\quad \mathit{ttype}(\mathit{null}, D) = \mathit{nullT}
\end{aligned}
$$

Well-Typed States. Based on the function *ttype*, we can define well-typed states: The most specific tagged type for a variable and a context has to be a subtype of the declared type of the variable (written as $[v]$ for a variable v).

Definition 1 (Well-Typed States). *A state is well-typed if (1) the local variables and formal parameters are correctly typed relative to the currently active context; (2) all instance variables $X.f$ are correctly typed relative to the context of X; (3) all static fields $T.f$ are correctly typed relative to any context:*

$$
\begin{aligned}
&\mathit{wt} : \mathit{State} \to \mathit{Bool}\\
&\mathit{wt}(\sigma) \Leftrightarrow (\forall v \in \mathit{VarId} \cup \{\mathtt{this}, \mathtt{p}\} : \mathit{ttype}(\sigma(v), \sigma(\mathcal{C})) \preceq [v]) \wedge\\
&\qquad\qquad (\forall L \in \mathit{Location} : L = \mathit{iv}(X, f) \Rightarrow \mathit{ttype}(\sigma(\$)(L), \mathit{ctxt}(X)) \preceq [f]) \wedge\\
&\qquad\qquad (\forall L \in \mathit{Location} : L = \mathit{sv}(T, f) \Rightarrow \forall C \in \mathit{Context} : \mathit{ttype}(\sigma(\$)(L), C) \preceq [f])
\end{aligned}
$$

For objects that are neither PEPs nor TEPs, *ttype* uses the context argument C to determine whether the object is internal to C (tag `i`) or not (tag `a`). If a local variable or formal parameter is typed `intern`, the referenced object has to be in the currently active context, $\sigma(\mathcal{C})$. If an instance field f is typed `intern`, the object referenced by $X.f$ has to be in the same context as X. Since static fields can be read and written from any context, they cannot be typed `intern`. Requiring that the object X referenced by static field $T.f$ is correctly typed relative to *any* context enforces that X is a permanent entry point object or $[f]$ has tag `any`.

Java Card is type safe w.r.t. the tagged type system. That is, the type rules – *without* the static firewall checks – ensure that tags correctly reflect dynamic context information. Type safety does not rely on the dynamic firewall checks. That is, it can be proved based on the weak operational semantics.

Theorem 1 (Type Safety). *If the evaluation of a well-typed expression e starts in a well-typed state, σ, and terminates then the final state, σ', is well-typed and has the same currently active context as σ. The resulting value is correctly typed:*

$$\vdash e :: TT \wedge \sigma :: e \rightarrow V, \sigma' \wedge wt(\sigma) \Rightarrow wt(\sigma') \wedge ttype(V, \sigma'(\mathcal{C})) \preceq TT \wedge \sigma(\mathcal{C}) = \sigma'(\mathcal{C})$$

If the execution of a well-typed statement s starts in a well-typed state, σ,and terminates then the final state, σ', is well-typed and has the same currently active context as σ:

$$\vdash s \wedge \sigma : s \rightarrow \sigma' \wedge wt(\sigma) \Rightarrow wt(\sigma') \wedge \sigma(\mathcal{C}) = \sigma'(\mathcal{C})$$

The proof of this theorem uses the following auxiliary lemma. This lemma is used to relate (i) the argument of a method call to the context in which the method is executed and (ii) the result of a call to the context of the caller.

Lemma 1 (Combination Lemma). *Let TS be the tagged type of object X relative to a context C. (i) If TT is the tagged type of value Y relative to C, then the tagged type of Y relative to the context of X is a subtype of $TS * TT$. (ii) If TT is the tagged type of value Y relative to the context of X, then the tagged type of Y relative to C is a subtype of $TS * TT$.*

$$(i)\ \ X \neq null \wedge ttype(X, C) = TS \wedge ttype(Y, C) = TT \Rightarrow ttype(Y, ctxt(X)) \preceq TS * TT$$

$$(ii)\ X \neq null \wedge ttype(X, C) = TS \wedge ttype(Y, ctxt(X)) = TT \Rightarrow ttype(Y, C) \preceq TS * TT$$

Proof: The proof of Lemma 1 runs by case distinction on the tags of TS and TT. It is straightforward and, therefore, omitted.

Proof of Type Safety. The proof of Theorem 1 runs by rule induction on the rules of the weak operational semantics. For brevity, we show only the most interesting case, calls of instance methods. Consider the invocation $e1.m(e2)$. We have to prove that if the evaluation of the call starts in a well-typed state then (1) the state in which the implementation of m is executed is well-typed. This is necessary to establish the induction hypothesis for the method implementation; (2) the induction hypothesis holds for the final state of the evaluation of $e1.m(e2)$. In the following, TS, TT, TP, and TR are used like in the type rule T-Invoke (Fig. 3).

Part 1: $TT_{\texttt{this}}$ denotes the tagged type of the implicit parameter of m's implementation. $TT_{\texttt{this}} = (\texttt{i}, S)$, where S is the class in which m is implemented. That is, $typeof(V1) \preceq_J S$.

$wt(\sigma)$
$\quad \Rightarrow$ [induction hypothesis for $\sigma :: e1 \rightarrow V1, \sigma'$ and $\sigma' :: e2 \rightarrow V2, \sigma''$]
$wt(\sigma'') \wedge ttype(V1, \sigma(\mathcal{C})) \preceq TS \wedge ttype(V2, \sigma(\mathcal{C})) \preceq TT$
$\quad \Rightarrow$ [Lemma 1 (i), $V1 \neq null$]
$wt(\sigma'') \wedge ttype(V2, ctxt(V1)) \preceq TS * TT$
$\quad \Rightarrow$ [$TT_{\texttt{this}} = (\texttt{i}, S)$, $typeof(V1) \preceq_J S$; $TS * TT \preceq TP$]
$wt(\sigma'') \wedge ttype(V1, ctxt(V1)) \preceq TT_{\texttt{this}} \wedge ttype(V2, ctxt(V1)) \preceq TP$
$\quad \Rightarrow$
$wt(initS[\texttt{this} := V1, \texttt{p} := V2, \$:= \sigma''(\$), \mathcal{C} := ctxt(V1)])$

Part 2:

$wt(\sigma)$
$\quad\Rightarrow$ [induction hypothesis for $\sigma :: e1 \to V1, \sigma'$ and $\sigma' :: e2 \to V2, \sigma''$]
$wt(\sigma'') \wedge ttype(V1, \sigma'(\mathcal{C})) \preceq TS \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C}) \wedge \sigma'(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow$ [induction hypothesis for method implementation]
$wt(\sigma'') \wedge wt(\sigma''') \wedge ttype(V1, \sigma'(\mathcal{C})) \preceq TS \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C}) \wedge \sigma'(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow$ [`res` is a local variable of m's implementation with type TR]
$wt(\sigma'') \wedge wt(\sigma''') \wedge ttype(V1, \sigma''(\mathcal{C})) \preceq TS \wedge$
$ttype(\sigma'''(\texttt{res}), ctxt(V1)) \preceq TR \wedge \sigma(\mathcal{C}) = \sigma''(\mathcal{C})$
$\quad\Rightarrow$ [Lemma 1 (ii), $V1 \neq null$]
$wt(\sigma''[\$:= \sigma'''(\$)]) \wedge ttype(\sigma'''(\texttt{res}), \sigma''[\$:= \sigma'''(\$)](\mathcal{C})) \preceq TS * TR \wedge$
$\sigma(\mathcal{C}) = \sigma''[\$:= \sigma'''(\$)](\mathcal{C})$

□

Type Progress. Besides type safety, progress is an interesting property of a type system: Progress means that a well-typed program can actually be executed, that is, applying the rules of the operational semantics does not lead to stuck configurations. We do not prove progress formally in this paper. However, one can easily show that the tagged type system guarantees progress if the original Java Card type system does: (1) If a program $\mathbf{P}_{tJC}$ is well-typed in the tagged type system, then the Java Card program $\mathbf{P}_{JC}$ obtained from $\mathbf{P}_{tJC}$ by omitting all tags is well-typed in the Java Card type system, because the tagged type system only imposes additional checks. (2) Besides minor differences for object creation and cast, the strong operational semantics for $\mathbf{P}_{JC}$ and $\mathbf{P}_{tJC}$ are identical. That is, since $\mathbf{P}_{JC}$ can be executed (progress of the Java Card type system), $\mathbf{P}_{tJC}$ can be executed as well. (3) The weak operational semantics is obtained from the strong operational semantics by omitting several requirements. Therefore, $\mathbf{P}_{tJC}$ can also be executed in the weak operational semantics.

4 Checking Applet Isolation

Tagged types provide a conservative approximation of runtime context information. This information can be used to impose *static checks* that guarantee that an applet respects the applet firewall at runtime. In the following, we explain these checks and prove that they enforce applet isolation.

4.1 Static Checks

Applet isolation is enforced by additional checks in the tagged type rules (Fig. 3), which are marked by ($\star$). We will explain these premises below.

Object Creation, Cast, and `instanceof`. Object creation is always allowed (T-New, T-NewArray). In Java Card, even finding out type information about objects in other applet contexts is considered a security violation. Therefore, casts are allowed if the object is in the currently active context, if it is an entry

point object, or if the object's class implements `Shareable` and the object is cast into a shareable interface. That is, the cast is legal if the object is intern, TEP, or PEP. For tag `any`, we check that the object's class implements `Shareable` and that it is cast into a shareable interface (T-Cast). Moreover, we allow casts from `any T` to `intern T` or `pep T` to refine the tagged type information. The rule for `instanceof` expressions (T-Instanceof) is analogous.

Method Invocation. Instance methods can be invoked on objects (including arrays) in the currently active context, on PEPs and TEPs, and if the static type of the receiver is a shareable interface. Rule T-Invoke requires that if the receiver can be in any context (tag `any`), then its static Java type must be a shareable interface. Static methods can be invoked from any context and need no checks (T-SInvoke).

Field and Array Access. As mentioned in Section 1.1, Java Card forbids field access on objects (including the `length` field of arrays) not in the currently active context. Therefore, the type of the receiver must have tag `intern` (T-FRead, T-FWrite). Since it is not allowed to store TEPs in fields, the right-hand side of a field update must not have tag `tep`. Static fields can be accessed from any context. Therefore, only the check for TEP objects is required (T-SRead, T-SWrite).

Access to an array element is only allowed if the array is either in the currently active context or a global array. Therefore, rules T-ARead and T-AWrite require the tag of the receiver expression to be `intern` or `tep`. Like for field updates, the tagged type of the right-hand side of an array update must not have tag `tep`.

Example. In the example in Fig. 2, the firewall violation would be detected statically. Since `sta` has tag `any`, the invocation `sta.isSuccess()` does not pass the static checks of rule T-Invoke: `Status` is a class and does not implement `Shareable`.

4.2 Applet Isolation Lemma

The static checks described above guarantee applet isolation: Each Java Card program with tagged types that passes the static checks behaves like the corresponding Java Card program with dynamic checks. That is, every Java Card program that can be correctly tagged does not throw `SecurityException`s (except for the dynamic checks for casts).

Theorem 2 (Applet Isolation). *Let e be an expression that can be typed in the tagged type system and that passes the static firewall checks. If e's evaluation in a well-typed state σ terminates normally then the evaluation of the corresponding Java Card expression without tags, $\hat{e}$, in σ terminates normally in the same final state and yields the same value:*

$$\vdash^\star e :: TT \wedge \sigma :: e \to V, \sigma' \wedge wt(\sigma) \Rightarrow \sigma :: \hat{e} \to^\star V, \sigma'$$

where $\vdash^\star e :: TT$ *denotes that* e *is well-typed and passes the static firewall checks.* $\rightarrow$ *and* $\rightarrow^\star$ *denote transitions in the weak and strong semantics, respectively.*

Note that omitting tags from expressions makes those casts dispensable that do not change the Java type of an expression. For instance, for a variable v of tagged type $(\mathtt{a}, T)$, omitting the tags from the cast $(\mathtt{i}\ T)v$ yields v. For such cast expressions, the implication of the theorem is trivially true.

Proof of Applet Isolation. The proof of Theorem 2 runs by rule induction on the weak operational semantics. It uses type safety of the tagged type system and the static firewall checks. Again, we show the proof for the most interesting case: the invocation of instance methods.

Since we have the transition $\sigma :: e \rightarrow V, \sigma'$, we know that all premises of rule S-Invoke in the weak semantics hold. Applying the induction hypothesis to these premises yields the corresponding premises in the strong semantics. It remains to show that the additional premise in the strong semantics holds: $\mathit{ctxt}(V1) = \sigma(\mathcal{C}) \vee \mathit{pepo}?(V1) \vee \mathit{tepo}?(V1) \vee \mathit{ShareItf}?([e1])$.

From the premise $\sigma :: e1 \rightarrow V1, \sigma'$ and type safety (Theorem 1), we get $\mathit{ttype}(V1, \sigma(\mathcal{C})) \preceq TS$. We may assume that TS is a reference type $(\gamma, [e1])$. We continue by case distinction on the tag γ:

1. *Case* `i`: Subtyping on tagged types gives that $\mathit{ttype}(V1, \sigma(\mathcal{C}))$ has tag `i`. The definition of *ttype* yields $V1 = \mathit{ref}(o(S, O, \sigma(\mathcal{C})))$ for some S, O. Therefore, $\mathit{ctxt}(V1) = \sigma(\mathcal{C})$.
2. *Case* `p`: Analogously to Case `i`, we get $V1 = \mathit{ref}(\mathit{pepo}(S, O))$ for some S, O. Therefore, $\mathit{pepo}?(V1)$ holds.
3. *Case* `t`: This case is analogous to Case `p`.
4. *Case* `a`: The static firewall check of rule (T-Invoke) gives directly the result $\mathit{ShareItf}?([e1])$

□

Theorem 2 shows that well-typedness and the static firewall checks guarantee that execution of an expression does not violate the firewall at runtime. Therefore, the checks can be used to enforce applet isolation statically.

5 Discussion

In this section, we discuss the expressiveness of our type system, the overhead it imposes on programmers, and its possible applications.

5.1 Expressiveness

The proposed type system does not significantly limit the expressiveness of Java Card: Almost all ordinary Java Card programs can be handled, possibly by introducing additional downcasts. This flexibility is due to the fact that `any` types are supertypes of the corresponding `intern` and `pep` types. Therefore,

variables that may hold references to objects in various contexts can be typed `any`, and casts can be used when such variables are read. In such situations, the expressiveness of the type system comes at the price of runtime checks. However, extra downcasts are only needed for two purposes: (1) when an internal object is stored in a static field and then read again (recall that static fields must not have `intern` types); (2) when an internal object is passed to a different context and then retrieved again (e.g., from a container in a different context).

The only pattern that cannot be typed in our type system is when a variable may hold a reference to a TEP or a non-TEP object. In such cases, the variable can neither be typed `tep` nor `any`. However, this situation is extremely uncommon since TEPs must not be stored in fields or arrays.

As presented in this paper, the tagged type system does not support contravariant subtyping, which prevents certain implementations that are admissible in Java Card. Assume that a class C inherits a method `void m(intern T p)` from its superclass, D, and implements a shareable interface, I, that declares `void m(any T p)`. Without tags, C would be a legal Java Card implementation, but C is forbidden by the tagged type system since it does not implement I's `void m(any T p)`. However, this is not a serious restriction: If D's implementation of `m` can handle parameter objects in other contexts, the parameter `p` should be declared `any`. Otherwise, C has to override the method anyway, and contravariant subtyping would allow C to widen the signature of `m` to `void m(any T p)`. Extending the tagged type system to contravariant subtyping w.r.t. tags is straightforward but omitted in this paper for simplicity.

5.2 Defaulting

The static safety of our type system comes at the price of some extra work for programmers, who have to add tags to their programs. However, for the majority of types, the tags can be determined easily. Except for static fields and program elements involved in the interaction with the JCRE or other applets, all tags are usually `intern`. Therefore, we can use `intern` as default tag for most types. More precisely, we default each untagged occurrence of a Java type T to the tagged type (δ, T), where δ is:

- `pep` if $T =$ `AID`;
- `tep` if $T =$ `APDU` or T is an exception class in the Java Card API;
- `any` if T is an interface extending `Shareable` or if the occurrence of T is in the declaration of a static field;
- `intern` otherwise.

These defaults reduce the overhead significantly. For instance, all tags of the Java Card API methods in Fig. 4 could be omitted. Although method `process` in Fig. 2 communicates with the JCRE and another applet, only the tags for the cast `(any Service)s` and the declaration `any Status sta` have to be specified manually. The fact that defaulting does not work in the latter case already indicates the error in the program. Usually, a well-formed applet does not hold references of type `any T` if `T` is *not* an interface that extends `Shareable`.

The cases in which defaulting is not sufficient are (1) the extra downcasts needed for reading static fields and (2) arrays of type `byte[]`, which are heavily used as internal objects and as global arrays for the `APDU` buffer.

An alternative to default tags would be type inference. Since inference has been applied to the complex context information in ownership type systems [2, 9], we assume that inference would be applicable here as well. We used defaulting since it works for modular programs.

5.3 Applications

In Section 4, we have shown that the type system can be used to check Java Card's applet isolation statically. The static context information can also be used to enforce stricter policies. For instance, an applet can easily be prevented from interacting with other applets by checking that no program element has tag `any` in the applet's code. Note that this policy cannot be enforced by just forbidding calls to `JCSystem.getAppletShareableInterfaceObject` since applets can also exchange references through static fields.

Our main motivation was to simplify the verification of source programs by checking applet isolation syntactically before verifying the program. Therefore, the type system is applied to source programs. However, the type system can easily be adapted to bytecode. An adapted bytecode verifier [16] could check applet isolation at load time. In that case, a modified virtual machine would only have to check applet isolation for downcasts from `any` types to `intern` or `pep` types. This would lead to a significantly faster program execution without weakening the security of the Java Card platform.

6 Related Work

The presented type system benefited from the work on ownership type systems [1, 2, 9, 11]. Like in these type systems, objects are grouped into contexts, and types approximate context information statically. However, ownership type systems provide hierarchic context structures, whereas the contexts in Java Card are flat. Like readonly references in the Universe type system [19, 17], the work presented here permits references between different contexts, but restricts the operations that can be performed on such references. Both Universes and the type system presented here use downcasts to specialize context information. Due to these commonalities, we expect that both type systems can be easily integrated into one type system that facilitates the verification of Java Card programs.

Most ownership type systems use owner parameters to keep track of the context an object belongs to. A similar mechanism could be useful in our work to provide more fine-grained context information than `any` tags do, and to make downcasts with dynamic context checks dispensable. However, references to SIOs are obtained through calls to `JCSystem.getAppletShareableInterfaceObject`. The most specific tagged result type for this method is `any Shareable` since it is not known statically from which context a reference is requested.

Similarly to Confined Types [8], Java Card provides one context per package. However, with Confined Types only the code in package P can modify objects in the context for P, whereas Java Card only uses the package structure to determine which applets share one context. The code that modifies the objects of an applet can reside in arbitrary packages.

Several static analyses for information flow between Java Card applets have been published. Bieber et al. [6, 7] present an approach that allows smart card issuers to verify statically by model checking that an applet satisfies a pre-defined security policy. This analysis is complementary to applet isolation. It is able to detect illicit information flow between several applets, whereas the applet firewall controls the interaction between two applets.

Caromel et al. [10] propose a dataflow analysis to infer context information statically. This information is then used to point programmers to potential firewall violations. Éluard and Jensen [13] combine a dataflow analysis with quantified conditional constraints to check more fine-grained sharing policies such as sharing between designated applets rather than all applets on a card. In contrast to dataflow analyses, the tagged type system allows programmers to record design decisions about applet sharing in the code, which serves as additional documentation and enables modular checking. Checking applet isolation based on dataflow analyses is too expensive to be performed on-the-fly by a virtual machine; our type system could be easily checked by a bytecode verifier.

The Java Card platform and, in particular, the applet firewall, have been formalized in different frameworks [5, 14]. These formalizations have been used to formally verify applet isolation and confidentiality properties [3, 4, 15]. With our type system, applet isolation can be mostly checked syntactically.

7 Conclusions

We presented a refined type system for Java Card that allows one to check applet isolation mostly statically. In theory, our type system can replace almost all dynamic firewall checks. However, unless all applets on a card are checked by our type system and a refined bytecode verifier, the dynamic checks have to stay in place to prevent applets from untrusted sources from violating the firewall. Still, the type system is useful to detect possibly fatal errors at compile time.

Our approach to checking applet isolation is complementary to formal verification of applet properties. Using this type system reduces the verification effort significantly since applet isolation does not have to be proved for each method call, field access, instanceof, etc. as it is the case in plain Java Card. On the other hand, verification techniques can be applied to prove that downcasts do not lead to `SecurityException`s. As future work, we plan to implement the type system in our verification tool Jive.

Acknowledgments

Piotr Nienaltowski provided us with helpful comments on a draft of this paper. We also thank the anonymous reviewers of this paper as well as the reviewers of

an earlier version, which was presented at the ECOOP 2001 workshop on Formal Techniques for Java Programs [18], for their valuable comments.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2004.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2002.
3. J. Andronick, B. Chetali, and O. Ly. Using Coq to verify Java Card applet isolation properties. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logic*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 2003.
4. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: A toolset for reasoning about JavaCard. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.
5. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A formal executable semantics of the JavaCard platform. In D. Sands, editor, *Programming Languages and Systems (ESOP)*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
6. P. Bieber, J. Cazin, A. El-Marouani, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. The PACAP prototype: a tool for detecting Java Card illegal flows. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2041 of *Lecture Notes in Computer Science*, pages 25–37. Springer-Verlag, 2001.
7. P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. *Journal of Computer Security*, 10(4):369–398, 2002.
8. B. Bokowski and J. Vitek. Confined types. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, 1999.
9. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. Doctor of philosophy, Electrical Engineering and Computer Science, MIT, February 2004.
10. D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java Card object sharing. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag, 2001.
11. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.
12. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 41–82. Springer-Verlag, 1999.
13. M. Éluard and T. Jensen. Secure Object Flow Analysis for Java Card. In *Proceedings of 5th Smart Card Research and Advanced Application Conference (Cardis'02)*, pages 97–110. USENIX, 2002.

14. M. Éluard, T. Jensen, and E. Denney. An operational semantics of the Java Card firewall. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 95–110. Springer-Verlag, 2001.
15. M. Huisman, D. Gurov, Chr. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: A case study. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 2984 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 2004.
16. X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.
17. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
18. P. Müller and A. Poetzsch-Heffter. A type system for checking applet isolation in Java Card. In *Formal Techniques for Java Programs*, 2001.
19. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
20. T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998.
21. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
22. D. von Oheimb and T. Nipkow. Machine-checking the Java specification: Proving type-safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*. Springer, 1998.
23. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.
24. Sun Microsystems, Inc. *The Runtime Environment Specification for the Java Card Platform, Version 2.2.1*, October 2003.

Verification of Safety Properties in the Presence of Transactions

Reiner Hähnle and Wojciech Mostowski

Department of Computing Science
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{reiner,woj}@cs.chalmers.se

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

1 Introduction

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

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

The experiments described in this paper were made with the KeY theorem prover, which is an interactive verification system for Java Card featuring a complete formalisation of atomic transactions [4]. It is part of the KeY system [1], an integrated tool for informal and formal development of object-oriented software described in Section 2.1. This paper makes the following contributions:

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 151–171, 2005.

- An experience report about the verification of parts of a JAVA CARD electronic purse application (*Demoney*) of realistic complexity [23]. The code includes atomic transactions. To our best knowledge, this is the first report on verification of JAVA CARD source programs with transactions without any simplification or abstraction. The case study and the experiments are described in Section 3.
- An assessment of current source code verification technology: what can be automatically proven in terms of LoC, complexity, etc.? Which desirable requirements can be expressed and which not? This is discussed in Section 4.1.
- An analysis of the limitations of current technology and how they can be overcome. We explain why the *Demoney* case study had to be partially refactored to make verification feasible. In particular, we make concrete suggestions towards overcoming the difficulties by *design for verification* in Section 4.2.

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

2 Background

2.1 The KeY Project

The work presented in this paper is part of the KeY project[1] [1]. The main goals of KeY are to (1) provide deductive verification for a real world programming language and to (2) integrate formal methods into industrial software development processes. For the first goal a deductive verification tool, the KeY Prover, has been developed. The verification is based on a specifically tailored version of Dynamic Logic – JAVA CARD Dynamic Logic (JAVA CARD DL), which supports most of sequential JAVA including the full JAVA CARD language specification. For the second goal we enhance a commercial CASE tool with functionality for formal specification and deductive verification. The design and specification languages of our choice are respectively UML (Unified Modelling Language) and OCL (Object Constraint Language), which is part of the UML standard. The KeY system translates OCL specifications into JAVA CARD DL formulae, whose validity can then be proved with the KeY Prover. All this is tightly integrated into a CASE tool, which makes formal verification as transparent as possible to the untrained user.

Of course, the use of OCL is not mandatory: logically savvy users of the KeY system can write their proof obligations directly in JAVA CARD DL and use its full expressive power. As we see later, this is even relatively straightforward.

2.2 JAVA CARD Dynamic Logic

We give a very brief introduction to JAVA CARD DL. We are not going to present or explain any of its sequent calculus rules. Dynamic Logic [28, 17] can be seen

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

as an extension of Hoare logic. It is a first-order modal logic with parametric modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of legal JAVA CARD statements). In the Kripke semantics of Dynamic Logic the worlds are identified with execution states of programs. A state s' is *accessible* from state s *via* p, if p terminates with final state s' when started in state s.

The formula $[p]\phi$ expresses that ϕ holds in *all* final states of p, and $\langle p \rangle \phi$ expresses that ϕ holds in *some* final state of p. In versions of DL with a non-deterministic programming language there can be several final states, but JAVA CARD programs are deterministic, so there is exactly one final state (when p terminates) or no final state (when p does not terminate). The formula $\phi \rightarrow \langle p \rangle \psi$ is valid if, for every state s satisfying precondition ϕ, a run of the program p starting in s terminates, and in the terminating state the postcondition ψ holds. The formula $\phi \rightarrow [p]\psi$ expresses the same, except that termination of p is not required, that is ψ needs only to hold *if* p terminates.

JAVA CARD DL is axiomatised in a sequent calculus to be used in deductive verification of JAVA CARD programs. The detailed description of the calculus can be found in [2]. The calculus covers all features of JAVA CARD, such as exceptions, complex method calls, atomic transactions (see below), JAVA arithmetic. The full JAVA CARD DL sequent calculus is implemented in the KeY Prover. The prover itself is implemented in JAVA. The calculus is implemented by means of so-called taclets [3], that avoid rules being hard coded into the prover. Instead, rules can be dynamically added to the prover. As a consequence, one can, for example, use different versions of arithmetic during a proof: idealised arithmetic, where all integer types are infinite and do not overflow, or JAVA arithmetic, where integer types are bounded and exhibit overflow behaviour [6].

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

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

It says that if the `card` object's `balance` attribute is equal to b in the initial state, then the execution of method `charge` with argument `amount` terminates normally (no exception thrown) and afterwards the `card` object's initial `balance` is increased by `amount`. The validity of this proof obligation under JAVA integer semantics depends on whether `charge()` accounts for overflow, the type of the $+$ operator, etc.

2.3 Strong Invariants

While working on one of the JAVA CARD case studies [27] it became apparent that the specification semantics based on the initial and final states of a program is not enough to specify and verify some JAVA CARD safety properties. It turned out that the JAVA CARD applet in question was not "rip-out safe": it is possible to destroy the applet's functionality by removing (ripping out) the JAVA CARD device from the card reader (terminal) while the applet on the card executes.

As a result of this the applet's memory may become corrupted and left in an undefined state, causing malfunctioning of the applet.

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

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

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

To introduce the notion of a strong invariant it was necessary to extend the JAVA CARD DL with a new modal operator $[\![\cdot]\!]$ ("throughout"), which closely corresponds to Temporal Logic's $\Box$ operator. In the extended logic, the semantics of a program is a sequence of all states the execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of intermediate states in traces of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the JAVA CARD DL calculus extended with additional sequent rules for the "throughout" modality [4].

2.4 JAVA CARD Atomic Transactions

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

The memory model of JAVA CARD differs somewhat from JAVA's memory model [12, 31–33]. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which is preserved between card sessions, and transient memory (RAM), whose contents disappears when power loss occurs, for example, when the card is removed from the reader. Hence, every memory location in JAVA CARD (variable or object field) is either persistent or transient. The

[2] In extended static checking a closely related concept called *object invariants* is used [21]. The semantics of OCL invariants is interpreted in the strong sense in [36], where a temporal extension of OCL is introduced.

JAVA CARD language specification gives the following rules (slightly simplified for this presentation): all objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Therefore, in JAVA CARD assignments such as "`o.attr = 2;`", "`this.a = 3;`", and "`arr[i] = 4;`" all have a permanent character; that is, the assigned values will be kept after the card loses power. A programmer can create an array with transient elements, but currently there is no possibility to make objects (fields) other than array elements transient. All local variables are transient.

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

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

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

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

A "throughout" property (formula) has to be checked after every single field or variable assignment which, according to the JAVA CARD runtime environment specification [32], is atomic. Such checks have to be suspended, however, when a transaction is in progress, because the assignments inside a transaction are not atomic, only the whole transaction is atomic. Moreover, as already said, each transaction can either finish successfully, in which case it commits all the conditional assignments, or it can fail and in that case the transaction is aborted and all the conditional assignments have to be rolled back. The logic has to account for the possibility of an abort and for the difference between persistent and transient data.

Observe that the possibility of an aborted transaction affects even the semantics of the standard modal operators $\langle\cdot\rangle$ and $[\cdot]$, because an abort affects the final state of the program. Details of how the extension of JAVA CARD DL that deals with transactions is handled in the calculus can be found in [4]. We do not repeat the technical solution in this paper, but we stress that the details are rather involved and surprisingly complex. The KeY Prover implements the whole extension of JAVA CARD DL with "throughout" and transaction mechanism. To our knowledge the KeY Prover is the only prover for JAVA CARD programs that fully handles JAVA CARD transactions.

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

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

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

3 Case Study: Java Card Electronic Purse

The case study presented here is based on the Java Card electronic purse application *Demoney* [23]. While *Demoney* has not all the features of a purse application actually used in production, it is provided by *Trusted Logic S.A.* as a realistic demonstration application that includes all major complexities of a commercial program.

Our target program is a somewhat refactored fragment of *Demoney* and concentrates on the important aspects of the application to highlight our verification results. The *Demoney* source code is at present not publicly available, and we do not show it. The program we verified is, however, very close to *Demoney* and follows the *Demoney* specification [23]. We deviate from *Demoney* mainly in that our program is designed to make verification simpler. We discuss these issues in detail in Section 4.2.

The safety properties that we discuss here were directly motivated by the ones described in [24]. In fact the property we prove (that the current balance of the purse is always in sync with the balance recorded in the most recent log entry) for the `processSale` method presented in Section 3.4 is exactly the one described in [24, Section 3.5]. The example mentioned there is also based on the *Demoney* application.

3.1 The LogRecord Class

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

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

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

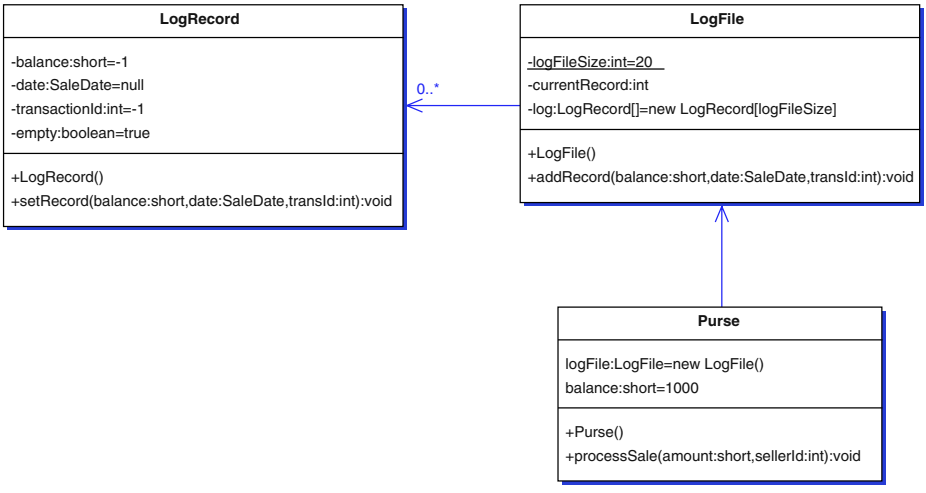


Fig. 1. `Purse` application class diagram

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

3.2 Specification and Verification of setRecord

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

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

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

```
  !self = null
& balance >= 0 & !date = null & transId > 0
& (self.empty = FALSE ->
    (self.balance >= 0 & !self.date = null & self.transactionId > 0))
```

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

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

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

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

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

3.3 The Purse Class

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

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

3.4 Specification and Verification of processSale

As stipulated in [24], we need to ensure consistency of related data. In our case, this means to express that the state of the log file is always consistent with the

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

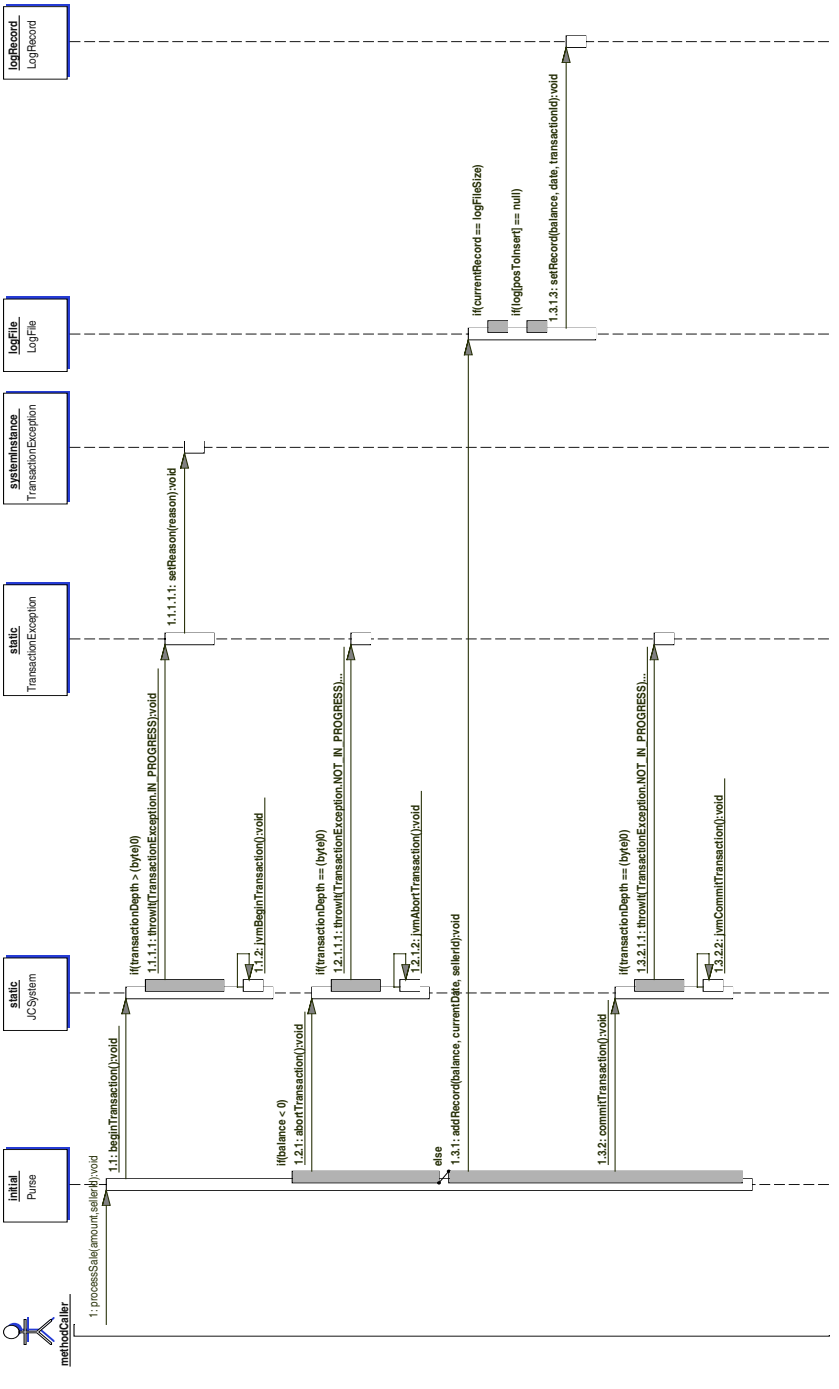


Fig. 2. Sequence diagram of the `processSale` method

current state of the purse. More precisely, we state that the current balance of the purse is always equal to the balance stored in the most recent entry in the log file. The corresponding strong invariant expressed in pseudo OCL is:

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

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

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

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

3.5 *Post Hoc* Verification of Unaltered Code

We just reported on successful verification attempts of a refactored and partial version of the *Demoney* purse application. When it comes to capabilities and theoretical features of the KeY Prover there is nothing that prevents us in principle from proving properties about the real *Demoney* application. There are, however, some design features in *Demoney* that make the verification task difficult. We discuss them in detail in Section 4.2.

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

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

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

3.6 Performance

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

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

Proof Obligation	Time (sec.)	Steps	Branches
$[\![\texttt{setRecord}]\!]$	2.0	230	20
$\langle\texttt{setRecord}\rangle$	1.5	125	6
$[\![\texttt{setRecord}]\!]^{\mathrm{F}}$	2.1	248	6 open
$\langle\texttt{keyNum2tag}\rangle^{\mathrm{D}}$	3.3	392	18
$\langle\texttt{keyNum2keySet}\rangle^{\mathrm{D}}$	5.5	640	33
$[\![\texttt{processSale}]\!]^{1}$	41.4	3453	79
$[\![\texttt{processSale}]\!]^{2}$	51.3	4763	248
$[\![\texttt{processSale}]\!]^{3}$	111.1	7264	338

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

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

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

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

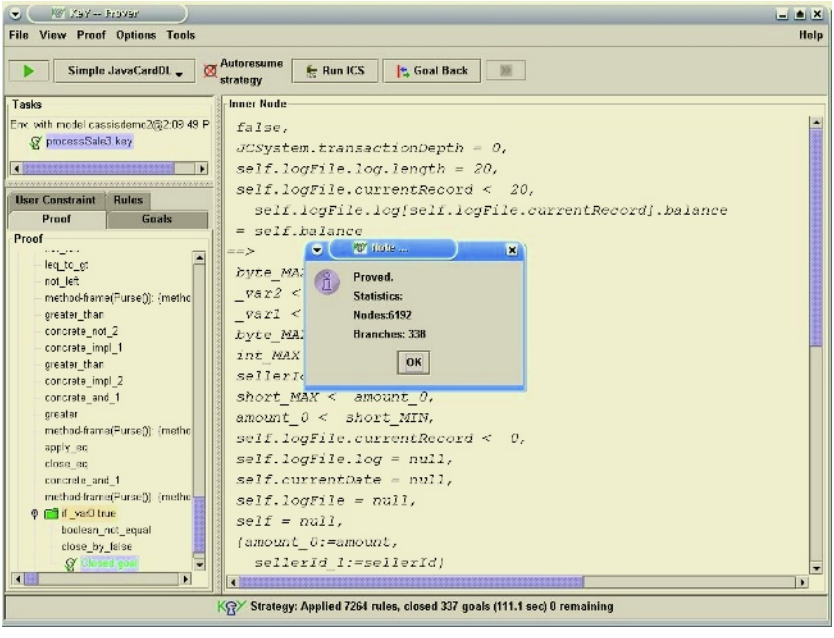


Fig. 3. KeY Prover window with successful proof

4 Results

4.1 Verification Technology

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

The largest example involves about 30 lines of source code. This may not seem much, but it clearly indicates that methods and classes of non-trivial size can be handled. In addition, the next version of the KeY prover will support composition of proofs including a treatment of representation exposure by computation of modifier sets [7]. Consequently, we expect that formal verification of Java Card programs comparable to *Demoney* is achievable before long.

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

It would be desirable to have a more formal statement here relating types of programs and proof complexity. The problem is that even loop-free Java Card programs contain control structures like exceptions and transactions that have a

global effect on control flow. Taking away all critical features yields an uninteresting programming language, while leaving them in renders general statements on proof complexity (at least the ones we could think of) simply untrue.

A principal obstacle against automating program verification is the necessity to perform induction in order to handle loops (and recursion). In most cases, the induction hypothesis needs to be generalised, which requires considerable user skill. There is extensive work on automating induction proofs, however, mostly for simple functional programming languages as the target. Only recently, preliminary work for imperative target languages [29, 16] appeared. If, however, *Demoney* is a typical JAVA CARD application, then loops might be much less of a problem than thought: of 10 loops in *Demoney* (9 `for`, 1 `while`) most are used to initialise or traverse arrays of known bounds. Such loops do not require induction at all. The next version of the KeY Prover contains a special automated rule for handling them. Our analysis showed that at most one loop in *Demoney perhaps* needs induction. There is no recursion.

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

An important question is whether we are able to express all relevant requirements. There is no agreement on standard requirements for JAVA CARD, but the report [24] can serve as a guideline. Many of the security properties related there can be expressed in JAVA CARD DL including strong invariants. In the present paper we concentrated on data consistency in connection with atomic transactions. The examples included also overflow control. In [14] it was shown that also information flow properties are expressible. We have strong evidence that also memory allocation control, error control and even the well-formedness of transactions can be formulated. For example, the following two properties, taken from [24] can be formulated in JAVA CARD DL: (i) no `TransactionException` related to well-formedness is thrown, (ii) only `ISOExceptions` are thrown at the top level of an applet.

The main limitation of the currently used version of JAVA CARD DL is the impossibility to express complex temporal relationships between the execution of different code fragments to establish advanced control flow properties such as a certain temporal order on method calls. This requires more complex temporal operators than "throughout" or some kind of event mechanism, and is a topic for future research. On the specification side, some work was done in [34], while [8] looked at abstracted byte code in a model checking framework.

4.2 Design for Specification and Verification

The way *Demoney* is designed and coded causes certain technical complications both when specifying and proving safety properties of programs with transactions. We demonstrate two issues and discuss their impact on the process of

specification and verification of JAVA CARD programs. Thereby, we give guidelines for the design of JAVA CARD applications to avoid such problems.

Byte Arrays. Following the specification in [23, p. 17] *Demoney* implements a cyclic log file in a very similar fashion to our `Purse` class. *Demoney* stores more information than our program in a single log record, but that's not an issue when it comes to formal verification. The major difference is that each single log record is implemented as a `byte` array instead of an object (of class `LogRecord` in our case). We suspect that the main reason for implementing a log record as a `byte` array is to ease the transportation of log data to the card terminal. Another reason, explicitly mentioned in the specification, is to follow the schema of recording data in the form of TLVs (Tag-Length-Value). Finally, because of memory costs in smart cards, `byte` arrays are still much used to save some small memory overhead one has to pay for object instances and booleans[5].

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

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

one has to say

```
Object[] logFile;
```

and then allocate this data structure by saying:

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

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

```
LogRecord[] logFile;
```

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

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

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

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

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

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

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

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

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

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

This specification expression is based on an educated guess of how the Java Card API method `Util.setShort` [31] is implemented (`setShort` is a native method and its implementation is not disclosed). Expanding the Dynamic Logic function symbol `castToByte` results in another modulo operation. Also note that all arithmetic function symbols have Java types and must be checked against overflow. Proving with expressions such as the one shown above is difficult, if not practically unfeasible.

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

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

One general guideline would be to use object types to store any kind of non-primitive data, at least if they are persistent (for transient data there is no choice but an array in Java Card). Furthermore, serialise objects only if necessary (in case of Java Card for communication). As part of a bigger picture one should consider to decouple application functionality from the communication

model. Such a decoupled design is likely to allow decomposable, and thus easier, verification. It is more robust, too. We point to the fact that the examples presented in [24] follow for the most part the guideline of using object types instead of `byte` arrays for storing data.

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

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

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

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

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

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

where all arithmetic function symbols have JAVA types and must be checked against overflow. This is certainly not impossible, but it adds substantially to the complexity of the resulting first-order proof obligations and, in connection with other phenomena, can make the problems too difficult to prove automatically.

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

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

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

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

Discussion. Asking a programmer to rewrite the code to ease verification may seem unrealistic. It may look as if we put the burden of making verification feasible on the programmer instead of enabling the prover handle arbitrarily complex programs. This is not the case. Our aim is to make the KeY prover powerful enough to deal with complex JAVA CARD code, however, one cannot expect a prover to deal with baroque programs optimised for performance. A trade-off

has to be found. The guidelines we proposed are simple to follow and, in addition, make sense from a software engineering point of view. In particular, we do not assume that the programmer has any knowledge of the theorem prover.

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

5 Related Work

A version of Dynamic Logic that extends pure Dynamic Logic with trace modalities "throughout" and "at least once" was first presented in [5]. The axiomatisation of transactions was provided in [4]. Paper [18] proposes another approach to reasoning about rip-out properties (called card tears there). It presents a theoretical framework for dealing with card tears and transactions based on global program (method) transformation (as opposed to the KeY approach of local transformations). This paper does not report on any practical verification attempts. In [34] temporal constructs are introduced to the JAVA Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Paper [19] is closely related to our work in the sense that it reports on successful verification attempts of a commercial JAVA CARD applet with different verification tools (ESC/JAVA2, JIVE, KRAKATOA, LOOP). The security property under consideration, also mentioned in Section 4.1, is that only `ISOExceptions` are thrown at the top level. Transaction related properties are not investigated. Like in the present study, it is stressed that two-dimensional `byte` arrays and the use of `byte` arrays in general are problematic in JAVA CARD verification, and have serious impact on the size and complexity of proofs. One of the main results is that subtle bugs were found in the applet.

GemPlus provides a JAVA CARD case study similar to *Demoney* [10], also a purse application and publicly available[6]. We do not use it at the moment, because it contains a large number of features that detract from the basic issues and make it less suitable as a starting point for JAVA CARD verification. In addition, it was not developed further in the last three years.

Related work in JAVA CARD verification can be classified according to several criteria. Working on byte code avoids the problems of source code availability and compiler trustworthiness, but makes full verification more difficult due to information loss during compilation. An overview of work done on the byte code level is provided in [9] – we concentrate on efforts targeted at source code: here, one can distinguish between methods that attempt complete modelling of the

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

JAVA CARD semantics and those that do not. The latter include model checking and extended static checking.

Model checking is based on a suitable abstraction of the execution model, which in the Bandera project [13] is JAVA, and of the requirements. The advantages are full automation of the model checking phase, trace generation for counter models, and treatment of concurrent JAVA programs. The drawback is the need for abstraction which poses difficulties for programs containing JAVA arithmetic and other inductive data structures. Bandera handles JAVA, not JAVA CARD, and hence no transactions. In design-by-contract [25] and extended static checking (ESC) [15] JAVA source code is decorated with annotations from a restricted language. Annotated programs (via an intermediate representation) undergo a dynamic analysis that produces first-order verification conditions for a theorem prover. The analysis does not attempt to be complete, but it is fully automatic and produces warnings, when annotations are potentially violated. ESC is related to our strong invariants, because arbitrary code locations can be annotated with object invariants [21]. An approximation of strong invariants within ESC can be obtained by annotating every program point with the desired invariant[7]. Again, atomic transactions are not supported, as the target language is JAVA.

Closest to our approach are source code verifiers for JAVA based on various program calculi. The LOOP tool [20] translates JAVA source code with JML specifications into theories for the PVS theorem prover. JAVA semantics is described with co-algebras and uses higher-order logic as an internal representation. Higher-order logic is also used to formalise syntax and semantics of a JAVA fragment in Isabelle [35] and in the KRAKATOA tool [22]. In the latter JAVA programs and their JML specifications are translated into an intermediate, mostly functional, language, then proof obligations are generated, which in turn are proved with the COQ proof assistant. The JIVE system [26] is based on an extended Hoare style calculus, Jack [11] on weakest precondition calculus, and KIV [30] on Dynamic Logic. The last three systems are closely related to the KeY Prover in that they all axiomatise JAVA with logical rules that can be seen as a small step operational semantics and proofs can be interpreted as symbolic execution with induction. The differences lie in the details and scope of the axiomatisation as well as support for automation. As far as we know, KeY is the only system that supports strong (object) invariants and, in particular, the semantics of JAVA CARD transactions.

6 Conclusions

In this paper, we presented and analysed a case study concerned with formal specification and verification of JAVA CARD programs. Our results show that largely automated formal verification of realistic JAVA CARD applications without abstraction is possible in the near future. It is possible already now provided that applications are designed with verification in mind from the start. We gave a

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

number of simple design guidelines that drastically simplify proofs while creating only a moderate performance overhead. We believe this to be acceptable, because even in the smart card world, performance restrictions become less of an issue. Besides, a small memory overhead seems an acceptable price for provably correct programs.

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

Acknowledgements

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

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and Systems Modeling*, April 2004. Online First issue, to appear in print.
2. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Revised Papers, Java on Smart Cards: Programming and Security, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
3. B. Beckert, M. Giese, E. Habermalz, R. Hähnle, A. Roth, P. Rümmer, and S. Schlager. Taclets: a new paradigm for constructing interactive theorem provers. *Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales, Serie A: Matemáticas*, 98(1), 2004. Special Issue on Symbolic Computation in Logic and Artificial Intelligence.
4. B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In M. Pezzè, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE) Conference*, volume 2621 of *LNCS*, pages 246–260, Warsaw, Poland, April 2003. Springer-Verlag.
5. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *LNCS*, pages 626–641. Springer-Verlag, 2001.
6. B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In E. A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, volume 2999 of *LNCS*, pages 207–226. Springer, April 2004.

7. B. Beckert and P. H. Schmitt. Program verification using change information. In *Proceedings, Software Engineering and Formal Methods (SEFM), Brisbane, Australia*, pages 91–99. IEEE Press, 2003.
8. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Checking secure interactions of Smart Card applets. *Journal of Computer Security*, 10(4):369–398, 2002.
9. R. Boyer. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, Amsterdam, 2003.
10. E. Bretagne, A. E. Marouani, P. Girard, and J.-L. Lanet. PACAP purse and loyalty specification v0.4. Technical report, GemPlus, January 2001.
11. L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In *Proc. Formal Methods Europe, Pisa, Italy*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
12. Z. Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.
13. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *Proc. SPIN Software Model Checking Workshop*, LNCS, pages 205–223. Springer-Verlag, 2000.
14. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. Technical Report 2004-01, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004.
15. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
16. R. Hähnle and A. Wallenburg. Using a software testing technique to improve theorem proving. In A. Petrenko and A. Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES), Montréal, Canada*, volume 2931 of *LNCS*, pages 30–41. Springer-Verlag, 2003.
17. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
18. E. Hubbers and E. Poll. Reasoning about card tears and transactions in Java Card. In *Fundamental Approaches to Software Engineering (FASE'2004), Barcelona, Spain*, volume 2984 of *LNCS*, pages 114–128. Springer, 2004.
19. B. Jacobs, C. Marché, and N. Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology And Software Technology*, volume 3116 of *LNCS*, Stirling, UK, July 2004. Springer.
20. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspective. Technical report, University of Nijmegen, 2003. NIII Technical Report NIII-R0316. To appear in the proceedings of International Symposium on Software Security (ISSS 2003).
21. K. R. M. Leino and R. Stata. Checking object invariants. Technical Note #1997-007, Digital Systems Research Center, Palo Alto, USA, January 1997. Available from `ftp://ftp.digital.com/pub/DEC/SRC/technical-notes/SRC-1997-007.ps.gz`.
22. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/Java Card programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.
23. R. Marlet and C. Mesnil. Demoney: A demonstrative electronic purse – Card specification. Technical Report SECSAFE-TL-007, Trusted Logic S.A., November 2002.

24. R. Marlet and D. L. Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic S.A., August 2001.
25. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, October 1992.
26. J. Meyer, P. Müller, and A. Poetzsch-Heffter. The Jive system—implementation description. Available from `http://softech.informatik.uni-kl.de/old/en/publications/jive.html`, 2000.
27. W. 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. Available from `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.
28. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.
29. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial loop invariants for imperative programs. Available from `http://www.lsi.upc.es/~erodri/ijcar04ex.ps`, November 2003.
30. K. Stenzel. Verification of Java Card Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001.
31. Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface*, 2002.
32. Sun Microsystems, Inc. *Java Card 2.2 Runtime Environment Specification*, 2002.
33. Sun Microsystems, Inc. *Java Card 2.2 Virtual Machine Specification*, 2002.
34. 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.
35. D. von Oheimb. *Analyzing Java in Isabelle/HOL*. PhD thesis, Institut für Informatik, Technische Universität München, January 2001.
36. P. Ziemann and M. Gogolla. An OCL extension for formulating temporal constraints. Technical Report 1/03, Universität Bremen, Fachbereich für Mathematik und Informatik, 2003.

Modelling Mobility Aspects of Security Policies

Pieter Hartel, Pascal van Eck, Sandro Etalle, and Roel Wieringa

Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{pieter,vaneck,etalle,roelw}@cs.utwente.nl

Abstract. Security policies are rules that constrain the behaviour of a system. Different, largely unrelated sets of rules typically govern the physical and logical worlds. However, increased hardware and software mobility forces us to consider those rules in an integrated fashion. We present SPIN models of four case studies where mobility plays a role. At present our models are ad-hoc. In each case the model captures both the system of interest and its security policy. The model is then formally checked against a security principle. The model checking activity shows examples of policies that are too weak to cope with mobility.

1 Introduction

Security policies are important both in the physical and the logical world. By a *policy* we mean "a rule that defines a choice in the behaviour of a system" [1]. A *security* policy rules out behaviour "that has been deemed unacceptable" [2]. A *spatial* security policy constrains this further by ruling out behaviour tied to particular locations [3]. Physical events may affect logical security and vice versa. For example a user moving a laptop with access to confidential data from a secure to an insecure environment must loose all authorisations on the data immediately. The problem we intend to solve is finding a means to analyse security policies from the point of view of logical and physical mobility. The urgency of solving this problem is caused by increased mobility. For example (small) mobile computing equipment is easily carried out of a building, past unsuspecting security guards, regardless of any measures like encryption, or tamper resistance. Mobility thus ties logical and physical security together, causing new and, as we will show, unanticipated security problems to arise.

We propose a first step towards an analysis method, with initial tool support. The method is based on developing a formal model of a system with its security policy, and on using a model checker to analyse the combination. From the analysis we are able to predict the occurrence of security problems that would not have occurred without mobility. We present case studies from four different domains to illustrate the method.

In our work, we do not distinguish between logical mobility (mobile code that roams from one processing and networking context to another) and physical mobility [4] (non-mobile code on mobile hardware that is carried from one network context to another by its user). In both cases, the execution context of

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 172–191, 2005.

some code under study changes, which is the level of abstraction at which we work.

Our method consists of manually creating (1) an abstract model of a system of interest, (2) its security policy and (3) a security design principle that the combination of (1) and (2) should satisfy.

The system model should be able to generate all possible behaviours of the system of interest, including behaviours of hacked versions of the system. For example consider the UNIX ping command, which normally makes a characteristic sequence of system calls. Hacked versions of ping would make different sequences of system calls.

The security policy should constrain the behaviours of the system model to desirable (safe) behaviours. For example, an execution monitoring system might allow ping to make a raw socket call (which requires root permission), but no other potentially dangerous systems calls.

The security principle finally constrains the joint behaviour of the system and its security policy to something the system designer would have had in mind when creating the system. In the case of ping this might be the principle of the least privilege, which would ensure that root permission is only used once, for the raw socket call.

We express our system, policy and principle in Promela, the input language of the SPIN model checker [5] as follows:

A System of interest is characterised by a trace of relevant events, for example system calls, or messages across a network. This is modelled in a most general way to capture only the essence of the system, particularly its mobility aspects. The system is modelled in SPIN by global data, channels and processes.

A Security Policy constrains the behaviour of the system to acceptable (safe) behaviour. The policy constrains traces of events using the same terms as the system, e.g. system calls, network messages etc. The policy is modelled by one or more processes.

A Security Principle is a design guideline for the system and the policy. To make the principle operational we translate it into a specification for the system and its policy, which, in our case studies, takes the form of a SPIN trace declaration (See [5, Page 485] for the technical reason why we use trace declarations rather than LTL formulae).

The system, policy and principle satisfy:

$$(system \parallel policy) \models principle$$

We use SPIN to analyse a system and policy with respect to a principle; a system and policy that does not abide by the principle gives rise to concrete counter examples. We demonstrate the approach by presenting four case studies in subsequent sections. In each case we identify system, policy, and principle. SPIN then shows that no deadlock can occur in $(system \parallel policy)$. This shows that we are working with sensible models that admit behaviour satisfying the

security policy. However, model checking $(system \parallel policy) \models principle$ gives traces showing that the principle can be violated. In all but the first case study (which is intended as an introductory example) the violations can be attributed to mobility.

Our modelling methodology is based on a combination of techniques developed by Alan Mycroft and his group from Cambridge, UK, for dynamic security policies in the logical domain (security policy abstraction [6]) and the physical domain (spatial security policy [3]). We extend the Cambridge work in the logical domain [6] by adding mobility considerations. Both works from Cambridge propose policy languages but no realised tool support. The inspiration for using SPIN by way of tool support comes from the work of Cheng et al [7], who also use SPIN to model check security policies, however without consideration of either mobility or the physical domain. A third source of inspiration is formed by the work of Sekar et al [8], who present a system which automatically derives models from code. Model checking is then used to analyse models with respect to a choice of security policies. Sekar et al do not consider physical mobility.

The four subsequent sections discuss one case study each. The final section concludes and discusses further work.

2 Ping

The first case study concerns the UNIX utility ping, which sends an IP packet to a network node, reporting on the availability of the destination and on the performance of the connection. Making the connection requires root permission (for the socket call), which is potentially dangerous and should thus be minimised. This is captured by the principle of the least privilege, which states that [9]: "Every program and every user of the system should operate using the least set of privileges necessary to complete the job". In the case of ping (and other commands) the principle translates into *dropping root permission after executing the first socket system call.*

This case study shows that the translation of a general principle such as least privilege to the case at hand is relative to the design charter that is given a-priori. In this case study, we assume that the design of the ping utility is within the design charter, while the design of the UNIX operating system is not. Consequently, it is a given that some system calls needed in the ping utility need root permission, which results in the translation of the principle of least privilege as given above. If the design of UNIX itself would have been within the design charter of this case study, most probably a solution would have been chosen in which the ping utility would not need root permission.

2.1 The System

We model behaviour by traces of system calls, which is abstract because we ignore the parameters of the system calls and any associated calculations. At the same time, naming system calls is concrete because we distinguish system calls

that do not need root permission, and which could be collapsed. The Promela enumeration type `mtype` below mentions the same system calls as used by Madhavapeddy et al [6].

```
mtype = {
   LibC_exit, LibC_gethostbyname, LibC_gettimeofday, LibC_printf, SysCall_brk,
   SysCall_mprotect, SysCall_recvfrom, SysCall_sendto, SysCall_sigaction, SysCall_socket
} ;
```

We use a synchronous channel to connect the ping system to the ping policy. A message consists of one of the symbols `LibC_exit` ... `SysCall_socket`.

```
chan c = [0] of {mtype} ;
```

The ping system is modelled abstractly by the process `ping_system`. The behaviour generated encompasses all possible traces of the ten system calls in the enumeration type `mtype`. This represents considerably more behaviour than real ping commands would exhibit and includes for example hacked versions of ping. This gives us a good model of reality.

```
active proctype ping_system() {
   do
   :: c!LibC_exit                          :: c!LibC_gethostbyname
   :: c!LibC_gettimeofday                  :: c!LibC_printf
   :: c!SysCall_brk                        :: c!SysCall_mprotect
   :: c!SysCall_recvfrom                   :: c!SysCall_sendto
   :: c!SysCall_sigaction                  :: c!SysCall_socket
   od
}
```

2.2 The Policy

The ping policy below represents an abstract version of the security policy described by Madhavapeddy et al [6]. (We have abstracted away from the fact that extra `brk` and `printf` system calls are always allowed.)

We define C pre-processor macros corresponding to the language constructs of the same name proposed by Madhavapeddy et al [6]. (The backward slashes at the end of each line except the last ensure that the macro definition extends across multiple lines.)

```
#define optional( x ) \
   if \
   :: x \
   :: skip \
   fi
```

```
#define multiple( x ) \
   do \
   :: x \
   :: break \
   od
```

The first non-deterministic choice below represents a call to ping that responds with a usage message. The second choice represents ping doing its proper work, i.e. a socket call, optionally followed by a call to `LibC_gethostbyname` etc.

```
active proctype ping_policy() {
   do
   :: c?LibC_printf ;
      c?LibC_exit
   :: c?SysCall_socket ;
      optional(c?LibC_gethostbyname) ;
      c?LibC_printf ;
      multiple(c?SysCall_sigaction) ;
      multiple(c?SysCall_sendto ; c?SysCall_recvfrom ; optional(c?SysCall_brk)) ;
   od
}
```

To understand how the ping system and the policy interact, compare the send actions `c!...` of `ping_system` to the receive actions `c?...` of `ping_policy`. This comparison reveals that the former is prepared to engage in any send action, whereas the latter is prepared only to engage in specific receive actions. This then explains how the policy constrains the generic behaviour of the system. We use the same technique throughout the paper to effectuate the security policies.

The separation of system and policy thus provides a convenient way to talk about the system (which is general) and the policy (which constrains general behaviour to specific behaviour). To indicate that this not an entirely trivial result we point out that many other ping implementations are possible that cannot be constrained to conform to the policy, for example a version of `ping_system` where the `do ... od` above would be replaced by `if ... fi`.

The security policy still permits attacks, for example mimicry attacks [10], which subtly alter the pattern of system calls to achieve malicious intent while avoiding detection by an IDS. Our use of a security principle below also captures this type of attack.

2.3 The Principle

The ping policy must satisfy the principle of the least privilege. This is mentioned but not explicitly specified as such by Madhavapeddy et al [6]. As stated before, the principle is interpreted as ping must drop root permission after one socket call, which we model here by saying that one socket call is ok, but not two. This is captured by the trace declaration below, which matches a trace that has exactly one `SysCall_socket`. When model checking, SPIN tries to find a sample trace that does not match the trace declaration, which is then a counter example for the desired principle.

```
#define anything_but_socket() \
      c?LibC_exit
   :: c?LibC_gethostbyname \
   :: c?LibC_gettimeofday
   :: c?LibC_printf \
   :: c?SysCall_brk
   :: c?SysCall_mprotect \
   :: c?SysCall_recvfrom
   :: c?SysCall_sendto \
   :: c?SysCall_sigaction
```

```
trace {
   do
   :: anything_but_socket()
   :: c?SysCall_socket ->     break
   od ;
   do
   :: anything_but_socket()
   od
}
```

2.4 Analysis

Model checking the parallel composition of `ping_system`, `mobility` as well as `ping_policy` against the built in formulae of the SPIN model checker (absence of deadlock) shows no errors. This demonstrates that the model is indeed a sensible one because behaviours are possible that satisfy the security policy.

Model checking the system and the policy against the principle reveals traces that do not match the trace declaration, i.e. traces that violate the principle. A concrete example is `SysCall_socket`; `LibC_printf`; `SysCall_socket`. To remedy the situation we have several options. For example we could replace `do ... od` in the ping policy by `if ... fi`, because then only one socket call would result.

A better alternative would be to exit the loop once the second non-deterministic choice has completed, because this allows an arbitrary number of 'usage' message to be generated but one 'proper' ping call.

The results seem trivial but we should like to point out that our system model and security policy are indeed representative for current intrusion detection systems. Therefore it is encouraging that SPIN has been able to discover a problem with our system and policy.

3 Database Application

The second case study investigates the separation of testing and production environments in a modern central database application. The application consists of three layers: data, business logic and presentation. The data and business logic layers reside on a central server. There are two datasets: one with production data and one with randomised test data. The business logic layer accepts network connections from the presentation layer, which consists of Java applets available throughout the organisation. Upon accepting a connection from a presentation layer client, the business logic layer should check the physical location of the client in a configuration database and depending on this location, use either the production or test dataset. We treat the presentation layer and the data base layer as the system, and the business logic layer as the policy because the business logic layer decides what constitutes acceptable behaviour.

3.1 The System

We define the symbols necessary for our example. `Test_0` represents the test data, `Data_0` ... `Data_3` represent production data. The symbols `Test_Env` and `Production_Env` identify the test and production environments, and `Connect` ... `Reply` represent commands exchanged between the three layers of the system.

```
mtype = {
   Test_0, Data_1, Data_2, Data_3, Test_Env, Production_Env,
   Connect, Disconnect, Request, Reply
}
```

The model comprises three processes (representing the presentation, business logic and database layers) and two synchronous channels connecting the layers. `p2b` connects the presentation to the business logic layer and `b2d` connects the business logic layer to the database layer. The message header may be `Connect` ... `Reply`. Depending on the message header, the message body may be one of `Test_0` ... `Data_3` or `Test_Env` or `Production_Env`.

```
chan p2b = [0] of {mtype, mtype} ;
chan b2d = [0] of {mtype, mtype} ;
```

The presentation layer process below generates an almost arbitrary sequence of requests for both production and test environments. The only form of protocol

obeyed is that the presentation layer insists that it receives a reply after each request. The database layer reports test or production data as appropriate.

```
active proctype presentation_layer() {
  mtype data ;
end:
   do
   :: p2b!Connect(Test_Env)
   :: p2b!Connect(Production_Env)
   :: p2b!Request(Test_Env) ->
      p2b?Reply(data)
   :: p2b!Request(Production_Env) ->
      p2b?Reply(data)
   :: p2b!Disconnect(Test_Env)
   :: p2b!Disconnect(Production_Env)
   od
}
```

```
active proctype database_layer() {
   mtype data ;
end:
   do
   :: b2d?Request(Production_Env) ->
      if
      :: data = Data_1 ;
      :: data = Data_2 ;
      :: data = Data_3
      fi ;
      b2d!Reply(data)
   :: b2d?Request(Test_Env) ->
      b2d!Reply(Test_0)
   od
}
```

3.2 The Policy

The business logic layer process mediates between the presentation and the database layers, ensuring behaviour consistent with the business rules. In particular the business logic layer insists that applications make data base requests only when connected to the data base. Each request is passed on to the database layer, indicating whether to use the test or production data base. (The function `eval` ensures that the body of the `Reply` message matches exactly with the current value of `data`).

```
active proctype business_logic_layer() {
   mtype env, data ;
end:
   do
   :: p2b?Connect(env) ->
      do
      :: p2b?Request(env) -> b2d!Request(env) ; b2d?Reply(eval(data)) ; p2b!Reply(data)
      :: p2b?Disconnect(env) -> break
      od
   od
}
```

3.3 The Principle

The principle of interest is that "Testing and production environments are physically separated". This is enforced by checking that once connected from a particular environment, all following request messages emanate from that same environment until a disconnect message arrives, again from the same environment. An implementation that checks the location once for each connection suffices in the case of a fixed network. In the case of a mobile network that supports roaming, this does not suffice.

```
trace {
   do
   :: p2b?Connect(Test_Env) ->
      do
      :: p2b?Request(Test_Env) ->          p2b?Reply(_)
      :: p2b?Disconnect(Test_Env) ->       break
      od
```

```
   :: p2b?Connect(Production_Env) ->
      do
      :: p2b?Request(Production_Env) ->    p2b?Reply(_)
      :: p2b?Disconnect(Production_Env) -> break
      od
   od
}
```

3.4 Analysis

Model checking the system and the policy against the principle reveals (as expected) that the presentation layer is ill behaved because message sequences with arbitrary test and production environment parameters are generated. The following trace gives a concrete counter example for our principle: `Connect(Test_Env)`, `Request(Production_Env)`.

The problem lies in the business logic layer (the policy), which should constrain the presentation layer to correct behaviour. Incorporating the principle in the form of a trace declaration points out this deficiency of the business logic layer. The business logic layer might implement the principle using `eval(env)` instead of plain `env` in the receive actions for `Request` and `Disconnect`. This would constrain the environment of the call to match the value of the `env` variable exactly.

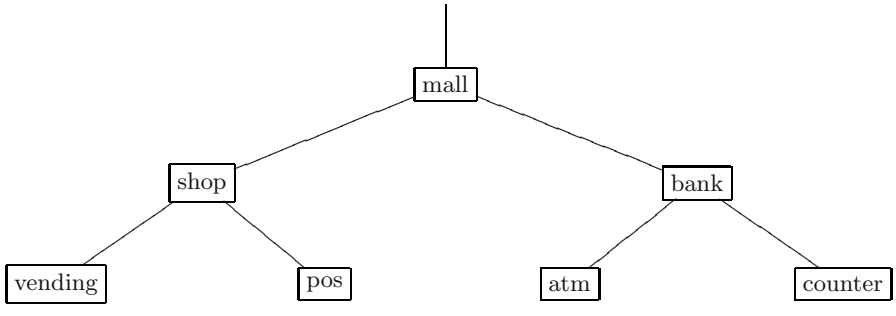


Fig. 1. Mall with a vending machine and a point of sale terminal (pos) at the shop, and an automated teller machine (atm) and the traditional counter at the bank.

4 Smart Cards

The third case study investigates the behaviour of next generation smart cards, which are the object of study in the European Inspired project[1]. The principle of interest here is *Be reluctant to trust.* To operationalise this principle we propose security policies that can be customised by the card issuer as well as the card holder. The concrete example that we will study is of (1) a card holder who states that she *does not permit applets to be loaded (i.e. smart card management)*

[1] http://www.inspiredproject.com

other than when she is at the bank and (2) a card issuer who states that *a payment transaction at a vending machine must always be followed by a loyalty transaction*. Using a smart card that operates under the Be reluctant to trust principle would give the user trust in the system because the user can hold the bank responsible for all the applets on the card. This would also mean that a shop could not load a loyalty applet unless the permission to load applets is explicitly delegated to the shop.

4.1 The System

Following Madhavapeddy et al [6] we assume a tree-shaped world model. Figure 1 shows a shopping mall at the root of the tree, a shop and a bank are at the intermediate layer, and four smart card readers at the leaves.

We use synchronous channels to model the identity of the (physically separated) parties. Three channels connect to the physical locations, four to the smart card readers and another three to the applets. The last three are shown here, the former are defined in the macros `run_leaf` and `run_node` below.

```
chan loyalty = [0] of {chan} ;
chan payment = [0] of {chan} ;
chan management = [0] of {chan} ;
```

We need two kinds of processes: one type representing interior nodes (embedded in macro `run_node`) and one for leaf nodes (macro `run_leaf`). (The macros generate both a channel with the given name, for example `vending`, and corresponding process `proc_vending`.) To model mobility, each process allows an applet to travel from any of its input channels to any of its output channels. In the node process this means that an applet can be received either from the parent of the node, or from one of the children. Similarly, the applet can be moved on to the parent or one of the children.

```
#define run_node(parent, left, right) \
chan parent = [0] of {chan} ; \
active proctype proc_##parent() { \
   chan applet ; \
end: \
   do \
   :: if \
      :: parent?applet   :: left?applet   :: right?applet \
      fi ; \
      if \
      :: parent!applet   :: left!applet   :: right!applet \
      fi \
   od \
}
```

A leaf process can exchange applets with the parent only. An applet can in principle be executed on a smart card connected to a smart card reader located at the leaf. This is modelled by sending the identity of the node onto the applet channel thus: `applet!parent`. A leaf makes a non-deterministic choice whether to execute the applet or not.

```
#define run_leaf(parent) \
chan parent = [0] of {chan} ; \
active proctype proc/**/parent() { \
   chan applet ; \
end: \
   do \
   :: parent?applet ; \
      if \
      :: applet!parent \
      :: skip \
      fi ; \
      parent!applet \
   od \
}
```

The world is instantiated to the configuration shown in Figure 1 by the seven macro calls below.

```
run_leaf(vending)              run_leaf(pos)
run_node(shop, vending, pos)
run_leaf(atm)                  run_leaf(counter)
run_node(bank, atm, counter)   run_node(mall, shop, bank)
```

The system described above is general. It allows free travel of applets, and is prepared to interact with any applet at any of the nodes. This is too liberal, and we need a policy to constrain the resulting behaviour to acceptable behaviour.

4.2 The Policy

The init process represents the policy that constrains the behaviour of the system. The init process begins by injecting a loyalty applet, a payment applet and a management applet into the system (via the parent channel of the root node `mall`). Eventually the mall will return the three applets, whence the system terminates. During its life time, the init process is prepared to receive the identity of the host of any of the applets on the corresponding channels `payment`, `loyalty`, and `management`, indicating that the relevant applet is executed while the smart card is connected to the indicated reader.

```
init {
   chan host ;
   mall!loyalty ;
   mall!payment ;
   mall!management ;
end:
   do
   :: loyalty?host               :: payment?host
   :: management?host            :: mall?eval(loyalty)
   :: mall?eval(payment)         :: mall?eval(management)
  od
}
```

4.3 The Principle

The trace specification below represents the principle *Be reluctant to trust* as operationalised by the two customised policies: one for the smart card issuer and one for the smart card holder. The first non-deterministic choice below

represents a payment transaction at the vending machine that must be followed by a loyalty transaction at the vending machine. The second non-deterministic choice represents that the only possibility for a management transaction to take place is at the counter of the bank. The other non-deterministic choices represent the remaining desirable behaviour.

```
trace {
end:
  do
  :: payment?eval(vending) ;loyalty?eval(vending)
  :: management?eval(counter)                    :: payment?eval(pos)
  :: payment?eval(atm)                           :: payment?eval(counter)
  :: loyalty?eval(vending)                       :: loyalty?eval(pos)
  :: loyalty?eval(atm)                           :: loyalty?eval(counter)
  od
}
```

4.4 Analysis

Model checking reveals (again as expected) that the system is ill behaved because the applets roam freely and therefore execute when the principle prohibits this. A concrete counter example shows that after some preliminaries the management applet travels to the point of sale terminal thus: `mall!management`, `shop!management`, `pos!management`. There the management applet executes, which is modelled by sending the identity of the host back to the init process: `management!pos`. The violation of the card holder specific part of the principle can be prevented in the policy by replacing `:: management?host` by `:: management?eval(counter)`. It is not easy to enforce also the card issuer specific part of the principle as this links two events (the payment and the loyalty transaction) that could in principle be separated by an arbitrary number of unrelated events. To introduce such linkage into the policy, a notion of history would have to be included, for example by adding a variable to the model. This shows that the separation of principle and policy brings at least notational convenience that would not be available otherwise.

5 Peer to Peer Music Sharing

The last case study concerns a peer to peer music sharing system [11]. The model of the relevant processes, message flows and computations is given in Figure 2. The boxes denote processes and their internal actions, the arrows denote messages exchanged between the processes. We will discuss each of the processes and messages below. The model is abstract in the sense that:

– We assume that there are two different classes of users: music (1) producers and (2) consumers; there are valid and expired (3) leases; there are valid and invalid (4) payment tokens; there is (5) music in plain text form, (6) encrypted music, and (7) watermarked music; there are (8) keys and there are (9) fingerprints. All of the above 9 categories are assumed to be distinct and incompatible, for example a fingerprint cannot be confused with the identification of music.

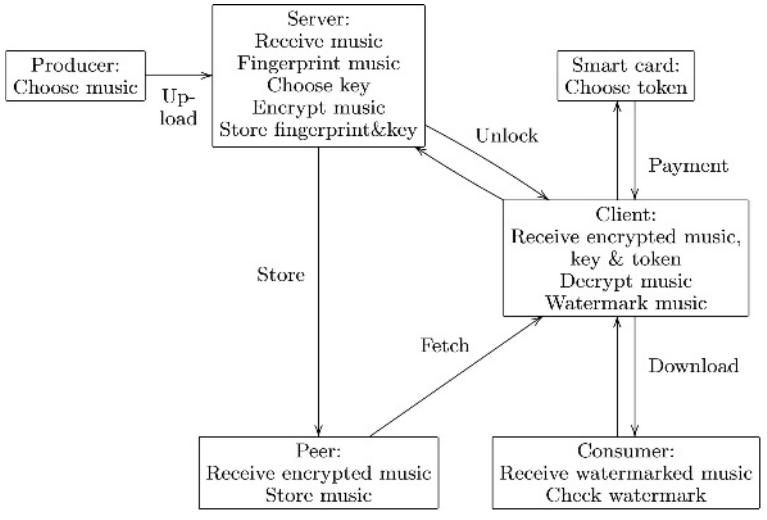


Fig. 2. Abstract Music2Share Protocols.

- We assume: (a) a simplistic peer network where peers do not actively redistribute or copy content; (b) the existence of a secure mechanism for looking up the fingerprint for the desired music; (c) idealised encryption, fingerprinting and watermarking.
- We assume that the producer and the server form a secure domain, that the client and the smartcard form another secure domain, and finally that the communication between the client and the server is secure. We make no security assumptions about the peers.
- Without loss of generality, we distinguish precisely two users, two pieces of music, two lengths of lease, two keys etc. This could be extended but no significant new lessons would be learned from doing so.
- We model small numbers of the different parties of the protocol, except the Server, which is centralised. Distributing the server could be accomplished but this would be a refinement that should remain invisible at the chosen level of abstraction.
- We assume synchronous communication so that the network does not have to store messages. We also assume the network to be reliable.

Under the assumptions above we are now able to present the two main scenarios of use: uploading and downloading music.

Scenario 1: upload. Starting top left in Figure 2, the producer chooses some music and uploads it onto the server (Upload message). The server receives the music, and calculates the fingerprint that will henceforth identify the music. The server then chooses an encryption key, and encrypts the music. The key is stored with the fingerprint for future use. An appropriate peer stores the

encrypted music (Store message) for future reference. At some point in time the server decides that the lease of the music expires and invalidates the key (not shown in the diagram).

Scenario 2a: successful download. Starting bottom right in Figure 2, the consumer chooses some music (identified by its fingerprint) and requests the music from a client (Download request). We assume the client receives a valid token from the smartcard by way of payment (Token message). The client then asks the server for the key (Unlock request). We also assume that the lease has not expired so that the client receives a valid key (Unlock reply). The client also receives the encrypted music from the P2P network (Fetch message). The music can now be decrypted and watermarked with the identity of the consumer. The result is sent back to the consumer (Download reply).

Scenario 2b: failed download. The scenario will change if either payment could not be arranged (because the valid tokens of the smart card ran out), or when the lease has expired. In both cases the consumer receives an appropriate apology, but no music.

5.1 The System

The relevant symbols of the model are:

```
mtype {
   Download, Fetch, Store, Unlock, Payment, Upload,
   Alice, Bob, Alice_Music, Bob_Music, No_Music,
   Long_Lease, Short_Lease, Expired_Lease, Valid_Token, Invalid_Token,
   Bach, Mozart, Bach_Cipher, Mozart_Cipher, Bach_Key, Mozart_Key,
   Bach_Fingerprint, Mozart_Fingerprint
}
```

Here `Download` ... `Upload` represent the different messages that may be transmitted; `Alice`, and `Bob` are the users wishing to download music; `Alice_Music`, and `Bob_Music` represent music watermarked by the identity of the user who downloaded the music; `No_Music` is a place holder for music whose lease has expired; `Long_Lease` ... `Expired_Lease` represents three different lengths of lease for shared music; `Valid_Token`, and `Invalid_Token` represent two possible payment token values; `Bach`, and `Mozart` represent two pieces of music; `Bach_Cipher`, and `Mozart_Cipher` represent the encrypted versions of the two pieces of music; `Bach_Key`, and `Mozart_Key` represent the encryption keys for the two pieces of music; `Bach_Fingerprint`, and `Mozart_Fingerprint` represent the fingerprints of the two pieces of music.

For technical reasons, we need a macro `ord` (below) to map `Mozart_Fingerprint` to 0 and `Bach_Fingerprint` to 1. `NIL` represents an out-of-band (error) symbol.

```
#define ord(m) (m - Mozart_Fingerprint)
#define NIL    0
```

Keys. We assume a unique key for each piece of music. (In addition to C preprocessor macros, SPIN also offers its own variety of macro, the `inline`):

```
inline choose_key(plain, key) {
   if
   :: plain==Bach   -> key=Bach_Key
   :: plain==Mozart -> key=Mozart_Key
   fi
}
```

Given a `plain` text, `choose_key` returns the corresponding `key`.

Fingerprinting and watermarking. Similarly for each piece of music there is a unique fingerprint:

```
inline fingerprint(plain, id) {
   if
   :: plain==Bach   -> id=Bach_Fingerprint
   :: plain==Mozart -> id=Mozart_Fingerprint
   fi
}
```

Once the user has downloaded her Bach or her Mozart, the music is watermarked for her personal use. We are no longer interested in the particular piece of music, only in the user for whom it has been watermarked. An invalid plain text is returned as an invalid marked result `No_Music`.

```
inline watermark(plain, user, marked) {
   if
   :: plain!=NIL && user==Alice -> marked=Alice_Music
   :: plain!=NIL && user==Bob   -> marked=Bob_Music
   :: else                      -> marked=No_Music
   fi
}
```

It should be possible to check whether a piece of content has been watermarked with the correct user identity. `No_Music` does not have a watermark. An incorrect watermark causes a SPIN assertion to fail:

```
inline check_watermark(user, marked) {
   if
   :: user==Alice && marked==Alice_Music -> skip
   :: user==Bob && marked==Bob_Music     -> skip
   :: marked==No_Music                   -> skip
   :: else                               -> assert(false)
   fi
}
```

Encryption and decryption. For each piece of music there is one cipher text:

```
inline encrypt(plain, key, cipher) {
   if
   :: plain==Bach && key==Bach_Key     -> cipher=Bach_Cipher
   :: plain==Mozart && key==Mozart_Key -> cipher=Mozart_Cipher
   fi
}
```

With a valid key, the cipher text decrypts uniquely to the original plain text. With an invalid (expired) key, the result is `NIL`:

```
inline decrypt(cipher, key, plain) {
   if
   :: cipher==Bach_Cipher && key==Bach_Key     -> plain=Bach
   :: cipher==Mozart_Cipher && key==Mozart_Key -> plain=Mozart
   :: else                                     -> plain=NIL
   fi
}
```

Network. The Music2Share network is modelled using two synchronous channels: one for `request` messages and another for `reply` messages. (Channels in SPIN are bi-directional). The channels carry messages with two parameters, where the message header is always one of `Download` ... `Upload`.

```
chan request=[0] of { mtype, mtype, mtype }
chan reply=[0] of { mtype, mtype, mtype }
```

Server. The key server is the only centralised component. The server must create and store keys, it must fingerprint and encrypt music, locate a suitable peer to store encrypted music, and it must serve keys.

Keys are stored together with a lease, which is decremented each time a key is served. This models the process of lease expiry. The data type declaration below defines type `record` holding a key and a lease.

```
typedef record {
   mtype key ;
   mtype lease ;
}
```

The `Server` itself sits in an endless loop waiting for one of two types of messages `Upload` and `Unlock` on the `request` channel.

```
proctype Server() {
   mtype cipher, id, key, plain, user ;
   record store[2] ;
   byte lease ;
   do
   :: request?Upload(plain, lease) ->
      fingerprint(plain, id) ;
      choose_key(plain, key) ;
      store[ord(id)].key = key ;    store[ord(id)].lease = lease ;
      encrypt(plain, key, cipher) ;
      request!Store(id, cipher)
   :: request?Unlock(id, user) ->
      if
      :: store[ord(id)].lease > Expired_Lease ->
         store[ord(id)].lease-- ;
         key = store[ord(id)].key
      :: else ->
         key = NIL
      fi ;
      reply!Unlock(key, user)
   od
}
```

Upon receipt of an `Upload` message with given `plain` text and `lease`, the server calculates the fingerprint `id`, chooses a `key`, stores the key and the lease in at the appropriate entry `ord(id)` in the array `store`, encrypts the plaintext with the key yielding a `cipher`, and finally transmits a `Store` request onto the

network, expecting an appropriate peer to pickup the request and to store the cipher text. This completes the handling of the upload request, no acknowledgement is returned to the requestor of the upload (The network is assumed to be reliable at the chosen level of abstraction).

An `Unlock` request message with a fingerprint `id` and identity `user` causes the server to check the expiry of the lease for the music with the fingerprint `id`. If the lease has expired an invalid key (`NIL`) is created, otherwise the lease is shortened and the correct key retrieved. The key is posted on the `reply` channel, expecting the requestor of the unlock message to pick it up.

Peer. A peer is a simple process that serves only to store and communicate the cipher text corresponding to a particular fingerprint.

```
proctype Peer(mtype id) {
   mtype cipher ;
   request?Store(eval(id), cipher) ;
   do
   :: request?Store(eval(id), cipher)
   :: request!Fetch(id, cipher)
   od
}
```

Before the peer enters its main loop, it expects a `Store` message with the initial cipher text. (The expression `eval(id)` states that the actual parameter of the message must have exactly the same value as the variable `id`; the variable `cipher` on the other hand will be bound to what ever actual value is offered by an incoming message). In the main loop, the peer either offers the cipher text to a `Client` in need of the cipher text, or is ready to receive an updated cipher text. If a second process is waiting for a `request!Store` and a third process is waiting for a `request?Fetch`, both transactions are enabled. In this case a non-deterministic choice is made as to which transaction proceeds first.

Smart card. The Smart card represents a source of pre-paid tokens.

```
proctype Smartcard() {
   do
   :: request?Payment(_, _) ->
      if
      :: reply!Payment(Valid_Token, NIL)     :: reply!Payment(Invalid_Token, NIL)
      fi
   od
}
```

The tokens may run out, which is modelled by the `Invalid_Token`. Subsequent valid tokens are the result of recharging the card (not explicitly modelled).

Client. The Client mediates between the consumer of music and the Music2Share system.

```
proctype Client() {
   mtype cipher, id, key, plain, marked, user ;
   do
   :: request?Download(id, user) ->
      request!Payment(NIL, NIL) ;
```

```
      if
      :: reply?Payment(Valid_Token, _) ;
         request?Fetch(eval(id), cipher) ;
         request!Unlock(id, user) ;
         reply?Unlock(key, eval(user)) ;
         decrypt(cipher, key, plain) ;
         watermark(plain, user, marked) ;
         reply!Download(marked, user)
      :: reply?Payment(Invalid_Token, _) ->
         reply!Download(No_Music, user)
      fi
   od
}
```

The `Client` sits in an endless loop waiting for `Download` messages for a given fingerprint `id` and `user`. The first action is to check payment. If unsuccessful a `NIL` result is returned. Otherwise we `Fetch` the appropriate `cipher` text from a `Peer`. (No request message is necessary here as the peers offer cipher text unsolicited.) Then the client requests the key for the content. The reply message is matched to the identity of the user. After decryption and watermarking the `marked` music is returned to the consumer who posted the download request.

The client receives an invalid key if the lease is expired. In this case the marked result will also be `NIL`.

5.2 The Policy

The producer and the consumer form the endpoints in the value chain, and as such decide the policy for acceptable behaviour. The producer's policy is to upload a choice of music; the consumer downloads a choice of music.

```
proctype Producer() {
   do
   :: request!Upload(Mozart, Long_Lease)   :: request!Upload(Bach, Short_Lease)
   od
}
```

The `Producer` repeatedly tries to upload `Mozart` (on a long lease) and `Bach`, on a short lease. Further combinations could be added freely.

```
proctype Consumer(mtype user) {
   mtype marked ;
   do
   :: request!Download(Bach_Fingerprint, user) ->
      reply?Download(marked, eval(user)) ;
      check_watermark(user, marked)
   :: request!Download(Mozart_Fingerprint, user) ->
      reply?Download(marked, eval(user)) ;
      check_watermark(user, marked)
   od
}
```

The `Consumer` does the opposite of the producer: the consumer tries to `Download` content, checking that the downloaded content is indeed for the intended user. The content is identified by its fingerprint; we assume but do not model here the existence of a secure mechanism for looking up the fingerprint for the desired music.

Initialisation. The initialisation takes care that all processes are started with the appropriate parameters. Here we choose non-deterministically whether to use Alice or Bob as the consumer.

```
init {
   atomic {
      run Server() ;
      run Peer(Bach_Fingerprint)     ; run Peer(Mozart_Fingerprint) ;
      run Smartcard() ; run Client() ; run Producer() ;
      if
      :: run Consumer(Alice)         :: run Consumer(Bob)
      fi
   }
}
```

There is one `Server`, for all other processes we assume that at most two versions exist.

5.3 The Principle

The system policy states that she gets the music she has asked for, unless the lease expires, or she fails to pay. This is captured by the `check_watermark` assertion: a failed assertion implies that the policy is violated. This represents acceptable behaviour (from the point of view of the producer) but not desirable behaviour (because the consumer does not get value for money). The guiding principle is thus *value for money*, translated into a trace declaration requiring on the one hand that each time a consumers pays, he or she is guaranteed to get music, and on the other hand that when the customer cannot pay, she gets no music.

```
trace {
   do
   :: reply?Payment(Valid_Token, _) ;
      reply?Unlock(_, _) ;
      if
      :: reply?Download(Alice_Music, _)      :: reply?Download(Bob_Music, _)
      fi
   :: reply?Payment(Invalid_Token, _) ;
      reply?Download(No_Music, _)
   od
}
```

5.4 Analysis

Model checking the system and the policy reveals that the system does not cause failed assertions, showing that the policy is satisfied. However, the principle may be violated, because the server refuses to deliver an appropriate key once the lease on the music expires. A concrete trace is a little too long to show here; suffice to say that payment takes place, before we request the key. So if the lease expires and no key is forthcoming the user does not get value for money. Swapping the order of payment and key delivery would solve the problem, but at the same time we might introduce a new problem, whereby a key gets delivered for which payment may not be forthcoming. Further study is needed to identify a suitable

business policy which would help to decide which alternative is preferred. The point here is that our methodology causes the right questions to be asked during the design stage.

A further point to note is that the trusted computing base (TCB) of the system is small: the producer, client, server and smart card must be secure, but the peers and the consumers do not have to be secure. The peers and the traffic to and from the peers is encrypted by the protocol, and may thus be transported freely on an open network. The music received by the consumer is watermarked with her identity, so that she can play and copy it for her own use, but if she tries to sell it, the watermark will reveal her identity.

6 Conclusions

We are able to model systems, security policies and security principles formally thus: $(system \parallel policy) \models principle$.

We have applied this idea to four case studies, showing how unexpected security problems arise that violate the principle.

In each of the four case studies the system is abstract, the policy is involved but the principle is short and clear. The systems and policies are more difficult to understand because of the concurrency involved. The principles by contrast are not concurrent.

None of the systems satisfy the relevant principle because of the mobility, showing that model checking leads to insight in the case studies.

SPIN's trace declarations are relatively inflexible. It would be useful to either increase the flexibility by changing the SPIN implementation, by generating trace declarations from higher level policies, or a combination of the two approaches.

We are planning to work on a language for policy patterns based on e.g. Ponder [1], from which SPIN models can be generated automatically. This would make it easier for practitioners to use our method. A particular challenge is to relate counter examples generated by the model checker back to the input language.

Finally we should like to investigate ways in which our models of policies and principles can be incorporated in applications by way of execution monitoring.

Acknowledgements

We thank Peter Ryan and the anonymous referees for their comments on an earlier draft of this paper.

References

1. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In Sloman, M., Lobo, J., Lupu, E., eds.: Int. Workshop on Policies for Distributed Systems and Networks (POLICY). Volume LNCS 1995., Bristol, UK, Springer-Verlag, Berlin (2001) 18–38
`http://springerlink.metapress.com/link.asp?id=1r0vn5hfxk6dxebb`.

2. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000) 30–50 http://doi.acm.org/10.1145/353323.353382.
3. Scott, D., Beresford, A., Mycroft, A.: Spatial security policies for mobile agents in a sentient computing environment. In Pezzè, M., ed.: 6th Fundamental Approaches to Software Engineering (FASE). Volume LNCS 2621., Warsaw, Poland, Springer-Verlag, Berlin (2003) 102–117 http://www.springerlink.com/link.asp?id=nyxyyrlkbe5c5acc.
4. Satoh, I.: Physical mobility and logical mobility in ubiquitous computing environments. In Suri, N., ed.: 6th Int. Conf. on Mobile Agents (MA). Volume LNCS 2535., Barcelona, Spain, Springer-Verlag, Berlin (2002) 186–201 http://springerlink.metapress.com/link.asp?id=yrwh37hleb9rxp81.
5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference manual. Pearson Education Inc, Boston Massachusetts (2004)
6. Madhavapeddy, A., Mycroft, A., Scott, D., Sharp, R.: The case for abstracting security policies. In Arabnia, H.R., Mun, Y., eds.: Int. Conf. on Security and Management (SAM). Volume 1., Las Vegas, Nevada, CSREA Press (2003) 156-160 http://cambridgeweb.cambridge.intel-research.net/people/rsharp/publications/sam03-secpol.pdf.
7. Cheng, B.H.C., Konrad, S., Campbell, L.A., Wassermann, R.: Using security patterns to model and analyze security requirements. In Hietmeyer, C., Mead, N., eds.: Int. Workshop on Requirements for High Assurance Systems (RHAS), Monterey, California, Software Engineering Institute, Carnegi mellon Univ. (2003) 13–22 http://www.sei.cmu.edu/community/rhas-workshop/rhas03-proceedings.pdf.
8. Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S., DuVarney, D.C.: Model carrying code: A practical approach for safe execution of untrusted applications. In: 19th ACM Symp. on Operating Systems Principles (SOSP), Bolton Landing, New York, ACM Press, New York (2003) 15–28 http://doi.acm.org/10.1145/945445.945448.
9. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE **63** (1975) 1278–1308
10. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: 9th ACM Conf. on Computer and communications security (CCS), Washington, DC, USA, ACM Press, New York (2002) 255–264 http://doi.acm.org/10.1145/586110.586145.
11. Kalker, T., Epema, D.H.J., Hartel, P.H., Lagendijk, R.L., van Steen, M.: Music2Share - Copyright-Compliant music sharing in P2P systems (invited paper). Proceedings of the IEEE Special Issue on Digital Rights Management **92** (2004) 961–970 http://ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=1299170.

Smart Devices for Next Generation Mobile Services

Chie Noda and Thomas Walter

DoCoMo Communications Laboratories Europe GmbH, Landsberger Strasse 312,
80687 Munich, Germany
{noda,walter}@docomolab-euro.com
http://www.docomoeurolabs.de/

Abstract. Ubiquitous computing requires new paradigms to assist users organizing and performing daily tasks. Supporting autonomy and providing secured execution environments are two among a lot of challenging issues. Next generation smart cards, so-called smart devices, are regarded as personal devices providing a secured execution and storage environment for application tasks and sensitive privacy information, respectively. Hardware evolution of smart devices enables them to provide capacity to host personal agents, implemented as mobile agents. Mobile agents perform essential tasks such as service negotiation in order to enable automated dynamic service delivery. This paper presents our middleware and a service delivery pattern for user-centric service provisioning by applying mobile agents to smart devices. We further discuss how the middleware platform can be integrated with a security framework for a federation of mobile agents, where a smart device is used as a central trusted entity.

1 Introduction

Increasing hardware capabilities of mobile devices fosters the emergence of a ubiquitous computing paradigm in mobile communication services. In this paradigm, ubiquitous devices surrounding users retrieve context information from the environment and can further actuate to assist users [1]. Most of ubiquitous devices may be invisible to users, for example, embedded or integrated devices to everyday objects or sensors for retrieving user environmental information (e.g., location, weather and temperature) and user status (e.g., at work, on vacation, car driving) [2].

Next generation smart cards, so-called smart devices, are regarded as personal and visible devices (i.e. users are aware of their existence) providing a secured execution and storage environment for application tasks and sensitive information, respectively. Furthermore, a smart device can take a more crucial role when incorporated in a middleware platform. It can also be considered as one of the most applicable places for a user's personal software agent to reside. A user can easily carry own personal agent on a smart device and let it migrate to other places to execute tasks autonomously but securely, such as negotiation with other agents, in both global and local networks. Especially, carrying an agent software component on a smart device fits to the case of local ad hoc network communications, where nodes may become temporarily disconnected and there are no nodes that are always available and reachable from the smart device.

Mobile middleware is regarded as a middleware specialized for a mobile network environment to run on resource-limited devices, i.e., smart devices within this paper,

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 192–209, 2005.

as also discussed in [3] [4]. Specific requirements for mobile middleware, which are usually not considered in traditional middleware running on wired networks, are as follows:

- **Limited resources:** Mobile middleware is to be distributed to resource-limited devices with the characteristics such as low performance, limited amount of memory, requirements of low power consumption, and narrow communication bandwidth.
- **Asynchronous communication:** Mobile devices can be disconnected when they are out of coverage. Asynchronous communication supports disruption of communication links.
- **Context awareness:** User contexts, such as user environmental information and user status, are dynamic. Middleware needs to support context awareness, i.e., it has to be aware of context changes and adapt dynamically.

Many research efforts have been focused on designing middleware systems for mobile environments by applying different approaches. Mobile agents support asynchronous operations by migrating to execution environments, which require network connections only for a migration period and not for an execution period, and dynamic code loading on resource-constrained devices (e.g., [4], [5], [6]). P2P (peer-to-peer) communication enables distributing computational load across multiple peers and enabling context awareness in a P2P manner (e.g., [7], [8]). However, none of them fulfill all the above-mentioned requirements.

Another assumption from our business perspective is that there is no central point or central gateway where services shall be registered or can be found in the heterogeneous network environments. We introduce a dynamic service delivery pattern for automated negotiations in a P2P manner. We build a negotiation framework by utilizing mobile agents integrated with a P2P platform, which represent users and negotiate with other agents autonomously. In particular, we use JXTA [7] as a P2P platform, which enables loosely coupled communications between peers. The fact that mobile agents can create agent communities in an ad-hoc manner, to achieve common goals, by utilizing JXTA peers, pipes and groups concepts, leads us to build an agent platform integrated with JXTA.

Assuming that the next generation smart devices have more powerful physical capabilities in terms of performance, communication speed, and memory, they can host users' mobile agents. As a reference, Moore's Law that observes the computation power available on a chip approximately doubles every eighteen months can be considered [9]. We may foresee that the smart devices support the same physical capabilities with the current mobile phones in a decade. However they will still have fewer resources than other nodes in the network infrastructure.

Another advantage to host a mobile agent on a smart device and migrate to other execution environments on demand is security. By utilizing cryptographic functions on smart devices, a security framework for a federation of agents can be established. We believe that providing a secured environment may become a new business role for mobile network operators in this decentralized service provisioning.

This paper presents our research results related to smart devices: (1) middleware for user-centric service provisioning by applying mobile agents to smart devices, and (2) a security framework for federated agents based on security functionality pro-

vided by smart devises. We discuss how to perform integration of both results into a comprehensive secure service provisioning platform.

The structure of this paper is as follows; in Section 2, starting from a layered middleware architecture, a middleware platform for dynamic service delivery toward user-centric service provisioning is introduced. A dynamic service delivery pattern for supporting flexible service interoperations among autonomous entities across decentralized systems is implemented on a mobile agent platform with integrating a P2P communication paradigm. In Section 3, we discuss how the middleware platform can be integrated with a security framework for a federation of agents, where smart devices provide a central point for trust. Section 4 summarizes the paper and discusses our future research plans.

2 User-Centric Middleware

Here is an example scenario for next generation smart devices. The scenario emphasizes a dynamic construction, negotiation, and delivery of services between users in a P2P manner. Users also take a role of a service provider to offer a specific service.

> *Alice has an overweighed baggage for flight XY1234. To check in her baggage without paying extra, she wants to trade her frequent flyer points for non-used baggage weights with other passengers. She publishes an advertisement to a community that corresponds to her flight XY1234 and shares a common interest for trading baggage weight and frequent flight points. Upon discovery of advertisements, which offer weights, her mobile agent on her smart device as well as other participants' mobile agents migrates to a meeting place. Her and other participants' mobile agents carry users' preferences and strategies. The mobile agents start negotiations and eventually reach an agreement. Once an agreement is achieved among agents, her agent migrates back to her home smart device. The agreed negotiation result is confirmed on her smart device. Her baggage is checked in with transferring her frequent flyer points to the other users' accounts.*

To assist users in a heterogeneous environment, where heterogeneity exists on devices, networks, and services, user-centric service provisioning is one of the key attributes of a middleware platform for next generation mobile communications. It enables secure personalized services according to user profiles and adaptation of services according to users' environmental resources, and to support virtual communities of users and devices in an ad hoc manner. We discuss the design of our mobile middleware and a dynamic service delivery pattern for flexible interactions and automated negotiation, and the implementation based on integration of a mobile agent platform with a P2P communication paradigm.

2.1 Logical Architecture

Most of traditional system architectures are based on a layered approach referring to the OSI Reference Model [10]. To reuse subsystems in existing middleware architectures, we follow a layered approach and add a sub-layer for user support, as also seen

in [11]. We specify a layered architecture for user-centric service provisioning from a logical point of view as shown in Figure 1, driven by focusing on the demands and the behaviors of users.

The network support layer provides functionality for network communications control in heterogeneous networks. The service support layer contains traditional middleware functionality, such as service discovery and advertisement, profiling, and security. The user support layer supports autonomous and proactive agent aspects that traditional service middleware lacks. It enables simplifying user interactions with a system and providing user-centric services, by personalization based on user preferences and autonomous coordination. More details of the logical architecture can be found in [12].

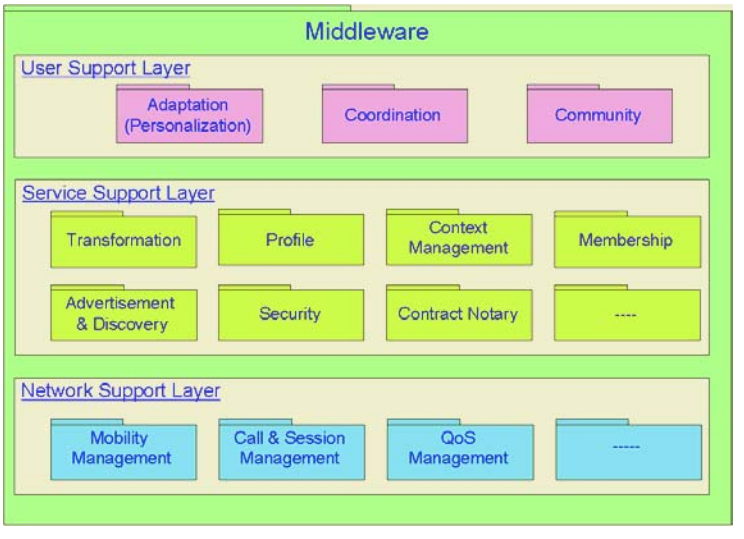


Fig. 1. Logical middleware architecture

2.2 Dynamic Service Delivery

A dynamic service delivery pattern is designed to enable flexible interactions and automated negotiations among autonomous entities in a P2P manner in a decentralized system. It allows service advertisement and discovery, negotiation, and aggregation, driven by negotiation strategies of participants and governed by a contract notary. *Advertisement & discovery*, *security*, and *contract notary* are subsystems in the service support layer, associated with the pattern. The pattern is used whenever a subsystem or a software object needs to interact with other objects. It enables a system to support adaptability and re-configurability. It differs from other general negotiation frameworks, such as [13] that also specifies an application independent negotiation framework but does not aim to support internal interactions within a system.

The dynamic service delivery pattern consists of three phases that require participants exchanging messages with a coordinator to negotiate construction and delivery of a service:

- **Introduction phase:** Service providers and users introduce to each other with respect to a service offering by using publish/subscribe interfaces for service discovery and advertisement (i.e., association with the *advertisement & discovery* subsystem).
- **Negotiation phase:** Service providers and users interact in a negotiation dialog, with defining negotiation template schemes, exchanging credentials (i.e., association with the *security* subsystem) and proposals, and validate agreements, in order to reach mutually acceptable agreements. They are stored as contracts in a persistent repository (i.e. association with the *contract notary* subsystem).
- **Service delivery phase:** Service providers and users interact to fulfill the terms of the negotiated contract. Upon completion of the service delivery phase, the dynamic service delivery lifecycle is terminated.

Any entity can be a *service provider* or a *user* for offering or being offered services. At the same time, it also takes a role of a *participant* or/and a *coordinator* for the negotiation, where a participant makes proposals and a coordinator governs participation, execution, and validation of proposals and agreements according to the *negotiation template scheme*. As an example, entities, their roles as well as relationships are shown in Figure 2. Entity A and B who share a common interest introduce each other as candidates for negotiation. Entity B further takes a coordinator role as a trusted entity during negotiation. Or an independent trusted entity can take a role of a coordinator. Negotiation interactions between participants are processed through the coordinator. Eventually the service is delivered between Entity A and B.

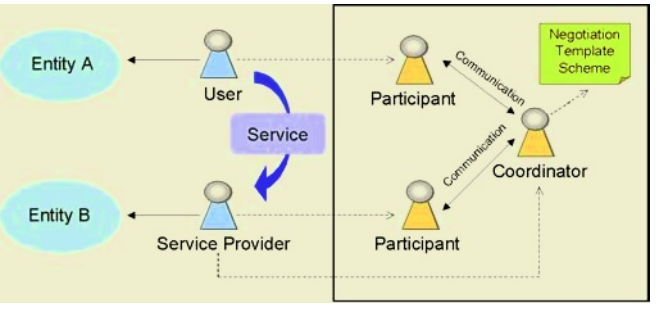


Fig. 2. Example of entities' roles for the dynamic service delivery pattern

Figure 3 shows the sequence of the dynamic service delivery pattern according to the roles of the entities in Figure 2. The negotiation is started with creating a coordinator. Agreeing a *negotiation template schema*, which defines rules and protocols on how the *coordinator* should carry out negotiations, follows as option. The *negotiation template schema* also can be pre-defined by the host of the *coordinator*. Exchanging credentials is supported optionally, which we discuss in details in Section 3. The *coordinator* asks the *participants* to submit initial proposals based on a participant's *negotiation strategy* such as an offer and a demand. It further creates modified proposals if demands are not matched. The *participants* check whether the modifications proposed by the *coordinator* are acceptable according to each *negotiation strategy*, and send back a revised proposal. Once a *service agreement* is achieved, i.e., a pro-

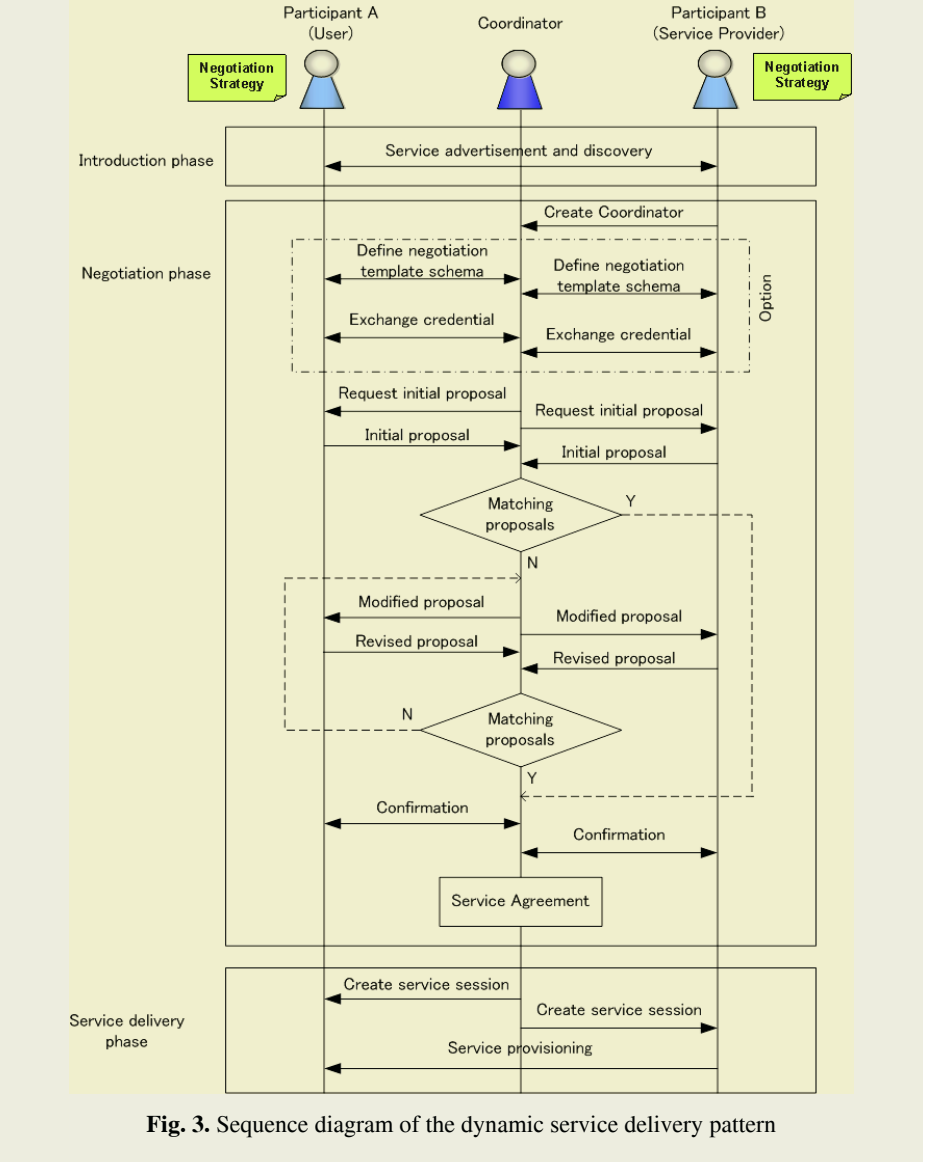


Fig. 3. Sequence diagram of the dynamic service delivery pattern

proposal and a demand match, the coordinator files it as a contract, creates a *service session manager* to control service sessions, and monitors transactions between participants' entities for service delivery enforcement. Figure 4 depicts the software design for dynamic service delivery.

2.3 Platform Implementation

We have assumed that users always carry own software agents on smart devices and agents act proactively for negotiation and personalized service provisioning. An implementation can be done taking the following advantage of mobile agents:

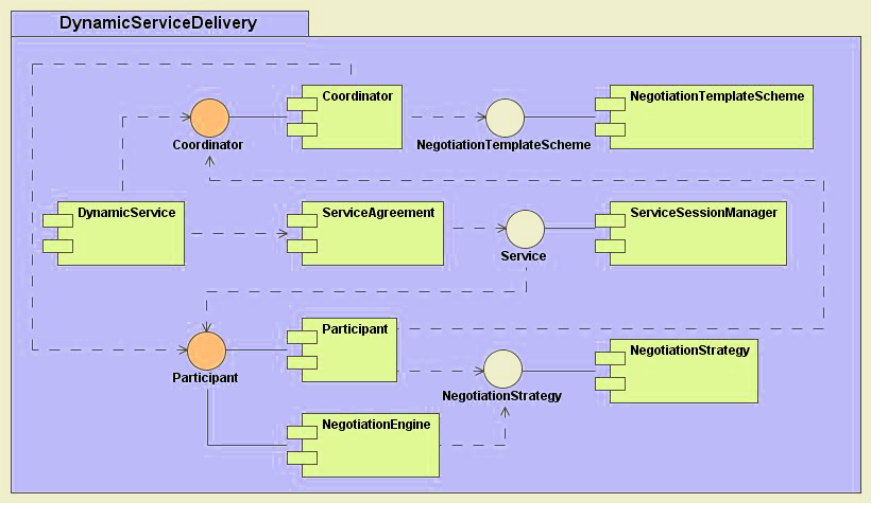


Fig. 4. Software design of the dynamic service delivery pattern

- **Dynamic code loading:** Mobile agents enable transferring code and state information from one host to another. They support dynamic code loading to resource-limited smart devices, to distribute subsystems or software components in a logical middleware architecture.
- **Asynchronous communication:** Communication is only required when mobile agents migrate to other execution environments. An agent moves to another place, executes processes there "off-line" from its home base, and after having finished negotiation comes back with the result. All is done basically without any interaction with home base.
- **Performance:** In case of a negotiation among multiple agents, performance may be improved by allowing mobile agents to negotiate locally.
- **Autonomy:** A software agent may act proactively, which allows delegating negotiation tasks to an agent that negotiates on behalf of an entity. This is especially beneficial in case of resource-limited devices, where the actual negotiation can then be performed by a delegate agent running on a more capable host, for example on a mobile network infrastructure.

In addition to the above-mentioned advantages, it is natural to build a dynamic negotiation framework on mobile agents. They represent users and service providers, and negotiate with other agents autonomously, as also discussed in [13]. For implementation, we have chosen the enago Mobile platform [14] that is the successor of the Grasshopper agent platform [15], because it supports in particular agent mobility.

To enable negotiation interactions among agents in a P2P manner, we propose to use JXTA as an underlying communication for mobile agents. JXTA enables loosely coupled communications between peers, participation of community-based activities and offering services across different systems. Our mobile middleware platform has been implemented completely in Java, using the JXTA reference implementation as the foundation. Figure 5 provides an outline of the implementation architecture. Implementation of *advertisement & discovery*, *community*, and *membership* subsystems

are mostly mapped to JXTA functions on the middleware, e.g., JXTA discovery service, and JXTA peer group service. Most of implementation of *security* subsystem is done on Java cryptographic functionality.

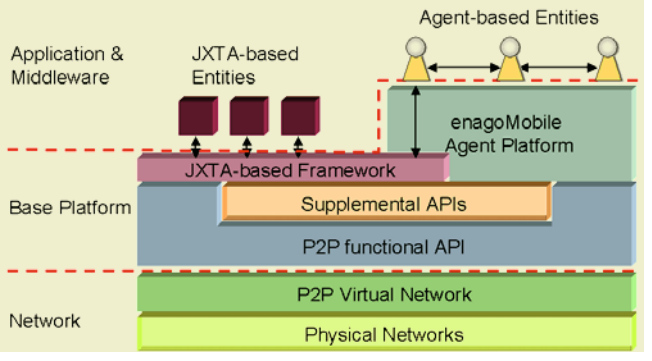


Fig. 5. Platform implementation architecture

In the current implementation, smart devices are implemented as Java Card plus standard PC or Java Card plus PDA/iPAQ, as shown in Figure 6. enago Mobile was integrated with JXTA as an alternative communication protocol on standard PCs. The Java Card provides two functionalities basically: (1) store and update user profiles and (2) executes security-related functions. The profiles represented in XML are compressed and stored as binary representation on the Java Card to save memory and get more performance. An RSA key pair and a public key certificate are stored on the Java Card. The Java Card applet provides the capability to authenticate an external entity, generate and verify the hashed signature, and encrypt and decrypt messages.

We implemented JXTA on IBM's J9 Java VM [16] on iPAQ. The current prototype on iPAQ only supports enago Mobile with proprietary P2P extensions, which are equivalent with JXTA functionalities. 2 Mbytes are required to support enago Mobile with P2P extensions and 13 Mbytes for the middleware architecture including dynamic service delivery pattern on iPAQ.

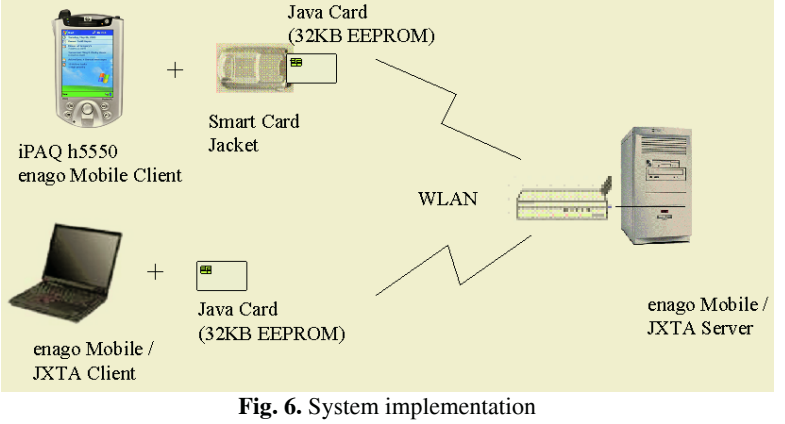


Fig. 6. System implementation

2.4 Negotiation by Mobile Agents

Figure 7 shows a typical negotiation scenario using mobile agents. There are two types of agents:

- **Negotiation agent:** This agent implements the *participant* role of the dynamic service delivery pattern, and negotiates in the negotiation phase on behalf of a platform entity – either user or service provider. The negotiation behavior of the agent is controlled by one or more *negotiation strategies*, which must be provided by the initiating platform entity. Negotiation agents are created on demand when entering the negotiation phase.
- **Coordinator agent:** This agent implements the *coordinator* role of the dynamic service delivery pattern. It allows coordinating the whole negotiation process according to one or more *negotiation template schemes*, which must be provided by the creator of the agent. Coordinator agents might be pre-installed and offered by service providers in the infrastructure, but also created on demand by a negotiation initiator in case of P2P negotiation.

We implemented the introductory scenario. Negotiation agents are implemented on users' smart devices, i.e., Java Card plus standard PC or Java Card plus iPAQ. The airline company provides a coordinator agent for negotiations, implemented in the network infrastructure. In our implementation, Alice broadcasts an advertisement to the community group of XY1234, "I need 10 kg weights and offer 100 points per kg". Upon discovery of advertisements for offering weights, the coordinator agent provides a meeting place on its network infrastructure, and calls the negotiation agents to migrate there. Users' negotiation agents generated on smart devices on demand migrate there carrying negotiation strategies, to meet and negotiate co-locally with other mobile agents. In the sense that the network infrastructure can provide more physical capabilities, migration of mobile agents to the network side allows us to get better performance. Another advantage is that the airline company can be seen as a trusted entity and enforce service provisioning to fulfill a service agreement, e.g., exchanging frequent flight points and baggage weights between users in our scenario. However in other scenarios, which allow trading personal assets between users for example, we may assume that one of them takes the role of a coordinator. In this case, one smart device, which hosts a coordinator agent, becomes a meeting place for negotiation agents.

Let us assume the negotiation agent of Alice has a negotiation strategy, "I need 10 kg weights. I offer 100 points to 140 points per kg as maximum for by half an hour before the departure time. I can offer 150 points to 200 points as maximum after that". Here are two advertisements of offering weights from Bob and John, which partly matches Alice's demand: Bob offers 12 kg weights with 200 points per kg, and John offers 14 kg weights with 160 points per kg. A negotiation schema can be to take the average between the offered and the demanded amount of frequent flyer points. The coordinator agent calculates the average of points, and makes a revised proposal for trading 10 kg and 150 points per kg between Alice and Bob, and 10 kg and 130 points per kg between Alice and John. The latter one is acceptable for Alice's negotiation agent. If it is also in the range of John's negotiation strategy, a service agreement is achieved. When only John's advertisement is available and it is already 20 minutes before the departure time, Alice's agent accepts the offer from John.

In our demonstration, negotiation strategies and schemas are implemented directly in Java. Formalization of a negotiation language based on descriptions such as RDF and OWL [17] are for further investigation.

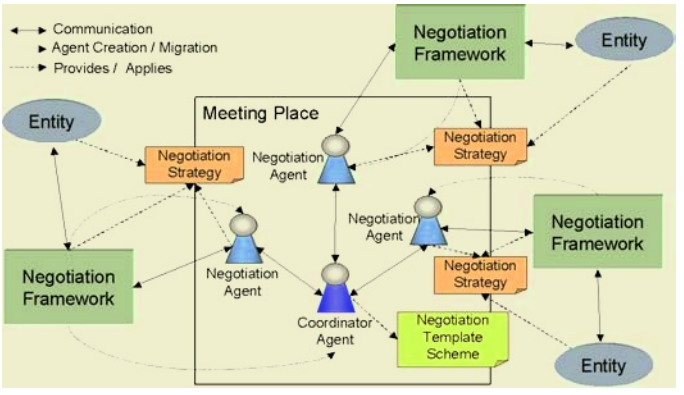


Fig. 7. Example of dynamic negotiation using mobile agents

As one can see here, the advantages of dynamic negotiation built on integration of mobile agents with underlying P2P communications are: (1) to support autonomic negotiation among mobile agents driven by negotiation strategies, (2) to improve performance through co-local negotiation because of mobility support by mobile agents, (3) to enhance privacy, which is a side effect of co-local negotiation, as negotiation information may not be intercepted by network transmissions, and (4) to support both centralized and decentralized business scenarios (i.e., a coordinator agent hosted by a sort of a centralized server or a network infrastructure as presented in the scenario, or a coordinator agent hosted by one of negotiation participant's smart device) through P2P extensions.

The identified fundamental requirements for smart devices are: (1) to support mobile agents based on P2P communication, (2) to support dynamic negotiation frameworks by utilizing mobile agents, and (3) to provide the above mentioned security functions, which we discuss in details in the next section.

3 Security Framework

In the following, we define a security framework for the mobile agent platform. Some of the important security functions to be supported have already been mentioned above: authentication, data confidentiality as well as digital signatures. Besides said functions, other important security topics can be identified such as security of execution platform for mobile agents. While we assume that it is the responsibility of every smart device to protect the execution platform, e.g., against malicious code, we discuss necessary mechanisms how service negotiation can be protected and thus the level of trust in the correct execution of negotiation process and service agreement can be assured. The proposed security framework is based on well-known concepts, basically *public-key cryptography* [18] but as well based on only recently published ones such as *federations of mobile devices* [19].

Our security framework provides two levels of security. First, where mobile agents perform authentication, protection of data for confidentiality and non-repudiation of negotiation results by themselves. The second higher level foresees that all critical security operations are only performed on smart devices (and thus in a secured execution environment); thus mobile agents have to move back to their home base for such operations to be performed.

Our work concentrates on security of the negotiation process. Any privacy issues, e.g., handling of customer data made available during service negotiation, is beyond our current scope of study.

3.1 Federation of Agents

As shown in Figure 7, mobile agents – either negotiation or coordination agents – meet in a meeting place. In the described setting, the meeting place runs on a single device. The coordinator agent performs the role of a mediator between negotiation agents and several negotiation agents may participate. After successful conclusion of service negotiation, a so-called service agreement is established.
From above identified negotiation process the following security requirements are derived:

- Authentication of mobile agents – negotiation as well as coordination agents - so that every agent knows the identity of the respective communicating peer. Alice and John need to know each other so that weights and points can be exchanged. In the example, it is the airline, which requires identity information to enable the respective exchange of incentives: weight vs. point. If required the authentication phase of a protocol run may, optionally, establish a common secret for protecting data exchanged during negotiation.
- Data confidentiality and data integrity so that every agent knows who can read data sent (i.e. data confidentiality) and whether received data is equivalent to sent data (i.e. data integrity). Since behaviors of negotiation agents depend on exchanged data, e.g., proposals, integrity of such data is required in order to avoid that third parties (i.e. other negotiation agents) can change proposals of others and thereby gain an advantage.
- Digital signing of negotiation results in order to facilitate non-repudiation so that any agent is protected against false denial of agreed results for the service delivery phase. Alice and John require that the respective negotiation result is signed so that they can claim it against each other and the airline if required.

In order to set up a communication environment where all above mentioned requirements are met, we introduce a *federation of agents* (see [19] for a motivation of the federation concept). The technical means to support authentication, confidentiality, integrity and non-repudiation are based on cryptographic techniques such as certificates, private and public key pairs (public key cryptography) and session keys (symmetric cryptography); refer to [18] for an in-depth discussion of cryptography and network security.

3.1.1 Certificates

Certificates are protected data. Protection is done by (1) cryptographic procedure based on public key cryptography and (2) a limited lifetime after which a certificate

becomes invalid (or should be regarded as invalid). In our application, certificates bind user data to public keys. To be more specific, mobile agents are those entities that hold certificates so that they can prove their identities. Certificates are signed by *trusted third parties*, thus enabling any entity to verify a received certificate. The latter is done by verifying the signature of the public key certificate. Verification here means to get evidence that the certificate has not been tampered or modified. It is another issue to verify that the sender of the certificate holds indeed the secret, i.e. the private key, that goes with the certificate (see Section 3.2.1 below how this can be achieved).

3.1.2 Public Key Cryptography

Private and public key pairs are used for en- and decryption of data and for digital signature. Private as well as public key can be used to perform en- and decryption, respectively. We use this unique property for session key exchange and digital signatures.

- For the exchange of session keys, we use the public key of the recipient to encrypt the secret session key. Only the recipient as owner of the corresponding private key is able to decrypt the session key; thus confidentiality is guaranteed.
- Since a key pair is uniquely bound to a single entity, applying the entity's private key to data enables recipients to verify the origin of said data. This feature is based on the assumption that an entity keeps its private key protected and that only this entity has access to its private key. A recipient uses the corresponding public key for validation of received data. If decryption is successful this provides evidence of the sending entity.

3.1.3 Symmetric Cryptography

Session keys have the same functionality as private and public key pairs; they can protect data for confidentiality. However, the computational effort to encrypt and decrypt data with a session key is in general much lower than using private and public key pairs. Therefore, they can be used with advantage for en- and decryption of bulk data. Bulk data such as the data generated during the negotiation phase.

3.1.4 Secret Data

In order to set up a federation it is required that every participating agent is carrying its public key certificate for authentication and that every agent being equipped with the certificate of the trusted third party for verification of other entities' certificates (Figure 8). The agent's private key must also be available. Additionally, a number of cryptographic functions must be supported such as en- and decryption with private and public keys as well as symmetric keys, hash functions and digital signatures.

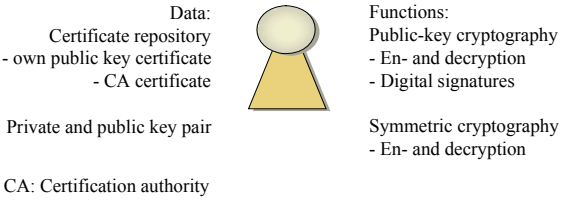


Fig. 8. Mobile agent, key pairs and certificates

3.2 Phase Transitions

The discussed federation of agents can be used to protect the different phases of the dynamic service delivery pattern discussed in Section 2.2. The described procedures are independent of any physical distribution of mobile agent across smart devices and meeting places. The details of each phase and flow of data between agents and entities is depicted in Figure 3 above and refined in Figure 9.

For the introduction phase – service advertisement and discovery process – no data is exchanged that requires any protection; thus, no specific security features are needed. The same holds for the delivery phase, although the basic mechanisms explained in Section 3.1 may optionally be applied in order to achieve security as well as privacy.

3.2.1 Negotiation Phase – Basic Security Level

Every negotiation agent provides its public key certificate to the coordination agent and vice versa. The agents perform a certificate validation. Authentication of agents is additionally supported by using a challenge-response protocol (see [20] for an in-depth discussion). The challenge is generated and encrypted to the private key of *requestor* agent and sent to the *receiver* agent; along with the requestor's certificate. After verification of certificate, the challenge is decrypted with the public key of the requestor. A response is computed (i.e. encrypted to the private key of the receiver) and together with the receiver's certificate sent to the requestor. Verification of certificate and response terminates authentication procedure. Because each agent uses its private key for encryption it thereby proofs that it indeed holds the secret that goes with the public key certificate.

Session keys are exchanged between coordination agent and each negotiation agent, respectively. Session keys are encrypted to the public key of every recipient agent. Subsequently, coordinator agent and negotiation agents perform the negotiation dialog. Session keys are used for confidentiality of data.

Upon successful termination of negotiation phase, the coordinator digitally signs the service agreement and sends it for signature to the respective participants. Participants sign the service agreement and return back the digital signature. Service agreement as well as all signatures are bound together and are sent to the participants. Performing this procedure provides a level of protection to achieve non-repudiation; neither negotiation agent can falsely deny having not agreed on a specific negotiation result. This becomes important as soon as service delivery phase is started.

3.2.2 Negotiation Phase – Enhanced Security Level

For the authentication procedure as discussed in the previous section, it is required that each agent has its private key available. The overall security and trustworthiness of authentication, negotiation process and service agreement depends on the execution platform as well as communication links between entities. For both we should provide a level of security so that no secret data is disclosed. This is particularly true for private keys of agents which should be stored in a protected environment.

If, for any reason, one does not trust the execution platform or the communication link between home base and execution platform, a mobile agent hopping to another platform should neither carry its private key nor should the mobile agent move across

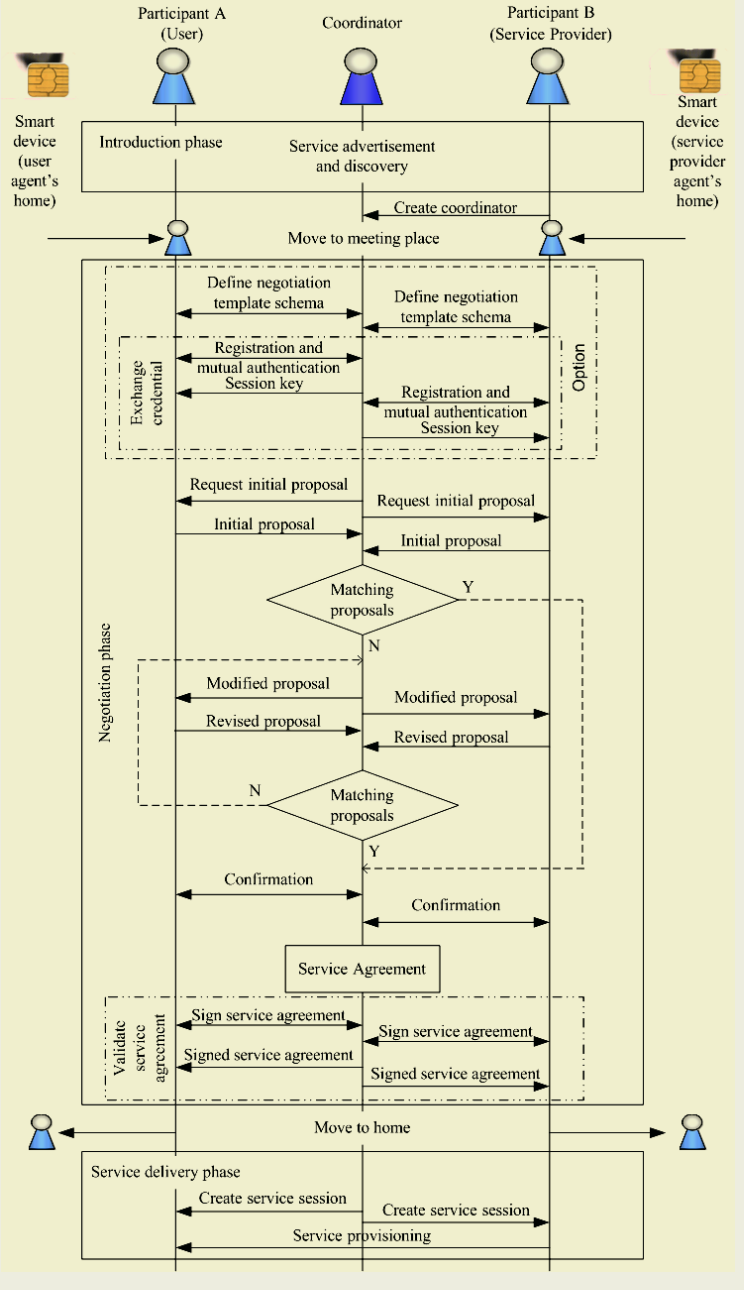


Fig. 9. Sequence diagram of secured service delivery pattern

an unprotected link. Protection of communication link between home and a meeting place can be established using standard procedures; including the one for authentication and key exchange described in Section 3.2.1. Additionally, if an operation is to

be performed requiring access to a private key, the mobile agent should move back and performs the private key operation in the trusted environment of its home. Moving forth and back for private key operations have to be done for decrypting challenge/response, decrypting session keys and for digitally signing the service agreement. The performance loss is outweighed by the additional level of security when all private key operations are performed on a smart device in tamper-resistant hardware (refer to discussion following section as well).

3.3 Design Decisions

In this sub-section we discuss implementation choices and motivate our approach.

3.3.1 Implementation Base

As written in the introduction to this section, the implementation of security functions is done using well-known mechanisms. The only non-standard elements in our setting are the federation of mobile agents' concept as well as its implementation support, i.e. smart devices. For the latter and as a proof of its feasibility, a smart card providing mentioned data storage and functions (Figure 8) have been implemented in the WiTness project [21].

3.3.2 Public Key Infrastructure

In order to support digital signatures, public key cryptography is an obvious (and only) choice. However, with public keys the problem of distributing and verifying said keys becomes evident. Certificates are a means to support both. The owner of a public key holds a certificate and when asked for his or her public key, the certificate is sent.

Verification of certificate is done as follows: Certificates are signed by a trusted third party and the digital signature is bound to the certificate. The receiver of a certificate simply has to check the digital signature and thus can determine the consistency of the certificate.

For issuing certificates, two alternatives exist. One alternative employs so-called *certification authorities* (CA). A CA generates a certificate for a user. The CA checks data provided by the user, CA fills in the certificate and digitally signs it using its private key. If another user receives a certificate, it uses the CA's own certificate to get the CA's public key which in turn is being used for the verification of the received certificate. Quite obviously, the CA has to apply for a public key certificate with another CA as well; and verification of the CA's certificate is done following the same procedure as outlined previously. Specific CAs may generate self-signed certificates which are regarded as per se trusted certificates.

The second alternative is used in PGP (Pretty Good Privacy) [22]. Every user maintains a library of validated public keys which is called *web of trust*. Verification of keys is done as follows: first, everyone may sign and thus certify public keys. Second, if one gets a signed public key, the receiver checks all signature of said key and if among all signature one is found which is known and trusted by the receiver, then the public key is regarded as validated. Otherwise, the receiver has to establish correctness of the public key by other means, e.g. manually checking the keys fingerprint with the originator of the public key.

The problem of maintaining public key certificates is more complex with the second approach; i.e. by chance a trustworthy certificate is among the certificates of a public key while in the first approach one trustworthy entity certifies all public keys. However, the advantage of the second scheme is that certification of public keys does not require a central infrastructure. The web of trust works fine with small and static communities but is impractical in an environment that is highly dynamic with respect to number of participating parties. Another point is that for a specific application there is a natural choice which administration or organization issues certificates; i.e., for the introductory example the airline may issue required certificates.

3.3.3 Performance Issues

As a matter of fact, any security mechanism used results in performance losses. This holds as well as for the described schemata. And, of course, there is even an additional effort inherent to the enhanced security level schema: mobile agents have to move back and forth whenever a security function is being called that requires access to the respective private key which is assumed to be on the mobile agents home base, i.e., smart device.

Compared to the basic security level, this additional effort is serious if the actual negotiation phase is short, i.e., only a few iterations. On the other hand, the longer the negotiation phase the less overhead is implied by the enhanced security level.

To run a negotiation under basic or enhanced schema depends on a subjective assessment of security requirements. For the overweight baggage example, one may go for the basic schema because the negotiation meeting place is run under control of the airline which is regarded as trusted because of an already established trust relationship (i.e., enrollment in the frequent flyer program).

3.4 Open Issues

Here we discuss two open issues that relate to the use of certificates and trust.

3.4.1 Certificates, Chains and Revocation Lists

Verification of certificates is done to the extent that the certificate's integrity is verified; using the public key of the certification authority to validate signatures attached to the certificate. In addition, it is necessary to assess the validity of the certificate. Two elements are used for this. First, every certificate has associated a lifetime. If lifetime has exceeded the certificate automatically renders invalid. Second, it should be verified whether the certificate has been withdrawn, e.g., in case the corresponding key pair has been compromised. To support withdrawal of certificates, certificate revocation lists are maintained by certification authorities. These lists can be accessed on-line and thus validity of certificates can be established.

Certificate revocation lists are fine in on-line scenarios where all entities always have network connectivity and have access to such lists. In off-line scenarios, i.e., smart devices or federations of smart devices become disconnected temporarily, the instantaneous validation of certificates by checking revocation lists is not possible. A solution applicable is to issue certificates which have a rather short lifetime, e.g., as long as it take to perform one iteration of the service delivery pattern. Before a mo-

bile agent leaves its host device, the device generates a short-lived temporary certificate and signs it with its private key. A mobile agent hopping to another smart device thus carries two certificates: the one of the hosting smart device as well as the temporary certificate generated for the service delivery pattern. The above description on agent authentication (Section 3.2.2) is extended using a chain of certificates (temporary certificate - mobile agent's host certificate – root certificate). Note that this approach does not solve the general problem of certificate revocations. On the other hand, certificate revocations can be better handled on the hosting platform than by the agent itself.

3.4.2 Trust Modeling

Next generation mobile networks are comprised of huge numbers of computation nodes that potentially may perform as service providers and users. Although authentication of entities is performed and non-repudiation services are applicable, an uncertainty remains that a negotiated agreement is not fulfilled by either entity. Trust becomes an issue. Trust is to be understood as the level of confidence that one entity has into another entity and that said other entity performs correctly a service request.

Trust is based on observations. Observation gained over time and aggregated into a trust value. In addition, aggregation may take observation of trustworthy third parties into account as well. Or, in case a peer entity is unknown, trust can be derived from recommendation of other entities. This approach to build trust is put forward in [23].

In the context of our current research, use of trust values in negotiation as well as service delivery phase is for further study.

4 Conclusion

In this paper, we have presented a middleware architecture for decentralized systems and a negotiation pattern for dynamic service delivery as an underlying flexible P2P-based interaction mechanism. In this paradigm, all entities can take roles for both a user and a service provider. For next generation smart devices, it is a challenge to support mobile agents integrated with P2P communications, so that mobile agents can migrate to other places to meet with other mobile agents on demand and negotiate for service delivery. It enables autonomic negotiation derived by negotiation strategies, enhances performance and privacy enhancement through co-local negotiation, and support both centralized and decentralized scenarios.

We further discussed to apply a security framework for mobile agents negotiation, to take advantage of smart devices, i.e., security functionality support and tamper-proof hardware. It enables a federation of software agents, with supporting mutual authentication, confidentiality, integrity and non-repudiation. Mobile agents are particularly useful for asynchronous communication where devices can be temporarily disconnected or being off-line. On the other hand however, such situation cannot be neglected for provisioning of certain security functions, i.e. access to revocation lists. Temporary certificates are a pragmatic solution only and further studies are required.

References

1. M. Weiser: The Computer for the 21st Century. Scientific American, September 1991.
2. F. Mattern, P. Sturm: From Distributed Systems to Ubiquitous Computing. Fachtagung Kommunikation in verteilten Systemen (KIVS), Leipzig, Springer-Verlag, February 2003.
3. L. Capra, W. Emmerich, C. Mascolo: Middleware for Mobile Computing. Wireless Personal Communications, Kluwer, 2002.
4. EU IST Project: LEAP. http://leap.crm-paris.com.
5. emporphia Ltd.: FIPA-OS. http://www.emorphia.com/research/about.htm
6. Telecom Italia Lab: JADE. http://jade.cselt.it
7. Project JXTA: JXTA. http://www.jxta.org
8. Steglich, S. Sameshima and all: I-Centric Services Based on Super Distributed Objects. ISADS 2003 The Sixth International Symposium (April 2003)
9. G. Moore: Cramming more components onto integrated circuits. IEEE Electronics, Vol 38, April 1965
10. International Standards Organization: Information technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model. ISO/IEC 7498-1, 1994.
11. WWRF – Wireless World Research Forum WG2: Service Infrastructure of the Wireless World. http://www.wireless-world-research.org/.
12. C. Noda, L. Strick, H. Honjo et al: Distributed Middleware for User Centric System. WWRF#9 conference, Zurich, July 2003.
13. C. Bartolini, C. Preist, N. Jennings: A Generic Software Framework for Automated Negotiation. AAMAS'02, Bologna, Italy, July 2002.
14. IKV++ Technologies AG: enago Mobile Agent Platform. http://www.ikv.de/content/Produkte/enago_mobile.htm.
15. IKV++ Technologies AG: Grasshopper Agent Platform. http://www.grasshopper.de.
16. IBM: WebSphere Studio Device Developer. http://www-306.ibm.com/software/wireless/wsdd/.
17. M. Smith, C. Welty, D. McGuinness: OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/.
18. W. Stallings: Cryptography and Network Security – Principles and Practice. Prentice-Hall, 1999.
19. T. Walter, L. Bussard, P. Robinson, Y. Roudier: Security and trust issues in ubiquitous environments - The business-to-employee dimension. SAINT 2004 Symposium on Application and the Internet, Tokyo, January 2004.
20. R. E. Smith: Authentication from passwords to public keys. Addison-Wesley, 2002.
21. WiTness consortium: WiTness – Wireless Trust for Mobile Business. www.wireless-trust.org, 2004.
22. Simson Garfinkel: PGP Pretty Good Privacy. O'Reilly, 1995.
23. Vinny Cahill, Elizabeth Gray et al: Using Trust for Secure Collaboration in Uncertain Environments. Pervasive Computing, July-September 2003.

A Flexible Framework for the Estimation of Coverage Metrics in Explicit State Software Model Checking*

Edwin Rodríguez, Matthew B. Dwyer, John Hatcliff, and Robby

Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan, KS 66506, USA
{edwin,dwyer,hatcliff,robby}@cis.ksu.edu

Abstract. Explicit-State Model Checking is a well-studied technique for the verification of concurrent programs. Due to exponential costs associated with model checking, researchers often focus on applying model checking to software units rather than whole programs. Recently, we have introduced a framework that allows developers to specify and model check rich properties of Java software units using the Java Modeling Language (JML). An often overlooked problem in research on model checking software units is the problem of *environment generation:* how does one develop code for a test harness (representing the behaviors of contexts in which a unit may eventually be deployed) for the purpose of driving the unit being checked along relevant execution paths?
In this paper, we build on previous work in the testing community and we focus on the use of coverage information to assess the appropriateness of environments and to guide the design/modification of environments for model checking software units. A novel aspect of our work is the inclusion of *specification coverage* of JML specifications in addition to code coverage in an approach for assessing the quality of both environments and specifications. To study these ideas, we have built a framework called **MAnTA** on top of the Bogor Software Model Checking Framework that allows the integration of a variety of coverage analyses with the model checking process. We show how we have used this framework to add two different types of coverage analysis to our model checker (Bogor) and how it helped us find coverage holes in several examples. We make an initial effort to describe a methodology for using code and specification coverage to aid in the development of appropriate environments and JML specifications for model checking Java units.

1 Introduction

Building concurrent object-oriented software to high levels of assurance is often very challenging due to the difficulty of reasoning about possible concurrent task interleavings and interference of interactions with shared data structures. Over the past years, several projects have demonstrated how model checking technology can be applied to

* This work was supported in part by the U.S. Army Research Office (DAAD190110564), by DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), by NSF (CCR-0306607), by Lockheed Martin, and by Rockwell-Collins.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 210–228, 2005.

detect flaws in several types of software systems [1, 10, 12] including concurrent object-oriented systems [6, 3, 19]. Model checking is very useful for the analysis of concurrent systems because it can explore all the program states represented by possible execution interleavings of the threads in a system, spotting very intricate errors. However, the exhaustive nature of model checking makes it difficult to scale to large systems. This lack of scalability is due mainly to the state-space explosion problem: as systems become more complex, the state space grows exponentially, making it very difficult to analyze large systems without having the analysis exhaust available memory.

For these reasons, many reseachers believe that it is most natural to apply model checking to software units, or modules, instead of whole systems. Specifically, software model checking is often envisioned as part of a development and quality assurance methodology in which it is incorporated with *unit testing*. When model checking a software unit, one typically desires to specify/check as much of the unit's behavior as possible in the hope of detecting as many bugs as possible. In the past, model checkers have only supported checking of temporal property specifications with simple state predicates and assertions. To enhance the ability of developers to specify meaningful properties of software units, we have recently extended Bogor[20], our software model checking framework, to support checking of rich specifications [19], written in JML (Java Modeling Language, [16]). Using JML, developers can specify class invariants and method pre/post-conditions that contain detailed constraints on variables and data structures using various forms of universal and existential quantification over heap-allocated data.

To apply model checking to software units (with or without JML specifications), a developer needs to follow an approach that is similar in many respects to the steps involved in traditional unit testing. In unit testing, one develops a *test harness* that makes method calls into the unit for specific sets of parameter values and examines the results of the method calls for invalid results (indicating failed test). When applying model checking to software modules, one must similarly use a test harness (also termed a *closing environment* or an *environment*) to drive the unit through particular execution paths. The scale and complexity of a software unit's interface may vary greatly: a unit may consist of multiple classes and interfaces that expose fields and methods through a variety of mechanisms, such as, reference, method call, inheritance and interface implementation. Consequently, a general environment for a unit must be designed to accommodate all legal modes of external interaction. The environment will sequence those interactions to represent the behavior of program contexts in which the unit will be used in a larger piece of software.

Since model checking aims to exhaustively explore program execution paths, it is important for the environment used in model checking units to generate, to as large an extent as possible, the execution paths through the unit that will occur when the unit is actually deployed as part of a larger software system. Constructing such environments for the purpose of model checking is surprisingly difficult. For example, [18] note that it took several months to construct an environment that correctly modeled the context of the DEOS real-time operating system scheduler. Furthermore, in recent work, Engler and Musuvathi [7] describe proper environment construction as one of the main impediments to applying model checking as opposed to other techniques including static

analysis for bug-finding in source code. Our work on the Bandera Environment Generation tools [22, 21] provides basic support that addresses many of the challenges encountered in the DEOS case study, but it does not treat the complexities that arise when model checking units with JML specifications. In this setting one encounters additional questions: how complete is the test harness when it comes to stressing all the behaviors referred to by the specification?, and dually, how complete are the specifications with respect to all the behaviors that are exercised by the test harness upon the module?

Since an environment determines the behaviors of the system that are explored during model checking it directly influences the cost of checking and the ability to find bugs. In this paper, we seek to build on extensive work done in the testing community and emphasize various types of *coverage* as a means of (a) determining the suitability of an environment, and (b) guiding the construction and refinement of environments. A variety of strategies for developing environments based on coverage are possible, for example, generalizing a given environment to increase coverage or adding additional environments that provide complementary coverage. Previous work on model checking has used code coverage information to guide heuristic search strategies [9] and to determine the completeness of analysis of large software systems including TCP implementations [17]. In this paper, we view the collection and use of coverage information as an integral part of a model checker's implementation and application, focusing specifically on its interaction with the checking of strong specifications written in JML. In particular, we use coverage information to not only determine adequacy of the environment, but we also distinguish the notions of *code coverage* and *specification coverage* for the purpose of addressing several interesting questions about the relationships between the code of the unit, the specification of the unit, and the environment(s) used to check a unit.

- In addition to determining the code coverage for a given unit with respect to an environment, what are appropriate notions of coverage for a specification?
- For effective bug-finding with respect to a specification, are both high specification coverage and code coverage required?
- If there is a mismatch between code and specification coverage, what revisions of the environment, code or specification might this suggest?
- How is the cost of checking and the ability to find bugs sensitive to the number and generality of environments used to achieve a given level of coverage?

To experiment with these ideas, and to begin to answer some of these questions, we have built an extensible framework in Bogor called **MAnTA** (**M**odel-checking **An**alysis of **T**est-harness **A**dequacy) for implementing the collection of a variety of forms of coverage information. We describe how we have used MAnTA to implement two coverage estimation procedures and show how we have used them to find coverage holes in some of our models, triggering the refinement of the test harnesses for these systems. We summarize lessons learned and attempt to lay out a methodology that uses coverage information to help developers build environments that are appropriate for model checking units with strong specifications such as can be written in JML.

The rest of the paper is organized as follows. Section 3 discusses the general architecture of the MAnTA framework, its major components, and how it can be configured for effective estimation of coverage metrics in Bogor. In section 4 we show the results

of experiments conducted using MAnTA to implement two different types of coverage estimation analyses and briefly discuss the benefits and implications of these results to the application of coverage metrics in model checking. Section 5 presents some preceding work in the area of coverage metrics estimation for model checking as well as some other related works. Finally, in section 6 we conclude giving some conclusions drawn from the experiences in this research.

2 The Need for Coverage Metrics in Model Checking

Slogans associated with model checking proclaim that it represents "*exhaustive* verification" and that it "explores all possible states" have misled many users to believe that if a model checker completes with no specification violations reported then the system is free of errors. We believe that it is commonplace for errors in specifications and environment encodings to lead to "successful" checks that are essentially meaningless, since they explore only a tiny portion of the behavior of the system.

Consider Figure 1 which shows excerpts of a concurrent implementation of a linked list from [15]. To analyze this code we must construct an environment. In general, one strives to develop environments that:

- are concurrent in order to expose errors related to unanticipated interleavings
- are non-deterministic in ordering method calls in order to expose errors related to unanticipated calling sequences

Unfortunately, the degree of concurrency and non-determinism in an environment are precisely the factors that lead to exponential explosion in the cost of model checking. For these reasons, users of model checkers typically develop restricted environments that have a limited number of threads where each thread implements only a specific pattern of method calls. We took this approach when developing the environment shown in Figure 2.

When we fed the linked queue code with the driver to the model checker, the model checker reported no specification violations. Upon examining MAnTA's output, we found that the code coverage was rather low. The problem lies in the method `run()` of the class `Process`. All processing done by each thread reduces to inserting an item into the list and then removing an item from the list. After some assessment, it is not so hard to see that by doing this, every thread is guaranteed that prior to extracting an element from the list, on any execution trace, the list is never empty.

The linked queue class implements the following blocking discipline: when a client tries to extract an object and the list is empty, it will block the client thread and make it wait until there is something (see method `take()`). However, as shown in Figure 3, all the code that implements the blocking behavior of the list is never executed by our given test harness (the figure shows all the code that was not covered by the test harness in a shaded area). This code was never exercised because the method is never called in a context where the list is empty. Thus, when using a model checker to analyze code units which are closed with the addition of an environment , a "no violations" result from the model checker cannot by itself give confidence that the unit being analyzed satisfies its specification.

```
public class LinkedQueue {
 final /*@ non_null @*/ Object putLock;
 /*@ non_null @*/ LinkedNode head;
 /*@ non_null @*/ LinkedNode last;
 int waitingForTake = 0; ...
 //@ invariant waitingForTake >= 0;
 //@ invariant \reach(head).has(last);
 /*@ behavior
   @   assignable head, last, putLock,
   @                waitingForTake;
   @   ensures \fresh(head, putLock) &&
   @           head.next == null; @*/
 public LinkedQueue() {
  putLock = new Object();
  last = head = new LinkedNode(null);
 }
 /*@ behavior
   @   ensures \result <==>
   @           head.next == null; @*/
 public boolean isEmpty() {
  synchronized (head)
   return head.next == null;
 }
 /*@ behavior
   @     requires n != null;
   @     assignable last, last.next; @*/
 void insert2(LinkedNode n) {
  last.next = n;
  last = n;
 }
 /*@ behavior
   @  requires x != null;
   @  ensures true;
   @ also behavior
   @  requires x == null;
   @  signals (Exception e) e instanceof
   @    IllegalArgumentException; @*/
 public void put(Object x) {
  if (x == null)
   throw new IllegalArgumentException();
  insert(x);
 }
 synchronized Object extract() {
  synchronized (head) return extract2();
 }
 /*@ behavior
   @  assignable head, head.next.value;
   @  ensures \result == null
   @    || (\exists LinkedNode n;
   @        \old(\reach(head)).has(n);
   @        n.value == \result
   @        && !(\reach(head).has(n))); @*/
 Object extract2() {
  Object x = null;
  LinkedNode first = head.next;
  if (first != null) {
   x = first.value;
   first.value = null;
   head = first;
  }
  return x;
 }
 /*@ behavior
   @  requires x != null;
   @  ensures last.value == x
   @          && \fresh(last); @*/
 void insert(Object x) {
  synchronized (putLock) {
   LinkedNode p = new LinkedNode(x);
   synchronized (last) insert2(p);
   if (waitingForTake > 0)
    putLock.notify();
locSpec2:
   return;
  } }
}
class LinkedNode {
 public Object value;
 public LinkedNode next; ...
 /*@ behavior ensures value == x &&
   @                 next == null; @*/
 public LinkedNode(Object x) {
  value = x;
 }
}
```

Fig. 1. A Concurrent Linked-list-based Queue Example (excerpts)

Using the results of MAnTA, we were able to redesign the environment to increase the level of coverage significantly. In this case, we generalized the environment to allow for more general calling sequences in the thread.

With a model checker alone, there is no tool support to indicate potential problems with the environment. We believe that this problem is especially important in our context where we are model checking JML specifications which typically specify more details about a program's state than specification languages traditionally used in model checking. Accordingly, we seek to develop forms of automated tool support not only for environment generation (as in our earlier work [21]), but also for providing feedback on the quality of environments in terms of both code coverage and *specification coverage*.

3 MAnTA Architecture

With MAnTA, we seek to provide a flexible framework that will allow different coverage metrics to be included as an integral part of the model checking process. Users

```
package linkedqueue;

public class LinkedQueueDriver {

 /*@ behavior
   @     ensures true;
   @*/
 public static void main(String[] args) {
  LinkedQueue lq = new LinkedQueue();
  new Process(lq).start();
  new Process(lq).start();
 }
}
```

```
class Process extends Thread {
 /*@ instance invariant lq != null; @*/
 final LinkedQueue lq;

 /*@ behavior
   @     requires lq != null;
   @     assignable this.lq;
   @     ensures this.lq == lq;
   @*/
 Process(LinkedQueue lq) {
  this.lq = lq;
 }

 /*@ also
   @ behavior
   @     diverges true;
   @     ensures false;
   @*/
 public void run() {
  Object data = new Object();
  while (true) {
   lq.put(data);
   data = lq.take();
  }
 }
}
```

Fig. 2. A test harness for the concurrent linked queue program

```
public Object take() {
  Object x = extract();
  if (x != null)
    return x;

  else {
    synchronized (putLock) {
      try {
        ++waitingForTake;
        for (;;) {
          x = extract();
          if (x != null) {
            --waitingForTake;
            return x;
          } else {
            putLock.wait();
          }
        }
      } catch (InterruptedException ex) {
        --waitingForTake;
        putLock.notify();
        // throw ex;
        throw new RuntimeException();
      }
    }
  }

}
```

Fig. 3. Portion of method `take()` not executed by test harness

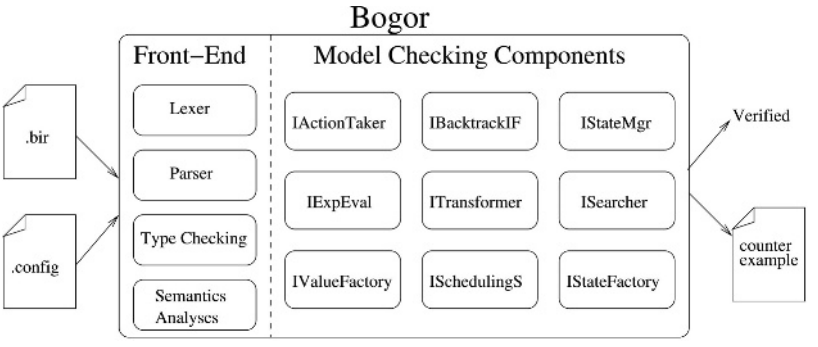


Fig. 4. Bogor Architecture

can add an extra layer of confidence to the verification process simply by activating Bogor's default coverage analyses through MAnTA options. In addition, users can easily include their own implementation of coverage estimation algorithms to customize coverage information for specific reasoning goals or to experiment with new coverage algorithms. Indeed, one of our primary goals in developing the MAnTA framework is to support our continuing research on exploiting coverage metrics in model checking.

MAnTA is built on top of the Bogor software model checking framework. Figure 4 shows the architecture of Bogor, as presented in [19]. The framework is composed of loosely coupled modules, each one enclosing a different aspect of the model checking process. The *ISearcher* controls the exploration of the state space graph. The *IExpEvaluator* is the module that computes the value of side-effect free expressions by accessing the heap and all relevant local activation records. Each module can easily be extended to add extra functionality. For example, in the work by Dwyer et al. in [5], the *ISearcher* was extended to add partial order reductions to the model checker. Also, in [20], the *IActionTaker* was extended to allow the verification of methods' frame conditions.

Building on this flexibility, MAnTA accumulates coverage information by monitoring specific events in any of the Bogor modules. This is achieved with the use of an Observer pattern [8]: MAnTA modules can subscribe to relevant events of specific Bogor modules. For example, the module for structural coverage described in section 4.1 subscribes to the *ITransformer* module and observes each transition that is executed, using this information to perform the analysis. Similarly, the specification analysis module in section 4.2 observes the commands executed by the *IActionTaker* looking for the execution of assertions.

Figure 5 shows the simple architecture of the MAnTA framework. The *IMantaManager* module takes care of initialization and finalization of all the modules that implement specific coverage estimation analyses. The *IMantaCoverageAnalyzer* interface provides the point of implementation for different coverage estimation algorithms. To incorporate a particular coverage analysis into MAnTA, one simply specifies the implementing classes and the manager loads them during the initialization period.

3.1 An Example

We illustrate the configuration aspects of the framework with the linked queue example from Figure 1. Suppose we wanted to model check this program and implement

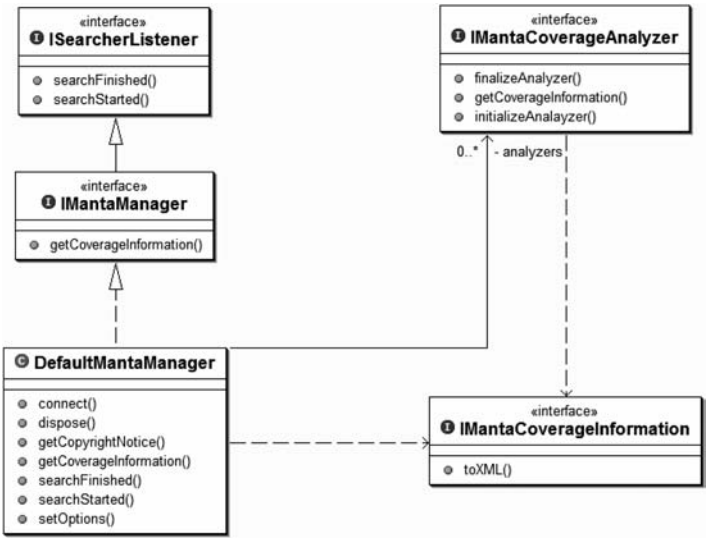


Fig. 5. MAnTA Architecture

two types of coverage analysis – the structural and specification analyses that we will describe in detail in Sections 4.1 and 4.2. First we specify options in the Bogor configuration file that indicate the implementation of the analysis manager (we will choose the default manager) and indicate the number of analysis modules that will be executed.

```
IMantaManager=DefaultMantaManager
IMantaManager.analyzers.size=2
```

Now, we have to specify the implementation classes that MAnTA will use for each analysis. Notice that MAnTA does not need to know about the specifics of every analysis. All it needs to know is the name of the classes that implement the analyses to be able to load them:

```
IMantaManager.analyzers.0=MantaEdgeCoverageAnalyzer
IMantaManager.analyzers.1=MantaSimpleBooleanCoverageAnalyzer
```

Finally we specify the options for each of the analyzers. Structural coverage does not require an option. The specification coverage analyzer takes a few options from the configuration file that indicate particular locations to be monitored. For example, suppose we want to monitor the post-condition of the method `insert(Object)` in the linked queue program. We specify the following options:

```
MantaSimpleBooleanCoverageAnalyzer.analyzeAll=false
MantaSimpleBooleanCoverageAnalyzer.monitoredLocations.size=1
MantaSimpleBooleanCoverageAnalyzer.monitoredLocations.0=insert(),locSpec2
```

The first option tells the analyzer that we are interested in specific locations and not in all the specification formulas. The second and third options specify the size of the set of locations that we want to monitor and the method name and location label for the program point that we want to monitor. To be able to refer to a specific location we simply label the program point with a Java label as shown in Figure 1. However, it is not

mandatory to refer to specific locations in order to monitor them. A common analysis will involve monitoring all the pre and post-conditions (and invariants). To do so, the three lines shown previously turn into:

```
MantaSimpleBooleanCoverageAnalyzer.analyzeAll=true
```

which tells the analyzer to monitor all pre-conditions, post-conditions and invariants checking.

At the end of the model checking run, the coverage manager invokes a finalization method on each module and collects the results. The coverage analyses results are reported from the analyzers to the manager in the form of an *IMantaCoverageInformation* object. This interface provides an uniform way to present the results, regardless of the type of analysis.

4 Case Studies

We have applied MAnTA to analyze coverage information during model checking of a collection of six Java programs with JML specifications. Model checking Java programs with Bogor is achieved by compiling Java byte codes into BIR (Bandera Intermediate Representation) [19]; BIR is a guarded-assignment language with primitives for defining all of the object-oriented features of Java. There is a direct correspondence between Java source lines, byte codes and sequences of BIR transitions. This facilitates the mapping of analysis results, including coverage information, calculated on the BIR model back to the input Java program. Figure 6 shows the BIR translation of the method `put(Object)` in the linked queue example. JML specifications are encoded as embedded assertions in the BIR model. Assertions are inserted during the translation process and correspond to JML pre and post-conditions, and invariants checking; only partial support for assertion generation from JML is implemented in the current toolset. For more details, we refer the reader to [20].

MAnTA, as a Bogor extension, operates on BIR models. In this section we describe our experiences with MAnTA while implementing two coverage analyses. The first example shows the implementation of structural coverage based on branch coverage. We report some interesting findings concerning some of our existing models and issues with their test harnesses. The second example is the implementation of a boolean satisfaction analysis. We report results for the same set of systems as the structural analysis case.

4.1 Structural Coverage Estimation with MAnTA

The first type of coverage analysis that we implemented on top of MAnTA was branch coverage. The classical definition of branch coverage is ensuring that each branch of a decision point (for example, an if statement) is executed at least once. For the BIR program model this is achieved by determining the proportion of transitions that are exercised during model checking. The information returned by the analysis is the percentage of transitions executed and information about the actual transitions that were not executed (location within the model, etc.).

The implementation of this analysis was very straightforward. As with each MAnTA analysis, the module implementing the analysis must implement the coverage analyzer interface of Figure 5. During the initialization period, the module constructs a

```
function {|LinkedQueue.put(java.lang.Object)|}
 ({|LinkedQueue|} [|r0|],
  {|java.lang.Object|} [|r1|])
{
 {|java.lang.IllegalArgumentException|} [|$r2|];
 {|java.lang.Object|} [|$r3|];
 boolean spec1;
 boolean spec2;
 loc locSpec0: live {}
  do invisible
  {
  }
  goto locSpec1;
 loc locSpec1: live { spec1, spec2 }
  do
  {
   assert([|r0|]./|LinkedQueue.head|\ != null);
   assert([|r0|]./|LinkedQueue.last|\ != null);
   assert([|r0|]./|LinkedQueue.putLock|\ != null);
   assert([|r0|]./|LinkedQueue.waitingForTake|\ >= 0);
   assert(Set.contains<({|java.lang.Object|})>
    (State.reachSet<({|java.lang.Object|})>
     ([|r0|]./|LinkedQueue.head|\),
     [|r0|]./|LinkedQueue.last|\)));
   spec1 := [|r1|] != null;
   spec2 := [|r1|] == null;
   assert(spec1 || spec2);
  }
  goto loc2;
 loc loc2: live { [|r1|], [|r0|], spec1, spec2 }
  when [|r1|] != null do
  {
  }
  goto loc6;
  when !([|r1|] != null) do
  {
  }
  goto loc3;
 loc loc3: live { [|$r2|], spec1, spec2 }
  do invisible
  {
   [|$r2|] := new ({|java.lang.IllegalArgumentException|});
  }
  goto loc4;
 loc loc4: live { [|$r2|], spec1, spec2 }
  invoke
   {|java.lang.IllegalArgumentException.<init>()|}
    ([|$r2|]) goto loc5;
 loc loc5: live { spec1, spec2 }
  do
  {
   throw [|$r2|];
  }
  goto loc5;
 loc loc6: live { spec1, spec2 }
  invoke virtual
   +|LinkedQueue.insert(java.lang.Object)|+
    ([|r0|], [|r1|]) goto locSpec2b;
 loc locSpec2b: live { spec1, spec2 }
  do invisible
  {
  }
  goto locSpec2;
 loc locSpec2: live { spec1, spec2 }
  do
  {
   assert([|r0|]./|LinkedQueue.head|\ != null);
   assert([|r0|]./|LinkedQueue.last|\ != null);
   assert([|r0|]./|LinkedQueue.putLock|\ != null);
   assert([|r0|]./|LinkedQueue.waitingForTake|\ >= 0);
   assert(Set.contains<({|java.lang.Object|})>
    (State.reachSet<({|java.lang.Object|})>
     ([|r0|]./|LinkedQueue.head|\),
     [|r0|]./|LinkedQueue.last|\)));
  }
  goto loc7;
 loc loc7: live {}
  do
  {
  }
  return;
 loc locSpec3: live {[|$r3|]}
  do invisible
  {
   assert ([|$r3|] instanceof
    ({|java.lang.IllegalArgumentException|}) && (spec2 && !spec1));
   throw [|$r3|];
  }
  goto locSpec3;
 catch ({|java.lang.Object|}) [|$r3|] at
  loc2, loc3, loc4, loc5, loc6, loc7 goto locSpec3;
}
```

Fig. 6. A Concurrent Linked-list-based Queue Example (excerpts)

set `NotExecuted`, containing references to all the possible transitions in the system. This module subscribe to the *ITransformer*; every time a transition is executed, the tranformer notifies the analyzer, and the analyzer removes the transition reference from the set `NotExecuted`.

We tested this coverage analysis implementation in all the models used in our work for [20]. All these programs are annotated with JML [16], which is one of the specification languages that we used for verifying properties in Bogor. Table 1 summarizes the results we obtained for each program. Most of the model checks exhibited high structural coverage. The reason none of the numbers are over 90% is because there is some dead code in each model that is generated by our compiler. We estimate that those models with coverage around 85% have around 95% coverage. They would not have 100% or close to it in part because our BIR models also model exceptional code, which is not exercised by our environments. It is possible, of course, to trigger exceptional code by defining appropriate environments. Two examples achieved significantly lower levels of coverage: *LinkedQueue*, from Figure 1, and *ReplicatedWorkers*.

When we model checked the *LinkedQueue* example with the structural coverage enabled we found a couple of surprises. First, we found an error in the model – specifically, in a fragment of the model derived from translating one of the JML specification annotations by hand to overcome a limitation in our compiler. An error in the hand translation yielded an assertion check that was short-circuited and, hence, never been

Table 1. Percentage of structural coverage for all the systems analyzed

Model	Total Number of Transitions	Number of Unexplored Transitions	Coverage Percentage
LinkedQueue	231	103	55.41%
BoundedBuffer	163	31	81.21%
RWVSN	238	34	85.71%
DiningPhilosophers	210	28	86.67%
ReplicatedWorkers	585	195	66.67%

checked. Second, as shown in Table 1, the coverage percentage for this program was very low (55.4%). It turns out that the environment for this program was inadequate for exercising certain implementation behaviors. As discussed in section 2, this was caused by failure to execute code in method `take()`.

The other program that also had low structural coverage is the *ReplicatedWorkers*. This time the low coverage was due to the fact that our compiler generates BIR code for abstract classes. Bogor only needs the concrete classes to perform the checking, therefore all the code generated from the abstract classes is never executed. Based on these observations, we are modifying the code generation of the compiler and the manner in which coverage information is accumulated to (a) avoid reporting spurious coverage problems (e.g., like those associated with abstract classes) and (b) incorporate options that allow the user to selectively mask non-coverage reports for different types of code such as exception handlers.

4.2 Boolean Coverage Estimation with MAnTA

Structural analysis was very useful in spotting environment limitations. However, as we mentioned in section 1, we are also interested in analyses that help extract information coverage at the specification level.

Coverage analysis at the specification level has been found useful in the past for specification refinement. For example, Hoskote et al. [13] show how they used a coverage analysis similar to the one described in [4] to refine the specification of several circuits from a microprocessor design. In our case, we are very interested in this type of analysis because of our goal of checking strong JML specifications. The type of specifications that can be expressed in JML are very rich in terms of expressing complex properties of heap allocated data, and in covering all the different behavioral cases of a method. Writing this type of specification is very hard and demands a lot of expertise. Even when written by an expert, the potential for overlooking or over-constraining aspects of method behavior is significant. We believe that coverage analysis can be successfully used for specification debugging.

In this section we describe an analysis we implemented for this purpose. This additional coverage estimation analysis is a simple boolean satisfaction analysis. Again, the implementation only involved developing the logic of the coverage analysis module and configuring the coverage framework to interact with the model checker. The coverage criterion was a simple monitoring policy: for each boolean expression, determine

Table 2. Number of formulas not satisfied during model checking

Model	Total Number of Monitored Formulas	Number of Unsatisfied Formulas
LinkedQueue	20	2
BoundedBuffer	15	0
RWVSN	14	0
DiningPhilosophers	40	0
ReplicatedWorkers	25	0

whether there are sub-formulas of the expression that are never satisfied. If there are such formulas, then report them as possible coverage problem, along with their position in the source code.

Table 2 summarizes the results of the analyses. The only program that showed lack of coverage under this criterion was the *LinkedQueue* example. Figure 7 shows the two methods that had a problem of low coverage.

This analysis uncovers problems in different methods than the structural coverage analysis. For method `put()` the tool reports that the condition **`x == null`** is never satisfied in the precondition. This is to be expected since this condition corresponds to exceptional behavior. In some sense, the structural coverage test also uncovered the same issue because it reported that the `throw` instruction was never executed. But the boolean analysis gives a more direct indication of the conditions that led to the `throw` code not being covered.

For the `extract2()` method the tool reports that **`\result == null`** never holds in the postcondition. After looking carefully we note that this condition corresponds to the case when the list is empty. We already determined, with the information from the previous test, that the list was never made empty. However, the problem has now been exposed at a different point. In the first test, we spotted the problem *from the code*. Now, we do so *from the specification*. This phenomenon suggests that these two techniques are complementary in the types of coverage problems they can find. They uncovered the same problem from different perspectives.

```
/*@ behavior
  @  requires x != null;
  @  ensures true;
  @ also behavior
  @  requires x == null;
  @  signals (Exception e) e instanceof
  @    IllegalArgumentException; @*/
public void put(Object x) {
 if (x == null)
  throw new IllegalArgumentException();
 insert(x);
}
```

```
/*@ behavior
  @  assignable head, head.next.value;
  @  ensures \result == null
  @    || (\exists LinkedNode n;
  @        \old(\reach(head)).has(n);
  @        n.value == \result
  @        && !(\reach(head).has(n))); @*/
Object extract2() {
 Object x = null;
 LinkedNode first = head.next;
 if (first != null) {
  x = first.value;
  first.value = null;
  head = first;
 }
 return x;
}
```

Fig. 7. Methods in `LinkedQueue` that had unsatisfied formulas

4.3 Discussion

The basic forms of coverage analyses we implemented have demonstrated that individual coverage analyses can be effectively integrated with model checking to provide useful information about the quality of environments. We believe that multiple coverage analyses, when aggregated, can yield additional benefit on top of what can be obtained by using them independently.

To see this, let us consider together the results obtained in Sections 4.1 and 4.2. For the *LinkedQueue* example, both analyses showed that there was a behavior, namely the blocking behavior of the list, that was not being exposed. On one hand, the structural analysis spotted this problem by pointing directly at the portion of code that was not being executed. On the other, the boolean analysis uncovered the same problem, but this time pointing at a specification case that was never enabled. Independently, the two analysis would give the same benefit: find a gap in the coverage produced by an environment. However, when run together they provide two extra benefits and we discuss them in turn.

First, the obvious extra benefit is the increase in the reliability of the reports given by each analysis. Certainly, if both analysis point at the same problem, then the chances are higher that the problem is actually a concern and not just some superficial issue (as discussed previously, some analyses may report spurious or superficial coverage problems, as is the case for structural analysis when it pointed to transitions corresponding to abstract classes). The second benefit, not so apparent, but extremely useful, is in the way in which, when working together, these two analyses can help not only by providing estimation of coverage, guiding in that way the refinement of the test harness, but also in the verification process itself by uncovering potential errors, both in the implementation and in the specification.

To see this last point keep in mind the duality of these two approaches. For each portion of the model not covered, every analysis should report some information in the form of transitions not-executed and conditions not-enabled, for structural and boolean analysis, respectively. Now, consider Figure 8-a. Suppose further that it is model checked with an environment such that the method is always called in a state where `b==true`. Obviously, the boolean satisfaction analysis will report that the pre-condition `!b` is never true. However, the structural analysis will report complete coverage. This mismatch suggests that the implementation is missing code that represents the specification that is not covered, otherwise that code would be reported as not executed. Or if the code is not missing, then it must contain bugs because it does not match the specification. In fact, when we examine this code we can see that the implementation is incomplete. As can be seen, results from analyses, that separate would be disconnected information, together have helped to find a bug.

Notice that a model checker would not be able to find this bug because the environment is incomplete in the first place. And the boolean analysis by itself only suggests that the environment should be refined. Indeed, after refining the environment the model checker would spot the error, but this would require another model checking run. On the other hand, by analyzing the information from the coverage analyses, we can find the same bug with much less effort.

Similarly, if we look at Figure 8-b, and if we consider the same environment, we see that in this case we have the opposite situation. Now, the structural analysis would

```
/*@ behavior
  @  requires b;
  @  ensures a == 1;
  @also
  @ behavior
  @  requires !b;
  @  ensures a == 2;
  @*/
public void m(boolean b) {
 if (b)
  a = 1;
}
```

(a)

```
/*@ behavior
  @  requires b;
  @  ensures a == 1;
  @*/
public void m(boolean b) {
 if (b) {
  a = 1;
 } else {
  a = 2;
 }
```

(b)

Fig. 8. Example methods that show how structural and boolean analysis can be complemented for finding bugs and debugging the specification

report that a portion of the method is never executed. However, the boolean satisfaction analysis would report that the specification is completely covered. Indeed, the implementation is correct with respect to the specification. However, the mismatch between the two analysis suggests the possibility that the specification is not complete. This must be true for otherwise there would be some portion of the specification that would not be covered because the state space that it represents is never explored. If the specification is already totally satisfied, it means that it is insensitive to the code that was not executed. In fact, when we look at the specification we see that, indeed, the specification is incomplete: it does not handle the case when `b == false`. All this would lead to a refinement of the specification to include the case that is not handled (if that is what we really want). In this case, the benefit is not in finding bugs in the implementation, but in *specification debugging*.

Coverage-Enabled Model Checking. All the concepts presented above suggest a general methodology for software verification using model checking extended with these coverage analyses:

- Run the model checker on the system to be verified with the two coverage analysis extensions described in this work.
- If the structural analysis reports complete coverage, but the boolean analysis reports incomplete coverage, then there is potential for an incomplete implementation (a bug caused by missing code for a specification case).
- If the structural analysis reports incomplete coverage, but the boolean analysis reports complete coverage, then there is potential for an incomplete specification (at the very least, the specification is insensitive to the code that was not covered).
- If both analyses report incomplete coverage, there should be a correspondence of non-covered cases from one analysis to the other (as in our example in sections 4.1 and 4.2). If such correspondence exists, then the coverage problems can be attributed to the test harness. Otherwise, there is potential for implementation/specification incompleteness, depending on which analysis has an unmatched non-coverage case.

Note that this extra benefit cannot be achieved if only the model checker is run, or if the analyses are included but done independently. For example, suppose we have

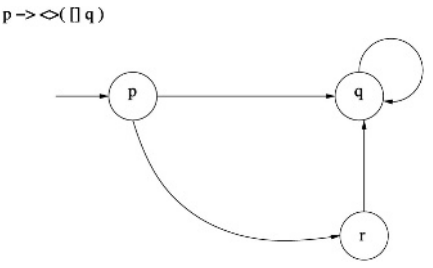


Fig. 9. Simple transition system and a specification to be verified

the case where the specification is incomplete and it does not cover some code that, additionally, is not executed by the test harness. The model checker would not detect the problem because from the point of view of the model checker there is no problem: the implementation satisfies the specification. The boolean analysis alone would not report any problem, and the structural analysis would simply point out that the test harness must be refined to cover the code not-executed.

Also, notice that this methodology will only detect incomplete specification cases when it is accompanied by the same incompleteness in the test harness. For example, in figure 8-b, if the test harness does cover the case when `b == false`, then both analyses, structural and boolean, achieve 100% coverage and nothing can be inferred. Spotting these specification incompleteness cases is another interesting direction that we want to pursue.

5 Related Work

Although coverage estimation has not received as much attention as other areas in model checking, there is still some work that has been done to address this problem. Among the most significant works in this area is the work done by Chockler et al. [4]. However, their definition of the coverage problem is slightly different. The concern in their work is the following: given a model and a specification, how much of the system state space is relevant to the specification. For example, consider the simple transition system in Figure 9. The specification shown with the system holds in the initial state because any of the two outgoing paths lead to the state where `q` is always true. However, the third state, where `r` is true, is totally insensitive to this specification. Under this definition of coverage we say that the node is not covered by the specification, so we have to refine the properties. To find these states, this technique would change the value of specific variables (for example, `p`) in each state and run the model checker. Those nodes in which the property does not fail, are nodes insensitive to the specification, and thus not covered.

The information obtained with this type of analysis gives a rough estimation of the proportion of behaviors covered by the specification. This approach, although useful, has several shortcomings. First of all, some specifications necessarily refer to only a portion of the states, due to the nature of the property they express. In this case, the

specification does not need to be refined and the coverage information is not very useful. Second, the cost of the analysis is the same as that of model checking itself (two model checks must be run – the first "regular" check, and then a second in which the values of the basic propositions are negated). However, as we show in this paper, by simply adding some book-keeping procedures to the model checker algorithm, some very useful coverage information can be gathered to support model refinement, at a much lower cost. For example, the structural analysis we implement simply needs to keep track of the transitions that are executed.

The combination of specification and structural coverage analysis yields results that are similar to that obtained by [4], but with much less effort. In that work, the analysis detects portions of the state space that are not covered by any aspect of the specification, pointing directly at those points of the state space. However, to do this an execution separate from the verification run must be made, which is as expensive as the model checking itself. On the other hand, our approach can be run while model checking the system, and the analysis information is done by monitoring the checking process. This monitoring is very inexpensive (with respect to one analysis, although the cost will grow as more analyses are attached to the framework). Nevertheless, our system won't point directly at the regions of the state space that are not covered and requires insight from the user to find this regions, once the evidence of the potential problem is found.

Another idea, explored by the same researchers mentioned in the previous paragraph, is the concept of vacuity checking [14]. This is a boolean satisfaction analysis. The idea is to verify that there are no formulas in the system that are being trivially satisfied, that is, that are proven correct by the model checker, but their truth is independent of the model. They show, surprisingly enough, how vacuity checking is harder to perform than one might imagine and that it can be very common in several models. Again, the usefulness of this analysis is limited just to some cases. Although vacuous formulas in specifications might be more common than expected, the calculation tends to be fairly expensive, yielding information about just one specific type of errors. For example, the approach described in the previous paragraph would also detect this formulas because the whole system would be insensitive to the formula. To see this, just consider what happens if `p` is made false in the initial state of the system in Figure 9. In this case, the formula is vacuously satisfied in all the states of the system, and would be detected because every state would be insensitive to this formula. Although it would be more expensive, the cost would be amortized by all the other type of errors that this approach would also find.

Musuvathi and Engler have used some coverage metrics in the verification of a TCP implementation using model checking [17]. Their coverage metric is based on simple line coverage, that is, whether a line of code is executed or not. However, their usage of the metric and motivation is quite different from ours. They use the coverage metrics to evaluate the effectiveness of the state space reduction techniques implemented in their model checker. However, they do note how identifying unexplored portions of the state space can help refine the test harness used to model check a system. Similarly, SPIN [11] has had for some time a feature through which it reports statements that are never reached.

In another similar line of work, Groce and Visser explore the use of coverage metrics to make heuristic-based model checking [9]. The coverage metric they use is branch coverage, which is exactly the one used in our structural coverage analysis. However, the purpose of this work is to use the information to both estimate the effectiveness of *partial checks* and use the branch coverage measure as a value function to guide the search in heuristic model checking.

Although the coverage metrics reported in the works mentioned in the previous two paragraphs are equivalent to our structural coverage, there are several differences in the usage of the metrics and in the motivations. The main difference is that, while those focus on using the metric to obtain a measure of how much of the state space is explored (useful in bounded search or when the model checker runs out of memory, i.e., partial checks), our focus is to use the metric to refine test harnesses used to model check open systems. Also, we have integrated the structural coverage with a specification coverage and proposed exploiting this integration to enhance the verification. This has to do with the main motivation for our work: we want to check *strong specifications* in software units and are deeply interested in verifying the depth to which the specifications are covered. Our interest in refining the test harnesses is focused towards increasing specification coverage rather than code coverage. This is why we have integrated the structural analysis with a boolean analysis for the specification.

The Bandera Environment Generator (BEG) project [21], takes another approach to providing tool support for deriving appropriate test harnesses and environments. It focuses on automatically generating test harnesses from high-level specifications of orderings of calls to the module being tested and from information gathered from automated static analysis (e.g., about side-effects and aliases) of the code being analyzed. It lets the user specify: (a) the components that are relevant for the verification, (b) constraints over these components, and (c) certain assumptions about the environment (such as the order in which methods in the unit being analyzed should be called), and then generates an environment made up of stubs and driving code that then is translated to the model checker language. Although the work in BEG is certainly relevant to the coverage issue, it is rather complementary to the work presented herein: BEG tries to generate adequate environments that achieve good coverage from user provided information (such as a regular expression indicating a pattern of method calls to the unit under test), whereas MAnTA estimates the coverage achieved by a given environment (whether it is generated automatically or by hand). For example, even though the user provides a regular expression to specify the ordering of unit method calls in the test harness, this ordering may still not give good coverage (and MAnTA can be used to detect these situations).

The coverage problem has been studied extensively by the testing community [2]. In this case, all the studies made in test coverage, for example [23], are directly relevant to the work presented in this paper. Several notions of structural coverage have been developed by the testing community. For example, there is the concept of node coverage, branch coverage, path coverage, etc. All this work includes potential lessons to be applied in monitoring coverage in model checking.

6 Conlusions

This paper presented MAnTA – a coverage analysis framework built on top of the Bogor software model checking framework to guide effective construction of environments for model checking. MAnTA allows a variety of coverage analyses of both code (e.g., branch coverage) and specification (e.g., boolean satisfaction of method pre/post-conditions) to be integrated in the model checking process. Since model checking is exhaustive (i.e., all interleavings are considered), thus, the results of the coverage analyses can pinpoint deficiencies in the environments used in the system. For example, a branch analysis can be used to determine code regions that are not exercised due to the specificity of the environment. Another example, the exercised behaviors of a particular method may not be sufficient because it always satisfy *a specific* pre-condition of the method instead of *each* of the pre-conditions of the method.

MAnTA eases the incorporation of these kinds of coverage analyses by providing a plugable API for adding new coverage analyses. We showed the effectiveness of the framework by easily incorporating widely used coverage analyses in the testing community as well as by implementing a novel specification coverage analysis. We showed how the analyses can uncover common deficiencies in test harnesses with respect to code and specification. Therefore, we believe that MAnTA can be used to guide analysts when constructing environments such as usually done in unit testing.

References

1. T. Ball and S. K. Rajamani. The slam toolkit. In *Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264. Springer-Verlag, 2001.
2. R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.
3. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
4. H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. A practical approach to coverage in model checking. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 66–78. Springer-Verlag, 2001.
5. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 2004. (to appear).
6. M. B. Dwyer, J. Hatcliff, and D. Schmidt. Bandera: Tools for automated reasoning about software system behaviour. *ERCIM News*, 36, Jan. 1999.
7. D. R. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference*, volume 2937 of *Lecture Notes in Computer Science*, pages 191–210. Springer, 2004.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Pub. Co., Jan. 1995.
9. A. Groce and W. Visser. Model checking java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.

10. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
12. G. J. Holzmann and M. H. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st international conference on Software engineering*, pages 597–607. IEEE Computer Society Press, 1999.
13. Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Proceedings of the 36th ACM/IEEE Design automation conference*, pages 300–305. ACM Press, 1999.
14. O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. In *10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1999.
15. D. Lea. *Concurrent Programming in Java: Second Edition*. Addison-Wesley, 2000.
16. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop*, Oct. 1998.
17. M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proceedings of The First Symposium on Networked Systems Design and Implementation*, pages 155–168. USENIX Association, 2004.
18. J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos scheduler kernel. In *Proceedings of the 22nd international conference on Software engineering*, pages 488–497. ACM Press, 2000.
19. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
20. Robby, E. Rodríguez, M. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 404–420. Springer, 2004.
21. O. Tkachuk, M. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 18th International Conference on Automated Software Engineering*, Oct. 2003.
22. O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proceedings of the Fourth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Sept. 2003.
23. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Combining Several Paradigms for Circuit Validation and Verification

Diana Toma, Dominique Borrione, and Ghiath Al Sammane

TIMA Laboratory, VDS Group,
46 Avenue Felix Viallet, 38031 Grenoble, France
{diana.toma,dominique.borrione,ghiath.al-sammane}@imag.fr

Abstract. The early validation of components specifications requires a proven correct formalization of the functional behavior. We use the ACL2 theorem prover the establish safety properties on it. After the first design step, we automatically translate the synthesizable VHDL into a functional form. The combination of symbolic simulation, automatic transfer function extraction, and theorem proving is used to show that the VHDL design is functionally compliant to the specification. The approach is demonstrated on a SHA-1 cryptographic circuit.

1 Introduction

Ensuring the correctness of circuits for safety-critical applications requires a rigorous design flow. Ideally, a formalized functional specification should first be thoroughly validated, and subsequent design steps should be proved correct, until the "synthesizable register transfer level" is reached, from which automatic synthesis software produces the physical design. In reality, the time pressure put over the designers is so high that verification software must be largely automatic to be effectively adopted.

The numeric simulation of test cases will probably remain the primary method to gain initial confidence in the design, at all levels of abstraction. This is particularly true for the validation of the initial behavioural specification, which corresponds to the first machine-readable model. Depending on the task, the type of circuit, and the initial specification level, one among many possible languages are used. Popular ones are: SystemC for mixed hardware/software systems, Matlab for hardware DSP operators, CCS or CHP for asynchronous designs, Verilog or VHDL for register transfer level (RTL) hardware.

A large number of errors are revealed by numeric simulation; but to provide a better quality assurance, formal methods are subsequently invoked, to validate initial specifications, or prove the correctness of a design step. Starting from RTL and below, where the design is expressed in terms of logical types, propositional logic functions for combinational circuits and finite state machines (FSM) for sequential circuits are routinely extracted from the design description. Efficient combinations of logic function representations (BDD's), symbolic algorithms (state space traversal, SAT solving, recursive learning, test generation)

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 229–249, 2005.

and model reduction techniques are currently implemented in commercial software. Designers have reported verifying combinational circuits with hundreds of ports and millions of gates, by checking their equivalence with a logic level "golden" circuit [5]. Control logic with several hundred memorizing bits has been submitted to property checkers: the validity of temporal logic formulas is computed on the reachable states of the underlying FSM model.

These Boolean level techniques are not applicable at the initial phases of a project, where the specification is expressed in terms of arithmetic operations and high-level algorithms. Here, the data types and the number of objects are not bounded, and the underlying model has not a fixed size. Yet, it is essential to establish (1) the functional validity of the specification and of the initial implementation choices, before investing extensive effort at more detailed levels; (2) that the manually derived synthesizable RTL correctly implements the specification. Both needs can be successfully addressed with mechanized theorem proving systems, which present reasoning capabilities, in particular induction, that free the proof argument from the data size complexity.

Writing the first formal model of the functional specification is, by necessity, a manual process. The source information comes from specification documents written in the traditional mixture of English sentences, drawings, timing diagrams, and expected responses to test scenarios. This is the reason why it is of particular importance, for the credibility of this model, that it be both efficiently executable on numerical values, and input to a reasoning engine that can prove mathematical and safety properties on that same model. We use the ACL2 theorem prover [4] for its ability to reuse libraries of pre-verified function definitions and theorems, for its high degree of automation, and its efficiency. The ACL2 model, being written in LISP, is both executable and formally verifiable in logic. To validate the functional specification, we first execute its formal Lisp model on the standard test benches to check that the returned result is as expected; then, we prove sanity theorems and basic mathematical properties on the Lisp model. This provides, as far as possible, a "correct" functional specification.

Building the synthesizable RTL is also a manual process, during which all the essential design choices are performed, leading to a model that is extensively simulated on numeric tests. In this paper, we consider VHDL as design language, but all the following would apply to Verilog as well. To prove that the RTL correctly implements the specification, no simple equivalence can be exhibited, as encoding details and timing information have enriched the model. To perform the verification, a functional model must be extracted from the RTL design, prior to showing the existence of a "correct implementation relation" between selected circuit output and state variables on the one hand side, and the result returned by the specification on the other hand side, taking into consideration the necessary type conversions and circuit computation cycles. To automate this task, we translate the RTL design into a functional format, simulate the model symbolically for one clock cycle, in effect corresponding to several simulation iterations, and extract the transition function for each output and state variable of the design. On the fly simplifications are performed by rewrite rules. The

resulting functional model, thus mechanically extracted from the RTL design, is proved compliant to the Lisp specification, using the ACL2 theorem prover.

In this paper, we present the semantic foundations for this method, and illustrate its application on the design of a cryptographic component for a secure smart card reader. We show how the SHA hash algorithms have been specified, keeping as parameter the input message length, block size, and word size of the circuit. A design is proven to correctly implement the SHA-1, for arbitrary messages. The article is organized as follows. Section 2 describes the running benchmark application. Section 3 summarizes the specification validation. Section 4 describes the principles of the automatic state machine extraction from a VHDL design, and section 5 is an overview of its application to the SHA-1 implementation verification; sections 4 and 5 constitute the core, and the original contribution of this paper. Finally, section 6 relates our approach to previous works and presents our conclusions.

2 The SHA Algorithm

The Secure Hash Algorithm (SHA) is a standardized hash function [1], which processes an arbitrary input message of bounded size seen as a bit stream, and produces a short fixed size message digest, with the following property: any alteration to the initial input message will result, with a very high probability, in a different message digest. Several versions of the SHA differ in the maximum input message size (2^m with m=64 or 128), component word size (32 or 64 bits), and message digest size (d bits, with d = 160, 256, 384 or 512). The applications of this algorithm include fast encryption, password storage and verification, computer virus detection, etc. Our interest in the SHA was motivated by the cooperative project ISIA2 with several industrial partners, aiming at the definition, design and verification of a circuit for secure communications between a computer and a terminal smart card reader. A SHA-1 component, designed at L2MP, is included in the circuit. Security considerations were at the heart of the project. It was thus of utmost importance to guarantee the correctness of the system components dedicated to security, and formal methods were applied both to the validation of the functional specification, and to the verification of the implementation.

The principle of the SHA-1 is shown on Figure 1. The input message M, a bit sequence of arbitrary length L< 2^{64}, undergoes two preprocessing steps:

- **Padding:** M is concatenated by bit 1, followed by k bits 0, followed by the 64-bit binary representation of number L. k is the least non-negative solution to the equation: (L+1+k) mod 512 = 448. As a result, the padded message holds on a multiple of 512 bits.
- **Parsing:** The padded message is read in blocks of 512 bits. After reading each block, it must be decided if it is the last block.

The **computation of the message digest** is an 80-iteration algorithm over each message block, in order; a block is viewed as a sequence of 32 bit

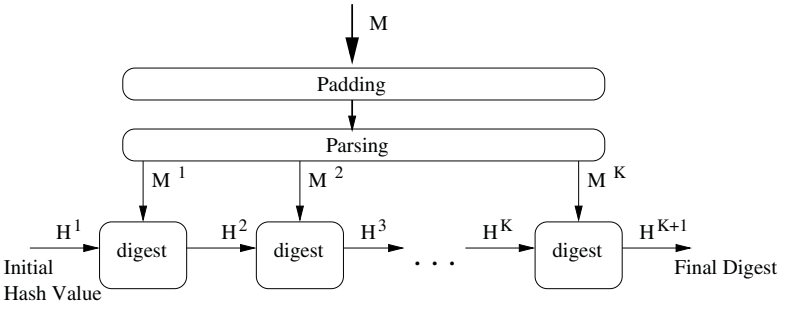


Fig. 1. Secure Hash Algorithm

words, which are selected and combined with the contents of five 32-bit internal registers, using XOR and shift operations. At the start of the computation, the internal registers are initialized with predefined constants; after processing each block, they contain the digest obtained so far, that serves as initial values if the message holds more blocks.

According to the SHA-1 standard [1], the digest phase operates on the 16 words W_i $(0 \leq i \leq 15)$ of a padded block in order to generate 80 words. The first sixteen words are made of the padded block itself. The 64 remaining words are computed by XORing and shifting previous words as follows, where S^n indicates a n-bit left shift operation:

for t=16 to 79
W_t=S^1 (W_{t-3} XOR W_{t-8} XOR W_{t-14} XOR W_{t-16}) endfor.

Likewise, the five main variables A, B, C, D, E are generated by:

for t=0 to 79 do
TEMP=$S^5(A)$ + f_t (B, C, D) + E + W_t + K_t;
E=D; D=C; C=S^{30}(B); B=A; A=TEMP;
endfor.

where f_t are the functions and K_t the constants defined in the SHA-1 standard. Before entering the loop, the main variables A, B, C, D, E are initialized by constant values H_0, H_1, H_2, H_3, H_4 that are specified in the standard.

When one block has been processed, the values of H_i are:
$H_0 = H_0$+A; $H_1 = H_1$+B; $H_2 = H_2$+C; $H_3 = H_3$+D; $H_4 = H_4$+E;

When the last block has been processed, the message digest is the concatenation of the last values of H_0, H_1, H_2, H_3, H_4.

In order to store only sixteen W words and not eighty, the SHA-1 standard proposes an alternate algorithm where the W_t and the five main variables are computed in the same loop, and where each W_t starting from t=16 is written in place of $W_{t\ mod\ 16}$. This alternate version was chosen to optimize the hardware space.

Example: Consider the SHA-1 algorithm applied to the message "abc". The length of the message is $8 \times 3 = 24$. The padded message is: "abc", followed with bit 1, followed with $448 - (24 + 1) = 423$ bits 0, and then the message length. The resulting 512-bit padded message is:

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ 1\ \overbrace{00..00}^{423}\ \overbrace{00..011000}^{64}$$

The message digest (in hexadecimal) for "abc" is:

A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D.

Validation of the Formal Functional Specification

The SHA-1 algorithm is formalized in Lisp; the detailed model can be found in [2], and is executed on the test benches given in the standard document to check that the returned result is as expected. A complementary validation is obtained by proving the mathematical properties of the algorithm, using the ACL2 theorem prover. In effect, a more general model has been written, to capture the common principles of the four versions of the SHA algorithm: SHA-1, SHA-256, SHA-384 and SHA-512, which differ essentially in the sizes of the message blocks, word, and digest. Seventy function definitions and over a hundred basic lemmas were written. Among the safety theorems that were proven for the SHA-1:

- The length of the padded message is a non-zero multiple of 512.
- The last 64 bits of the padded message represent the binary coding of the length.
- The first L bits of the padded message represent the initial message.
- The bits between the end-of-the-message bit and the last 64 bits are all 0.
- After parsing the padded message, the result is a vector of blocks, each of 512 bits.
- The final result of the SHA-1 is a five 32-bit words message digest.

Due to the nature of the digest computation, there is no easy to write mathematical expression for it. The Lisp code of the iterative algorithm is the function definition.

3 Extraction of the Formal Model from the VHDL

The RTL design is written in the so-called synthesizable subset of VHDL [3] Two data abstraction levels are usually considered: (1) the logic level, where all data is encoded as bits and bit-vectors; (2) the high-level, where control states remain symbolic, and arithmetic operations operate on mathematical integers. The extraction of a Mealy finite state machine (FSM) from a logic-level RTL design is at the basis of all sequential verification tools [6],[7]. At this level, all memorizing variables are considered state elements, the control and the operative parts are not distinguished. Despite the introduction of "word level decision

diagrams", drastic data reduction must be performed on the source text before proceeding with the FSM extraction, lest we face a combinational explosion of the model representation. The method thus preferably applies to control logic verification. A formal and complete treatment can be found in [8].

In contrast, despite the existence of various formal semantics proposals for VHDL [9], and of the use of theorem proving techniques to verify *soundness properties on the language* definition [10], the authors are not aware of an efficient FSM extraction tool for the *implementation verification* of high-level circuits that are data oriented. Models such as Finite State Machine with Data path have proved useful for the high-level synthesis of control-dominated circuits [11], and the verification of the synthesized resulting RTL [12]. However, to our knowledge, no such model could be proved compliant to an abstract algorithmic specification, essentially due to the lack of adequate link to an appropriate proof system. This is precisely where our work takes all its significance.

The method we developed takes as input a clock synchronized sequential circuit, and builds a finite state machine, where all states transition functions and output functions are written in a normalized conditional format, and the time unit is the clock cycle. In the absence of memorizing element, according to the semantics of synthesizable VHDL [3], the combinational circuit is a special simple case that has no state and no state transition function. The originality of our method is the *static stabilization* of signal propagation in combinational logic, which corresponds to a re-ordering and fixed-point computation over the concurrent signal assignment statements of the VHDL model.

3.1 Rewriting the Source Model into an Elementary Language

We recall that a VHDL description is a **component**, defined by an **entity** declaration together with a corresponding **architecture** body (boldface are VHDL reserved words). The entity declares the interface signals, and possibly defines constraints on the inputs, and some **type** and **constant** declarations. The architecture imports the entity declarations, and may have its own declarative part: local types, functions, procedures, constants, embedded components and signals. From the entity and the architecture declarative part, we extract the lists of input, output and local signals, and store their characteristics (i.e. name, type, initial value, etc).

The architecture statement part describes the behavior of all these objects. It is a set of *concurrent* statements: processes, signal assignments, component instantiations, **generate** statements that macro-generate repetitive or conditional statement blocks. It is essential to understand that the architecture statements are concurrent and unordered: the language semantics ensure that the results of a simulation run do not depend on the occurrence order of concurrent statements. In particular, a **signal** has a current value and one or more future values (only one future value is needed in the synthesizable subset that we consider in this paper). A signal assignment computes its right-hand side expression with the current values of signals, and assigns the future value of its left-hand side. When all signal assignments and all processes have been computed in all the

components of the description, this is the end of one simulation cycle. Current values are replaced by future values, and an **event** is detected for each signal that had a current value change. At any point in simulated time, several simulation cycles may be necessary to propagate signal changes in combinational logic.

Inside a **process**, a **function** or a **procedure**, statements are *sequential*, with semantics similar to usual programming languages, except for **wait** synchronization statements, which are specific to simulation. In particular, a locally declared **variable** has only one value, and an assignment to a variable immediately assigns this unique value.

Because of the syntactic complexity of VHDL, we identified a small list of most primitive statements, in terms of which the more elaborate ones can be expressed. We rewrite sequential statements (**wait**, **assert**, **signal** assignment, **variable** assignment, **if**, **while**, **for**, **procedure** call, **case**) and concurrent statements (sequential **process**, **signal** assignment, **component** instantiation, **generate** statement) to a subset that contains only signal and variable assignments, component instantiations, procedure calls, if statements and processes. This first transformation greatly simplifies the subsequent extraction of the design functional model.

Rewriting Sequential Statements

Let $Sequential_{VHDL}$ be the sublanguage of sequential statements. It is derived from the *Seq* non terminal symbol, according to the abstract syntax defined in Table 7 (Appendix A). Let $Seq\_Normalize_{VHDL}$ be the sequential sublanguage constructed over the reduced set of primitive sequential statements: variable assignments, signal assignments, if conditional and procedure call. $Seq\_Normalize_{VHDL}$ is derived from the *Seq_Norm* non-terminal symbol defined in Table 7 (Appendix A).

Function *Seq_Rewrite* rewrites a block of sequential statements into a normalized sequential block, according to the following principle:

- signal assignments, variable assignments and procedure calls are left unchanged
- assert statements, wait synchronizations, for loops, case conditionals and elsif clauses are rewritten into primitive if...then...else conditionals.

Rewriting Concurrent Statements

Similarly, let $Concurrent_{VHDL}$ be the sublanguage of concurrent statements, derived from the *Conc* non terminal symbol in Table 8 (Appendix A). Let $Conc\_Normalize_{VHDL}$ be the concurrent sublanguage constructed over the reduced set of primitive concurrent statements: signal assignments, component instantiations and process statements. $Conc\_Normalize_{VHDL}$ is derived from the *Conc_Norm* non-terminal symbol (Table 8, Appendix A)).

Function *Conc_Rewrite* rewrites a block of concurrent statements into a normalized concurrent block as follows:

- simple signal assignments, and component instantiations are left unchanged
- conditional and selected signal assignments are given a simple nested if...then ...else equivalent right-and side expression (not compliant to the VHDL syntax)
- for...generate and if...generate are expanded
- *Seq_Rewrite* is called on the statement part of all processes

The full inductive definitions of *Seq_Rewrite* and *Conc_Rewrite* are given in Appendix A.

3.2 Modeling the Normalized Primitive Language

The extraction of the functional design semantics takes as input the result of the application of *Seq_Rewrite* and *Conc_Rewrite*, and produces a list of assignments, exactly one for each signal and variable assigned in the design. This resulting list of assignments models the computation performed by one simulation cycle.

An assignment takes the form $NextSig$(s, IF_expression) if s is a signal, $ChangeVar$(v, IF_expression) if v is a variable: $NextSig$ replaces the future value of the signal with the value specified by IF_expression, whereas $ChangeVar$ replaces the current value of the variable. IF_expression may be a constant, Boolean, arithmetic or if-then-else conditional expression.

For readability, we define the target functional language for the semantic model in pseudo-code, and assume bool_expression and arith_expression to be self-explanatory. Table 1 gives the abstract syntax. In our implementation, a Lisp syntax is adopted.

Table 1. Language of the target semantic model

Seq_Assign ::= **nil** | $Nextsig(s, IF\_expression)$ | $ChangeVar(v, IF\_expression)$
| $Seq\_Assign^1; Seq\_Assign^2$
IF_expression ::= bool_expression | arith_expression
| $IF(condition, IF\_expression_1, IF\_expression_2)$
Conc_Assign ::= $Nextsig(s, IF\_expression)$ | $Conc\_Assign^1; Conc\_Assign^2$

Modeling the sequential behavior

$\mathbf{Trans}_{Seq} : Seq\_Normalize_{VHDL} \longrightarrow Seq\_Assignments$

Let *Seq_Assignments* be the sublanguage derived from *Seq_Assign* (see Table 1).

Function $\mathbf{Trans}_{Seq}$ takes as input a list of sequential statements in a process, and produces the list of single assignments to each modified signal and variable. The difficulty consists in transforming several assignments to a same object, under successive, and possibly non disjoint conditional statements, into a single assignment, where all conditions are grouped in the right-hand side. $\mathbf{Trans}_{Seq}$ is defined by structural induction over the primitive sequential statements.

Table 2. Semantic function extraction for sequential statement blocks

1. $\mathbf{Trans}_{Seq}(target <= expression) \stackrel{\triangle}{=} NextSig(target, expression)$
2. $\mathbf{Trans}_{Seq}(target := expression) \stackrel{\triangle}{=} ChangeVar(target, expression)$
3. $\mathbf{Trans}_{Seq}$ ($procedure\_name[(parameter\_list)]$) $\stackrel{\triangle}{=}$ the application of the $\mathbf{Trans}_{Seq}$ to the body of the procedure, and the replacement of the object names with the corresponding ones given in the call.
4. $\mathbf{Trans}_{Seq}$ (**if** *condition* **then** Seq_Norm **endif** $\stackrel{\triangle}{=}$ $\mathbf{Distribute}(IF(condition, \mathbf{Trans}_{Seq}(Seq\_Norm), \mathbf{nil}))$
$\mathbf{Trans}_{Seq}$ (**if** *condition* **then** Seq_Norm^1 **else** Seq_Norm^2 **endif** $\stackrel{\triangle}{=}$ $\mathbf{Distribute}(IF(condition, \mathbf{Trans}_{Seq}(Seq\_Norm^1), \mathbf{Trans}_{Seq}(Seq\_Norm^2)))$
5. $\mathbf{Trans}_{Seq}$ ($Seq\_Norm^1; Seq\_Norm^2$) $\stackrel{\triangle}{=}$ $\mathbf{Group}(\mathbf{Trans}_{Seq}(Seq\_Norm^1), \mathbf{Trans}_{Seq}(Seq\_Norm^2))$

In Table 2:

- Cases 1 to 3 are straightforward.
- Case 4 corresponds to the processing of a conditional statement. First, the statement is transformed into an intermediate $IF(condition, then - part, else - part)$, and $\mathbf{Trans}_{Seq}$ is called recursively on the $then - part$ and the $else - part$, both in $Seq\_Normalize_{VHDL}$. Then, the resulting IF form, where all leaf statements are calls to $NextSig$ or $ChangeVar$, is fed to function *Distribute* which moves the *condition* to the right-hand sides of all assignments of the $then - part$ and the $else - part$. A single assignment is produced for objects that are assigned in both the then-part and the else-part. The details of function *Distribute* can be found in Appendix B.
- Case 5 corresponds to consecutive sequential blocks. After translation by $\mathbf{Trans}_{Seq}$, two lists of assignments are produced. Objects assigned in the first list must be rewritten in the right-hand side of assignments in the second list. Double assignments are eliminated in the process. This transformation is done by function *Group*, fully defined in Appendix B.

Example: In the following small sample of the control part of SHA (Table 3), describing the behavior of the component in the final state (cnt_reset), we show a VHDL nested if statement, and its transformation into two signal assignments.

Modeling the Concurrent Behavior

$$\mathbf{Trans}_{Conc} : Conc\_Normalize_{VHDL} \longrightarrow Conc\_Assignments$$

Let *Conc_Assignments* be the sublanguage derived from *Conc_Assign* (see Table 1): it is simply a list of invocations to function $Nextsig$, performing concurrent signal assignments. Function $\mathbf{Trans}_{Conc}$ is applied to the statement part

Table 3. Transformation of if-statement into signal assignments

VHDL	Translated assignments

if CNT='0' then
  if bl="000000" then
  done<='1'; etat<=idle;
  else etat<=init;
  end if;
else etat<=cnt_reset;
end if;
	NextSig (etat,
	  IF (cnt = '0',
	  IF (bl="000000", idle, init),
	  cnt_reset))
	NextSig (done,
	  IF (cnt='0' and bl="000000",
	  '1', *NextSig* (done)))

of a component architecture, which is a block of concurrent statements, and produces the list of assignments to all local and output signals of the component. $\mathbf{Trans}_{Conc}$ is defined by structural induction over the primitive concurrent statements. In Table 4:

- Case 1 corresponds to the simple signal assignment with a (possibly conditional) expression. According to the VHDL semantics, the assignment is performed only if there was an event on one or more signals in the right-hand side expression; a signal has an event at a simulation cycle if its value at the previous simulation cycle is different from the current one. This inserted condition is denoted: $\mathbf{Event}(Sensitivity(cond\_expression))$. *Sensitivity* returns the list of signals in an expression, i.e. the sensitivity signals for the target.
- Case 2 is the concurrent process with a sensitivity list (list of signals such that an event on one of them resumes the process execution). Function $\mathbf{Trans}_{Seq}$ is first invoked on the sequential statements block in the process, distributing the sensitivity list condition on all statements. Then **Update** replaces all occurrences of operator $ChangeVar$ with $Nextsig$, and all occurence of the memorizing mark $Nextsig(s)$ with s, for all signals assigned in the process.
- Case 3 is the component instantiation. Label is the name of the instance, and the actual generic values, and input-output ports of the component are provided at this point. Local objects are prefixed with the instance name, to guarantee unique naming, as several instances of a component may be used in a model.
- Case 4 stands for concurrent statement lists, it is straightforward.

3.3 Symbolic Simulation

Function **Step** computes the value (IF_expression) of the memory and output signals of the design, after one simulation cycle. It is the direct functional model of the VHDL simulation semantics, and consists in the composition of all the $NextSig$ constructed in the previous section:

$\mathbf{Step} : \mathbf{I} \times \mathbf{M} \times \mathbf{O} \longrightarrow \mathbf{M} \times \mathbf{O}$
$\mathbf{Step}\ (Input, Memory, Output) =$
$\quad \mathbf{Trans}_{Conc}(Conc\_Rewrite(architecture\_body))$

Table 4. Semantic function extraction for concurrent statement blocks

1. **Trans**$_{Conc}$ ($target$ <= cond_expression) $\stackrel{\triangle}{=}$ $NextSig(target$, IF (**Event** ($Sensitivity$ (cond_expression)), **Trans**$_{Conc}$ (cond_expression), $target$)
2. **Trans**$_{Conc}$ ([$label$:] **process** [($sensitivity\_list$)] [**is**] $process\_declarative\_part$ **begin** Seq_Norm **end process** [$label$]) $\stackrel{\triangle}{=}$ **Update** (**Distribute** (IF (**Event** ($sensitivity\_list$), **Trans**$_{Seq}$ (Seq_Norm), **nil**)))
3. **Trans**$_{Conc}$ ($label$: $component\_name$ [**generic map** ($association\_list$)] [**port map** ($association\_list$)] $\stackrel{\triangle}{=}$ in-line replacement of the component instantiation by the result of the application of **Trans**$_{conc}$ on the corresponding architecture statement block, renaming alllocal objects X by label/X, and replacing all the generic parameters and ports by values provided in the respective $association\_list$
4. **Trans**$_{Conc}$ ($Conc^1_{Bl}; Conc^2_{Bl}$) $\stackrel{\triangle}{=}$ **Trans**$_{Conc}$($Conc^1_{Bl}$); **Trans**$_{Conc}$($Conc^2_{Bl}$)

Usually a VHDL design needs several simulation cycles (called delta cycles) to become stable. We adopt symbolic simulation to stabilize the design by executing the following algorithm:

```
for simulation_time=1 to time_max do
    Input = Get_test_vectors (simulation_time);
    Do (Memory_next,Output_next) = Step(Input, Memory, Output);
        Memory_last = Memory; Memory = Memory_next;
        Output = Output_next;
    while Event (Sync_signals);
    Input_last = Input;
end for;
```

For each design object $obj \in Input \cup Memory \cup Output$, we define three variables: obj_last (the value at the previous cycle), obj (the current value), and obj_next (the new value being computed); all are initialized with a symbolic value that corresponds to their names. $Sync\_signals \subseteq Input \cup Memory$ is the set of (assignment and process) sensitivity list signals.

At each simulation cycle, the input values can be numeric or symbolic. The current expressions of memory signals and outputs are computed using the already extracted and normalized assignments. During this computation, the expressions are simplified using static rules; some of these rules are shown in Table 5. Then, the simulated objects are updated. The computation is repeated if an event occurs in one of the synchronization signals.

The inputs are considered stable during the internal stabilization loop. The designer controls the synchronizing inputs at each time point. For clock-synchro-

Table 5. IF-expression simplification rules

$IF(True, Y, Z) \longrightarrow Y$
$IF(False, Y, Z) \longrightarrow Z$
$IF(IF(A, B, C), X, Y) \longrightarrow IF(A, IF(B, X, Y), IF(C, X, Y))$
$IF(X, Y, Y) \longrightarrow Y$
$f_n\ (A_1, A_2, ..., IF(X, Y, Z)...A_n) \longrightarrow$
$\quad IF(X, f_n(A_1, A_2, ..., Y, ..., A_n), f_n(A_1, A_2, ..., Z, ..., A_n))$

nized designs, the model of interest is the result of two simulation times, when the clock successively takes value '1' and '0', i.e. a clock cycle.

3.4 The State Machine

Function **Sim_step** models the state machine transition function. **Sim_step** takes as parameters the inputs of the design and the state of the machine (i.e. the memory and output signals) at simulation cycle k, and produces the state of the machine at simulation cycle $k + n$ (k, n are naturals, n is the number of simulation cycles needed for the stabilization).

$$\textbf{Sim\_step} : \mathbf{I} \times \mathbf{M} \times \mathbf{O} \longrightarrow \mathbf{M} \times \mathbf{O}$$
$$\textbf{Sim\_step}\ (Input, Memory, Output) \stackrel{\triangle}{=} (NextSig\_s_1(parameter\_list_1), ..., NextSig\_s_n(parameter\_list_n))$$

The body of **Sim_step** is the composition of the IF_expressions obtained by symbolic simulation. For each $s_i \in Memory \cup Output$, we define a function $NextSig\_s_i(parameter\_list) = IF\_expression_i$, where $IF\_expression_i$ is the final $IF\_expression$ computed in s_i by the symbolic simulation of the previous section.

The general state machine is defined as a recursive function that takes a sequence of inputs (one symbolic value/input signal/clock cycle) and an initial state and returns the state obtained after consuming all inputs.

$$\textbf{System} : \textbf{Input\_Seq} \times \mathbf{M} \times \mathbf{O} \longrightarrow \mathbf{M} \times \mathbf{O}$$
$$\textbf{System}\ (nil, Memory, Output) \stackrel{\triangle}{=} (Memory, Output)$$
$$\textbf{System}\ ((Input.Input\_Seq), Memory, Output) \stackrel{\triangle}{=} \textbf{System}(Input\_Seq, \textbf{Sim\_step}(Input, Memory, Output))$$

We use Mathematica to perform the symbolic simulation and extract the functional model, to benefit from its pattern-matching algorithm, rewriting features, and efficient symbolic computation needed during the stabilization phase. Functions **Sim_step** and **System** are translated to Lisp, and subsequently input to the ACL2 theorem prover. As Mathematica uses a functional language, the translation to Lisp is mainly a syntactic one. Slight modifications in object names are done as needed to avoid conflicts with ACL2 key words. Standard

VHDL operations on Boolean and bit vectors are replaced with corresponding operations defined and proved correct in ACL2.

We have developed a "book" (library of functions and proved ACL2 theorems) on bit vectors that models the *Numeric_bit* VHDL package (two's complement binary arithmetic, high-order bit on the left, with logical, arithmetic, shift, and numeric conversion operations).

Along with the functions above, information about inputs and state variables are translated to Lisp and two predicates are created: **hyp_input**(input), which states the type for each input element of the design, and **hyp_mem**(mem), which states the type for each state variable of the design.

4 Case Study: SHA-1

The VHDL description of SHA-1 is clock-synchronized, so a step of the state machine model corresponds to a clock cycle. In the VHDL design provided to us, all outputs are memorizing, so the model is a Moore machine.

The SHA-1design is intended for use with a RAM, that holds the message blocks; so the RAM is added to the state of the Moore machine. We model the RAM as a list of pairs (address, cell_content), where address is a symbol. The message to be processed by the SHA algorithm starts at address *base*. Function **sha_vhdl**, defined similarly to **System**, simulates the circuit. It takes two parameters: the sequence of inputs *L-input* and the state *st*. *st* is composed of three parts: *memory* is the set of all internal signals of the SHA-1, *out* is the set of output signals, and *ram* models the external RAM. The length of *L-input* gives the number of clock cycles, and *L-input* represents the list of symbolic or numeric values for the SHA input ports at each clock cycle:

(inputs_cycle-1 inputs_cycle-2 ... inputs_cycle-k)

If the inputs list is empty, the computation is finished and **sha_vhdl** returns the state *st*. Otherwise, the next state is computed, and *memory*, *out* and *ram* are updated, by calling the step function **Sim_step**. Although effectively generated in Lisp, we give function **sha_vhdl** in pseudo code, for legibility. Again, this model is executable, and we have initially checked it using the test benches provided in the SHA standard.

```
function sha_vhdl(L_input, st) returns state is
   if empty(L_input) return st;
   else memory=st[0]; out=st[1]; ram=st[2]; inp=car(L_input);
        (new_ram,new_out)=
                  Sim_step(concat(read_ram(memory, ram), inp), memory, out);
        new_ram= write_ram(new_mem, ram);
        return sha_vhdl(cdr(L_input), concat(new_mem, new_out, new_ram));
   endif
```

At this point, we have two models of SHA-1 in the functional Lisp subset of ACL2: the translation by hand of the standard FIPS-180-2 **sha_norm** (20 functions) which has no timing information, and the automatic translation of

the VHDL description **sha_vhdl** (35 functions), which is clock cycle accurate. Clearly, **sha_norm** involves a time abstraction with respect to **sha_vhdl**, and the two models are not directly equivalent.

To prove that the VHDL implementation is compliant to the functional specification, we must show that, whatever the input message, the execution of **sha_vhdl** for the appropriate number of clock cycles (until the computation is done) returns the same message digest as the one returned by **sha_norm**. The main theorem states:

For any positive integer *nb*, for any message of *nb* blocks, after the execution of **sha_vhdl** for 3+(342**nb*) clock cycles, the system is in its final state (*done*=1) and the values of the state variables H_0, H_1, H_2, H_3, H_4 are equal to the result of **sha_norm** on the same message.

The inputs are the symbolic vectors of Table 6, where X stands for "don't care", *nb_bv* is the binary representation of *nb* on 6 bits, *base* is a bit-vector of size 12. *nb_bv* and *base* are symbolic.

Table 6. The symbolic input for *sha_vhdl*

Cycle	**1**	**2**	**3**	...
Input	*input_cycle_1*	*input_cycle_2*	input	
Reset	1	0	0	...
Start	X	1	X	...
Reset_done	X	X	X	...
Nb_bloc	X	nb_bloc	nb_bloc	...
Base_addr	X	X	base_addr	...

Sha-vhdl needs 3 clock cycles to initialize the system and set a, b, c, d, e to their initial values; then it needs 342 clock cycles to compute the digest for one block. The 342 cycles are decomposed as: 16 for reading the first 16 words, 320 to compute an intermediate digest, 3 to combine the results with the initial hash values of the block, 2 to store the message digest obtained so far. The last cycle returns to the digest computation for the next block, or to the idle state.

The above could let the reader believe that we are performing simulation. This is not the case. Although the actual number of cycles is precisely depicted in our model, the reasoning engine considers the initial value of all memories and registers as arbitrary, and *nb* to be an unbounded (but finite) natural integer. Induction is performed over *nb*. The key intermediate theorem used in the induction is: starting from state *init*, after 342 clock cycles, if the number of blocks to be processed is higher than 0, the system is in the *init* state, otherwise it is in the *idle* state, and in both cases H_0, H_1, H_2, H_3, H_4 hold the same digest as computed by **sha_norm**.

In turn, to prove the above theorem, a stepwise approach must be adopted, which proves that each main computation step of the overall **sha_vhdl** is equivalent with some intermediate digest function. Then, the compliance between these intermediate functions and the **sha_norm** must be proved.

Here is an overview of the stepwise approach (it follows the VHDL state machine of SHA):

- Starting from state *init*, after 1 cycle the system is in the *sha_init* state and the memory has not changed;
- Starting from state *sha_init*, after 16 cycles, the system is in the *compute_w* state and the first 16 steps of the digest algorithm are computed;
- Starting from state *compute_W*, after 320 cycles, the system is in the *result* state, the RAM is modified and the state variables a, b, c, d, e, hold the result of the last 64 steps of the digest computation:
- Starting from state *result*, after 3 cycles, the system is in *cnt_reset*, the number of blocks to be processed is decremented and the state variables a, b, c, d, e are added to the state variables $h0$, $h1$, $h2$, $h3$, $h4$, which hold the hash values during the computation.
- Starting from state *cnt_reset*, after 2 cycles, if the number of blocks to be processed is higher than 0, the system is in *init* state, otherwise it is in the *idle* state, *done* is 1 and the values of a, b, c, d, e are available as output.

Few theorems are proved by symbolic execution. Most of them are proved by generalization followed by induction. All theorems are using a large number of properties that we proved about bit-vectors, about operations with bit vectors (logical, arithmetic, concatenation, shifting, conversions, etc), and about the RAM. The total proof, including the two models, needed 150 functions and 750 theorems, from which 45% are reusable. One of the difficulties of the proof was to find the intermediate digest functions. Also the generalization and the definition of the right induction schemes required some experience with the prover.

5 Related Works and Conclusion

Symbolic simulation was proposed in 1979 by J. Darringer [5], but the early implementations could not handle large circuits for lack of effective simplification techniques; when the condition is a symbolic term, all alternative paths of conditional statements had to be explored, and the simulation tree grew exponentially.

The BDD representation of Boolean expressions gave a new start to symbolic techniques. At switch and gate-level, Bryant and Seger proposed the successful "symbolic trajectory evaluation" (STE) [14], a restricted form of model checking for a limited class of LTL formulas: states are abstracted with ternary logic (encoded with BDD pairs), and symbols are used to encode the inputs and parts of the initial state. To reduce the size explosion of BDD's, generalized graph structures, the inclusion of SAT and the use of quaternary logic have enhanced the capacity of generalized STE to enable the verification of FIFO memory [15]. Derived from these works, a commercial symbolic simulator from Innologic can handle realistic size circuits, still described at the logic and switch level.

These technologies are not applicable at the initial design levels, when a specification involves abstract data types rather than their Boolean encoding.

To simplify symbolic simulation, reduce algebraic expressions and control the expansion of the simulation tree, most proposed solutions use an automated reasoning tool. The PVS reasoning system has been experimented on the Java processor JEM1 produced at Rockwell Collins Advanced Technology [16]. The automatic theorem prover ACL2 was shown to be more efficient to implement a symbolic simulator for the same application [17]. Taking Common Lisp as input format for its models, ACL2 can execute the simulation both numerically and symbolically, which is a practical advantage. A systematic approach for using ACL2 as a symbolic simulation engine was proposed by J. Moore as a support to the previous reference [18]. On this base, the simulation semantics of a subset of VHDL were defined in ACL2 in order to simulate a VHDL design symbolically [3].

The work discussed in this paper aims at providing a more efficient and more compositional model for conventional design languages. Stabilizing combinational signals dynamically, during symbolic simulation, in ACL2 was too space consuming. The rewriting and fixed point computation capabilities of Mathematica brought an elegant and automatic solution to the problem of computing the transfer function for combinational operations at compile time. By combining Mathematica for symbolic simulation [20] and the construction of the state machine model, and ACL2 for reasoning about this model, and proving that it correctly implements an abstract function, we have been able to prove a real design correct. To our knowledge, this is the first mechanized verification flow to apply theorem proving technology and check that a design written in VHDL correctly implements an algorithm. We stress two important characteristics of the formal model on which we reason: (1) it is executable, and can be applied to numeric test cases for conventional debugging; (2) its complexity is independent of the bit width of the data objects.

Despite the fact that the formal model is automatically generated from the design description, the VHDL writing style has a large impact on the quality of the semantic model produced by our prototype system. In addition, expert knowledge is needed to direct the proof, and the use of this technology involves a significant initial investment. Our ongoing work includes the application of the tool to a wide variety of circuit types (on-chip communications, specialized operators, DSP, etc), and the development of the appropriate ACL2 libraries, with the objective of providing modeling guidelines and proof strategies for use by verification engineers outside our research group.

Acknowledgments

The MEDEA+, MESA and RNTL ISIA2 projects provided partial support.

References

1. National Institute of Standards and Technology: “Secure Hash Standard”, Federal Information Processing Standards Publication 180-2, 2002.
2. D. Toma, D. Borrione: “SHA Formalization”, Proc. ACL2 Workshop, Boulder, USA, 13-14 July 2003.

3. IEEE CS: 1076.6-1999 IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, New York. March 2000.
4. M. Kaufmann, P. Manolios, and J S. Moore: Computer-Aided reasoning: ACL2 An approach.(Vol.1) and ACL2 Case Studies (Vol.2), Kluwer Academic Press, 2000.
5. W. Roesner: "What Is Beyond The RTL Horizon for Microprocessor and System Design?", Keynote, CHARME'03, L'Aquila, Italy, 21-24 October 2003.
6. O. Coudert, C. Berthet, J.C. Madre: "Verification of synchronous sequential machines based on symbolic execution", in Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science No407, Springer, pp.365-373.
7. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill: "Sequential circuits verification using symbolic model checking", Proc. 27th Design Automation Conf., IEEE CS Press, 1990.
8. D. Déharbe: "Vérification formelle de propriétés temporelles: étude et application au langage VHDL", PhD, Université Josph Fourier, Grenoble, 15 nov. 1996 (in French).
9. C. Delgado Kloos, P. T. Breuer, ed: "Formal Semantics for VHDL", International Series in Engineering and Computer Science, No307, Kluwer, 1995.
10. W. Damm, B. Josko, R. Schlor: "Specification and Verification of VHDL-based System-level Hardware Designs", in E. Berger ed. Specification and validation methods for programming languages and systems, Oxford University Press, 1995, pp. 331-410.
11. D. Gajski, L. Ramachandran: "Introduction to High-level Synthesis", IEEE Design and Test of Computers, Vol. Winter, 1994, pp. 44-54.
12. D. Borrione, J. Dushina, L. Pierre: "A Compositional Model for the Functional Verification of High-level Synthesis Results", IEEE. Trans. on VLSI, Vol8, No5, Oct. 2000, pp. 526-530.
13. J. Darringer. "Application of Program Verification Techniques to Hardware Verification". In Proc. IEEE-ACM Design Automation Conference, pp. 375-381, 1979.
14. R. E. Bryant and C. J. H. Seger. "Formal Verifcation by Symbolic Evaluation of Partially-ordered Trajectories". Formal Methods in System Design, 6(2):pp. 147-189, March 1995.
15. J. Yang and C.I. Seger. "Generalized Symbolic Trajectory Evaluation Abstraction in Action". Proc. FMCAD, Portland, USA, Nov 2002. Springer LNCS 2517, pp. 70-87.
16. D. Greve. "Symbolic Simulation of the JEM1 Microprocessor". Proc. FMCAD'98, Palo Alto, CA, 1998. Springer LNCS 1522, pp. 321-333.
17. M. Wilding, D. Greve, and D. Hardin. "Efficient Simulation of Formal Processor Models". Formal Methods in System Design 2001.
18. J. S. Moore. "Symbolic Simulation: An ACL2 Approach." Proc. FMCAD'98, Palo Alto, CA, 1998. Springer LNCS 1522, pp. 334-350.
19. P. Georgelin: "Vérification formelle de systémes digitaux synchrones, basée sur la simulation symbolique". PhD Univ. Joseph Fourier, Grenoble, France, 2001 (in French).
20. G. Al Sammane, D. Borrione: "Formal Validation of High Level Specification of Data-path Digital Circuits by Symbolic Simulation" Proc. DDECS'03, Poznan, Poland, April 2003.

Appendix A: Rewrite Rules for the VHDL Statements

Rewriting Sequential Statements

Table 7. Abstract syntax of sequential statements in VHDL

Seq ::= *target* <= *expression* | *target* := *expression* | **null** | [*label*] **assert** *condition* [**report** *expression*] [**severity** *expression*] | procedure_name [(*parameter_list*)] | **if** *condition* **then** Seq [elsif_statement] **endif** | **case** *condition* **is** case_alternative **end case** | **for** *identifier* **in** $expression_1$ direction $expression_2$ **loop** Seq **end loop** | Seq^1 ; Seq^2 | [*label*] **wait on** *sensitivity_list* ; Seq | [*label*] **wait until** *condition* ; Seq elsif_statement ::= **else** Seq | **elsif** *condition* **elsif** Seq elsif_statement case_alternative ::= **when** choice => Seq | **when** choice => Seq case_alternative direction ::= **to** | **downto** choice ::= *expression* | **others** Seq_Norm ::= *target* <= *expression* | *target* := *expression* | procedure_name [(*parameter_list*)] | **if** *condition* **then** Seq_Norm^1 [**else** Seq_Norm^2] **endif** | Seq_Norm^1 ; Seq_Norm^2

$Seq\_Rewrite : Sequential_{VHDL} \longrightarrow Seq\_Normalize_{VHDL}$

Seq_Rewrite (target <= expression) $\stackrel{\triangle}{=}$ target <= expression

Seq_Rewrite (target := expression) $\stackrel{\triangle}{=}$ target := expression

Seq_Rewrite (procedure_name [(parameter_list)]) $\stackrel{\triangle}{=}$
procedure_name [(parameter_list)]

Seq_Rewrite ([label] **assert** condition [**report** expression]
[**severity** expression]) $\stackrel{\triangle}{=}$
if condition **then** label:=true **else** label:= false **endif**

Seq_Rewrite (**if** condition **then** Seq [elsif_statement] **endif**) $\stackrel{\triangle}{=}$
if condition **then** *Seq_Rewrite* (Seq)
[**else** *Seq_Rewrite* (elsif_statement)] **endif**

The definition of *Seq_Rewrite* for the sub-block elsif_statement is given below:

Seq_Rewrite (**else** Seq) $\stackrel{\triangle}{=}$ *Seq_Rewrite* (Seq)

Seq_Rewrite(**elsif** condition **then** Seq elsif_statement) $\stackrel{\triangle}{=}$
if condition **then** *Seq_Rewrite* (Seq)
else *Seq_Rewrite* (elsif_statement) **endif**

Seq_Rewrite(**case** expression **is when** choice => Seq **end case**) $\stackrel{\triangle}{=}$
if (expression = choice) **then** *Seq_Rewrite* (Seq) **endif**

Seq_Rewrite (**case** expression **is when** others => Seq **end case**) $\stackrel{\triangle}{=}$
Seq_Rewrite (Seq)

Seq_Rewrite (**case** expression **is**
when choice => Seq case_alternative **end case**) $\triangleq$
if (expression = choice) **then** *Seq_Rewrite* (Seq)
else *Seq_Rewrite* (**case** expression **is** case_alternative **end case**) **endif**

Seq_Rewrite (**for** identifier **in** $expression_1$ **to** $expression_2$ **loop** Seq **end loop**) $\triangleq$
identifier := $expression_1$; *Seq_Rewrite* (Seq);
identifier := succ($expression_1$); *Seq_Rewrite* (Seq);
identifier := $expression_2$; *Seq_Rewrite* (Seq)
When direction is **downto**, the predecessor function pred replaces the successor function succ.

Seq_Rewrite (Seq^1 ; Seq^2) $\triangleq$ *Seq_Rewrite* (Seq^1) ; *Seq_Rewrite* (Seq^2)

Seq_Rewrite ([label] **wait on** sensitivity_list ; Seq) $\triangleq$
if *Event*(sensitivity_list) **then** *Seq_Rewrite* (Seq) **endif**
Seq_Rewrite ([label] **wait until** condition ; Seq) $\triangleq$
if *Event*(condition) **and** condition **then** *Seq_Rewrite* (Seq) **endif**

Rewriting Concurrent Statements

Table 8. Abstract syntax of concurrent statements in VHDL and their normalized form

Conc ::= [*label*:] **process** [(*sensitivity_list*)] [**is**] *process_declarative_part*
begin Seq **end process** [*label*]
| *target* <= *conditional_forms*
| **with** *expression* **select** *target* <= selected_forms
| *label*: generation_scheme **generate** Conc **end generate** [*label*]
| *label*: *component_name* [**generic map** (*association_list*)]
[**port map** (*association_list*)]
| $Conc^1$; $Conc^2$
conditional_forms ::= *expression*
| *expression* **when** *condition* **else** conditional_forms
selected_forms ::= *expression* **when** *choice*
| *expression* **when** *choice* selected_forms
generation_scheme ::= **for** *identifier* **in** $expression_1$ direction $expression_2$
| **if** *condition*
Conc_Norm ::= *target* <= cond_expression
| *label*: *component_name* [**generic map** (*association_list*)]
[**port map** (*association_list*)]
| ([*label*:] **process** [(*sensitivity_list*)] [**is**] *process_declarative_part*
begin Seq_Norm **end process** [*label*])
| $Conc\_Norm^1$; $Conc\_Norm^2$
cond_expression ::= bool_expression | arith_expression
| **if** *condition* **then** $cond\_expression^1$ [**else** $cond\_expression^2$ **endif**

$Conc\_Rewrite : Concurrent_{VHDL} \longrightarrow Conc\_Normalize_{VHDL}$
$Conc\_Rewrite$ ([*label*:] **process** [(*sensitivity_list*)] [**is**] *process_declarative_part*
begin Seq **end process** [*label*]) $\triangleq$
([*label*:] **process** [(*sensitivity_list*)] [**is**] *process_declarative_part*
begin *Seq_Rewrite*(Seq) **end process** [*label*])

$Conc\_Rewrite$ (target <= conditional_forms) $\triangleq$
target <= $Conc\_Rewrite$ (conditional_forms)

$Conc\_Rewrite$ (expression) $\triangleq$ expression

$Conc\_Rewrite$ (expression **when** condition **else** conditional_forms) $\triangleq$
if condition **then** expression
else $Conc\_Rewrite$ (conditional_forms) **endif**

$Conc\_Rewrite$ (**with** expression **select** target <= selected_forms)$\triangleq$
target <= $Conc\_Rewrite$(**with** expression **select** selected_forms)

$Conc\_Rewrite$ (**with** $expression_1$ **select** $expression_2$ **when others**)$\triangleq$
$expression_2$

$Conc\_Rewrite$ (**with** $expression_1$ **select** $expression_2$ **when** choice) $\triangleq$
if $expression_1$=choice **then** $expression_2$ **endif**

$Conc\_Rewrite$ (**with** $expression_1$ **select** $expression_2$ **when** choice select_forms) $\triangleq$
if $expression_1$=choice **then** $expression_2$
else $Conc\_Rewrite$ (**with** $expression_1$ **select** select_forms) **endif**

$Conc\_Rewrite$ (label: **for** identifier **in** $expression_1$ direction $expression_2$
generate Conc **end generate** [label]) $\triangleq$
replace identifier with $expression_1$ in $Conc\_Rewrite$ (Conc);
replace identifier with succ($expression_1$) in $Conc\_Rewrite$ (Conc);
replace identifier with $expression_2$ in $Conc\_Rewrite$ (Conc)
Similary when direction is **downto**.

$Conc\_Rewrite$ (label: **if** condition **generate** Conc **end generate** [label]) $\triangleq$
if condition **then** $Conc\_Rewrite$ (Conc) **endif**
$Conc\_Rewrite$ (label: *component_name* [**generic map** (*association_list*)]
[**port map** (*association_list*)]) $\triangleq$
label: component_name [**generic map** (association_list)]
[**port map** (association_list)]

$Conc\_Rewrite$ ($Conc^1$; $Conc^2$) $\triangleq$
$Conc\_Rewrite$ ($Conc^1$) ; $Conc\_Rewrite$ ($Conc^2$)

Appendix B: Functions for the Construction of the Semantic Model

Distribute transforms an if_statement into a list of sequential assignments.
IF_Statements = {IF (condition, Seq_Assign^1, Seq_Assign^2) with
Seq_Assign^1 , Seq_Assign^2 $\in$ Seq_Assignments}
Distribute: IF_Statements $\longrightarrow$ Seq_Assignments

Distribute (IF (condition, Seq_Assign^1 , Seq_Assign^2)) $\triangleq$ Seq_Assign^3
where Seq_Assign^3 is defined below:

- if signal s is assigned in the two branches:
  NextSig (s, $expression_1$) $\in$ Seq_Assign^1 and
  NextSig (s, $expression_2$) $\in$ Seq_Assign^2,
  then NextSig (s, IF (condition, $expression_1$, $expression_2$)) $\in$ Seq_Assign^3.
- if variable v is assigned in the two branches:
  ChangeVar (v, $expression_1$) $\in$ Seq_Assign^1 and
  ChangeVar (v, $expression_2$) $\in$ Seq_Assign^2,
  then ChangeVar (v, IF (condition, $expression_1$, $expression_2$)) $\in$ Seq_Assign^3.
- if variable v is assigned only in one branch:
  ChangeVar (v, $expression_1$) $\in$ Seq_Assign^1 and
  ChangeVar (v, $expression_2$) $\in$ Seq_Assign^2 implies
  ChangeVar (v, IF (condition, $expression_1$, v)) $\in$ Seq_Assign^3.
  ChangeVar (v, $expression_1$) $\in$ Seq_Assign^1 and
  ChangeVar (v, $expression_2$) $\in$ Seq_Assign^2 implies
  ChangeVar (v, IF (condition, v, $expression_2$)) $\in$ Seq_Assign^3.
- if a signal s is assigned only in one branch, it is memorizing in the other branch where this is marked by an empty assignment NextSig(s). Marking the place where the signal is memorizing helps to combine correctly the sequential statements, without losing information.
  NextSig (s, $expression_1$) $\in$ Seq_Assign^1 and
  NextSig (s, $expression_2$) $\in$ Seq_Assign^2 implies
  NextSig (s, IF (condition, $expression_1$, NextSig (s))) $\in$ Seq_Assign^3.
  NextSig (s, $expression_1$) $\in$ Seq_Assign^1 and
  NextSig (s, $expression_2$) $\in$ Seq_Assign^2 implies
  NextSig (s, IF (condition, NextSig (s), $expression_2$)) $\in$ Seq_Assign^3.

Group takes two lists of assignments and for each assigned object it computes a new assignment.

Group : Seq_Assignments Seq_Assignments $\longrightarrow$ Seq_Assignments
Group(Seq_Assign^1, Seq_Assign^2) $\stackrel{\triangle}{=}$ Seq_Assign^3, as follows:

- if a variable v is assigned only in Seq_Assign^1:
  ChangeVar (v, $expression_1$) $\in$ Seq_Assign^1 and
  ChangeVar (v, $expression_2$) $\in$ Seq_Assign^2,
  then ChangeVar (v, $expression_1$) $\in$ Seq_Assign^3.
- if a signal s is assigned only in Seq_Assign^1:
  NextSig (s, $expression_1$) $\in$ Seq_Assign^1 and
  NextSig (s, $expression_2$) $\in$ Seq_Assign^2,
  then NextSig (s, $expression_1$) $\in$ Seq_Assign^3.
- if a variable v is assign in Seq_Assign^2 - ChangeVar (v, expression) $\in$ Seq_Assign^2, then ChangeVar (v, new_expression) $\in$ Seq_Assign^2, where new_expression = $expression_{[v'/expression']}$, for all variable v; vóccurs in expression and ChangeVar (v; expression) $\in$ Seq_Assign^1.
- if a signal s is assigned in Seq_Assign^2 - NextSig (s, expression) $\in$ Seq_Assign^2, then NextSig (s, new_expression) $\in$ Seq_Assign^3, where new_expression = $expression_{[v'/expression', NextSig(s)/expression'']}$, for all variable v; vócurrs in expression and ChangeVar (v; expression) $\in$ Seq_Assign^1, and NextSig (s, expression) Seq_Assign^1. $expression_{[v'/expression']}$ is expression where all occurrences of vwere replaced by expression;

Smart Card Research Perspectives*

Jean-Jacques Vandewalle

Gemplus Systems Research Labs
La Vigie, Z.I. Athélia IV, 13705 La Ciotat cedex, France
jean-jacques.vandewalle@research.gemplus.com

Abstract. This short paper introduces the issues and challenges of next generation Java-based smart card platforms. Betting on a continuous evolution toward open computing devices, next generation cards will consist in embedded Java micro-server platforms. Those platforms will be able to serve various types of services and applications thanks to two important system features: adaptability and maintainability. Two features that have to be carefully taken into account in the research perspectives described in this paper: real Java for cards, cards integration in a networked world, and flexible and adaptable cards.

1 Introduction

The research perspectives described in this paper leverage the vision of generic, adaptable, and maintainable smart cards that will meet the requirements of both the adjustment of smart card systems to the environment and the persistence of smart card applications in an evolving environment. The technology used to implement these smart card systems blur classical boundaries such as those between distributed and embedded systems, those between low- and high-end card platforms, or those between pre- and post-issuance (the term "post-issuance" describe the ability for smart cards to host and to run applications after they have been issued [3]). Thus, the goal of research perspectives is to bring some consistency to the possible solutions to these issues.

The emergence of powerful and personal services supported by multi-purpose devices has the potential to dramatically enrich the range of applications that smart cards will be able to serve. With the right platform and infrastructure in place, these smart cards will radically leverage card businesses as they enable card makers to offer to their customers solutions to implement and deliver complex and flexible business services. In consequence, the second goal of these research perspective is to participate in the understanding of this new possibilities by explaining and describing the research topics and their challenges.

This paper is organized as follows. Research perspectives are presented within the following three categories:

* This paper is an author's reviewed version of a previous paper. This revision is for the only purpose of accompanying the author's invited talk to the CASSIS 2004 conference. The previous paper by Jean-Jacques Vandewalle and Gilles Grimaud has been published (but couldn't be presented) in the proceedings of the Smart Objects Conference (SOC'2003), Grenoble, 15-17 May 2003.

G. Barthe et al. (Eds.): CASSIS 2004, LNCS 3362, pp. 250–256, 2005.

- "Real Java for smart cards"; this category reminds the central place of Java in smart cards, the Java limitations of the current Java Card 2.x specifications, and the will to provide a new ground-breaking release for next generation Java-based smart card;
- "Cards integration in a networked world"; extends the notion of interoperability from the current point of view of portability of Java Card programs to any Java Card platforms to integration of Java Card programs with other programs residing outside of the card;
- "Flexible and adaptable cards"; refines the use of Java to support multi-application smart cards, to the use of Java to support well-tailored customization of the card system depending on the target needs.

2 Real Java for Smart Cards

2.1 State-of-the-Art

Java Card [1] is now seen as the dominating platform for high-end microprocessor-based smart cards. And according to market analysts [2] this trend should continue for at least the next four years. Smart card vendors are focusing their strategy on this technology and an important part of their research and development resources is working on Java Card. There is no important development of a proprietary multi-application operating system in none of the major smart card companies.

Despite Java Card lives up to its promise, Java Card is not without challenges. Looking at standard Java, we see that the main Java strengths are well designed object-oriented language, cross-platform compatibility, virtual machine support for application sandboxing, garbage collection to guard against memory leaks, tight integration with other Java systems (back-ends, servers, three-tier architecture, etc.), and its industry and tools support. Java Card takes benefits from some of these strengths but it misses lots of them because of its stripped-down version to adapt to small devices. This stripped-down version has resulted in choices introducing issues that are difficult to solve without a re-foundation of the specifications.

2.2 Issues

These issues with the Java platform as it is defined by the current Java Card specifications are three-fold:

1. The very stripped-down version of Java provided by the Java Card 2.x specifications is targeted to very low-end chips. Therefore, drastic choices have been made that yield to very poor specifications in term of functionality compared to "standard Java". For instance, these drastic choices are optional integer numbers, no multi-threading, no (or optional) garbage-collection, specific file format different from the class file format that prevents on-board linking and reflection, a persistent memory model, etc.
2. Due to the specialization of the Java Card specifications to the smart card specific device and its constraints (for instance, see the APDU class specification), the Java Card platform is hard to understand, to handle, and to master by traditional Java developers that do not have any competencies in smart card technologies. Moreover,

this specialization does not permit to take benefits from the plethora of standard Java tools and has not encouraged the creation of interoperable, productive and easy-to-use tools to help smart card developers build applications.
3. Finally, because the Java Card specifications only consider the execution platform (virtual machine and runtime environment) plus the standard APIs, they miss some aspects of the smart card life cycle such as its initialization, its personalization, the way applications are installed during the pre-issuance stage, or during the post-issuance stage.

2.3 Challenges

In order to overcome the limitations and difficulties raised by the above mentioned issues (mainly, the ones in the first two categories), the Java Card specifications will surely evolve toward a ground-breaking release with the following principles: to target high-end (and possibly at the same time low-end) 32-bit chip platforms and take benefits of their new hardware capabilities, to get closer to mainstream Java functionality, to ease development and deployment processes, and to bypass the bottleneck of the specific smart card ISO 7816 communication protocols. Research perspectives should clearly enclose these principles. But it is also important that research perspectives don't only adopt an on-card standpoint, but also adopt a system standpoint in which a smart card is seen as part of global systems, as well as a flexible system by itself. These two last aspects complement the vision of a radically new and richer Java-based platform for smart cards. They are discussed in the two following sections.

3 Cards Integration in a Networked World

3.1 State-of-the-Art

One of the most promising feature for paying the price of Java in smart cards was the cross-platform compatibility brought by the use of Java and by the Java Card specifications. Looking backward, we now know that it wasn't painless. For instance, mobile networks operators using a Java Card from one card manufacturer had to virtually re-develop the applications to run on different vendor's Java cards. Card vendors have made substantial progress during the past year to make the cards interoperable. In GSM business, SIM card vendors are stepping up their efforts to bring about true interoperability through the SIMAlliance they have formed. Outside of the GSM world, Visa and the U.S. government are concentrating their efforts to ensure that the applications they wish to run on smart cards work on cards from multiple vendors. Thus, cross-platform compatibility can still be seen as an important challenge the card industry has to cope with.

The need for a true interoperability of the card platform within information systems is one of the feature for which it is worth paying the price of Java in smart cards. This kind of interoperability can be better described as the set of technologies for card integration in information systems. So far, some limited efforts have been done on this topic. They are limited to generic, multi-purpose sets of APIs for supporting interoperability of smart cards and card readers within the Windows operating system series (PC/SC), within the Java environments (OCF), or within the Linux world (Muscle).

This sets of APIs suffers from incompatibility and limited functionality because they are stuck with the card platform capabilities and because they only abstract the communication link to smart card applications to the level of messages exchanged with client applications running on the terminal where the card is plugged in. Messages have to be constructed by hand according to the card application specification, multiple and multiplexed communication channels are difficult (even impossible) to manage concurrently, reverse communication from the card applications and distant accesses from the network are not supported.

Efforts to overcome some of these bottlenecks are appearing with the 2.2 release of Java Card [7] in which the Remote Method Invocation (RMI) technology enables better integrated distributed operations from Java client applications, and opens up the door for distant accesses as it has been shown using the Jini technology [4]. Also, the Java Card 2.2 release offers multiple logical channels (at most 4), which enables multiple client applications communicating concurrently with multiple card applications. Nevertheless, all of these initiatives are still quite limited technologies to achieve an efficient integration of card devices and operations within networked applications.

3.2 Issues

Networked applications is about connecting information systems and exchanging information. They provide a great opportunity to make information more convenient to use and to be pieced together in order to achieve a common goal such as a complex service requiring, for example, distributed data and distributed processing. Fueling personalized services with power and flexibility is the key issue for getting new products to market more quickly and with the ability to make these products evolving according to the customer's needs. Notable examples of this evolution are the numerous attempts to deploy m/e-commerce applications carried out by many companies. These applications mix two visions that are apparently conflicting:

1. the assembly of complex and context-rich data (catalogues, orders, locations, contracts, etc.) with the cooperation of various information system sources (manufacturers, retailers, brokers, bankers, etc.), and;
2. the users' needs, because users expect services that are personalized, ubiquitous, non-intrusive regarding their privacy, secure, and fitted to their ever changing habits.

To summarize, information must be at the same time gathered for the client goals and available from different places with strong constraints of trust and convenience.

Though smart card platforms have been recognized as a key technology to leverage the development of such applications they suffer from many handicaps–e.g., their reduced internal hardware and software features, their very specific hardware and software interfaces, and their low-level development tools-among them their lack of interoperability with information system is a crucial one because it prevents easy ways of building end-to-end solutions incorporating smart card operations.

3.3 Challenges

One of the research perspectives is about providing application developers with Java Card supported technologies that will ease the card integration, and that will support the end-to-end argument [6]. These technologies range into the following categories:

- a standard communication protocol stack able to interconnect with diversity of networks, allowing distant accesses, and supporting multiple bi-directional communication exchanges at the same time;
- the support of multiple (non predefined and non frozen) application models to be able to interconnect with any current or future information system, such as information systems based on message-oriented middleware, mobile code infrastructure, or object-oriented invocation broker;
- the use of "standard" Java components to take benefits from the existing Java tools and from the widespread use of Java components (file format, idioms, APIs, etc.) in the information systems surrounding smart cards.

4 Flexible and Adaptable Cards

4.1 State-of-the-Art and Issues

Java Card has been marketed as the ideal platform for multi-application smart cards. Technically speaking, it is not erroneous. The flexibility to offer new services and to update data without physically swapping out the chip card is supported by Java Card. But there is a missing gap in Java Card security and management characteristics because the Java Card specifications do not define how an application is uploaded or removed from a card, nor who may add or delete applications from a card. Visa played a central role in promoting a system called Open Platform that specifies how these functions ought to be performed. In late 1999, Visa turned Open Platform to an industry consortium, GlobalPlatform, hoping to create a standard card manager specification that would be used by many industries.

The features provided by the GlobalPlatform application-management piece are dedicated and specific to the current state-of-the-art Java Card platforms. Moreover, they only bring flexibility for the management of the card applications and not for the platform itself.

One issue is to broaden this flexibility to the platform itself in order to make the next generation Java-based card platform adaptable to different needs according to the content and service providers requirements. This broaden flexibility and adaptation capabilities can be implemented in two ways: either statically for building the well-tailored Java-based card platform configuration before issuance, or dynamically in order to adapt on the fly the Java-based card platform to new application management rules or to different application models from those installed originally.

4.2 Challenges

Finally, another research perspective is about providing application issuers with Java Card supported technologies that will ease the card flexibility and adaptation in order to allow the platform to incorporate at best the only needed features for the context in which they operate and the applications they serve. These technologies range into the following categories:

- a dedicated and powerful dynamic linking model (DLM) adapted to smart cards. Today the smart card linking model is mostly off-card and static though DLM provides one of the main benefit of Java. DLM enables just-in-time code loading and linking, and thus implies minimal space consumption that is crucial for limited devices. But smart cards as well as others embedded devices; use static models (e.g., for classes preloaded in ROM). The goal is to extend the Java DLM to allow coexistence between dynamic and static linking models;
- an extended application-management component able to dynamically manage the applications and the platform extensions (like application model required libraries and services). To that extent, a key feature of the Java platform is the dynamic code loading supported by the Class Loader component. This component does not exist in Java Card because the platform uses exclusively a static linking model. Without a Class Loader component, code management (and thus application management) [5] is difficult to extend and to adapt in the context of an virtual machine that acts as an application server and that supports code mobility;
- the introspective nature of the Java-based card platform itself, which enables to apply instrumentation and architecture/assembly/factory tools for statically composing and building a dedicated configuration of the platform that reflects the state suitable for its usage;
- complementary to the previous point, the design of some pieces of the Java-based card platform as pluggable components that could be chosen among available instances (for instance, data-link layers for the communication stacks, applications models-required libraries) in order to configure the platform to its target usage;
- the use of "standard" Java mechanisms and components (for instance, serialization or standard Java service management technologies) to implement the above-mentioned technologies while taking benefits from the existing mechanisms, and leveraging on their associated tools.

5 Conclusions

We have defined next generation Java-based smart card as a platform enabling the efficient development, deployment and management of the on-card parts of networked applications. For that purpose, the platform has to provide a powerful and efficient execution environment as well as a flexible and operable management context within constrained environments. Such a platform also has to offer an harmonized interface to operate the services in a distributed and dynamic fashion required by m/e-commerce and m/e-services operations. For that purpose, the platform has to provide a standard way of communication within different types of networks as well as a rich and evolving framework for different application models.

Dealing only with the platform is not sufficient. Other technologies also have to be developed and should be added to these research perspectives. Some of them are quite independent of the platform capabilities. But some are tightly linked to the platform and then would benefit from being intimately developed in synergy with the platform. Most evident ones are: secure Java technologies, delegation of operations to card, and card management. Their efficiency and reliability have to be precisely measured whether or

not they are developed in convergence with the card platform, or with the possibility to influence some features of the card platform.

References

1. Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison Wesley, 2000.
2. D. Davis and D. Balaban. Wake up and smell the java! *Card Technology Magazine*, February 2002.
3. D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart card operating systems: past, present and future. In *Proceedings of the 5th NORDU/USENIX Conference*, February 2003.
4. P. George. A java technology-based, end-to-end solution for corporate products for the java card platform. Talk given at the 2001 Java One Conference, San Fransisco, June 2001. http://servlet.java.sun.com/javaone/conf/sessions/2013/google-sf2001.jsp.
5. S. D. Halloway. *Component Development for the Java Platform*. Addison Wesley, December 2001.
6. J. H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.
7. Sun Microsystems, Inc. *Java Card 2.2 Platform Specifications*, February 2002. http://java.sun.com/products/javacard/javacard22.html.

Author Index