Toward Automatic Generation of Provably Correct Java Card Applets

Alessandro Coglio

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
Ph. +1-650-493-6871 Fax +1-650-424-1807
http://www.kestrel.edu/~coglio
coglio@kestrel.edu

Abstract. This paper overviews an ongoing project aimed at developing an automatic generator of Java Card applets from higher-level spec(ification)s written in a domain-specific language called "SmartSlang". The generator is based on Specware, a system for the formal specification and refinement of software. The applet generator translates a SmartSlang spec into the logical language of Specware, re-expresses the translated spec in terms of Java Card concepts via a series of refinement steps using Specware's machinery, and generates Java Card code from the refined spec. The Java Card concepts used for refinement and code generation are captured as a shallow embedding of the Java Card language and API in the logic of Specware. Since proofs are associated to refinement steps, the applet generator produces a machine-processable proof tree along with the code, enabling the correctness of the generated code (with respect to the spec) to be checked independently from the applet generator, via a smaller and simpler applet checker to be also developed in this project.

1 Introduction

A smart card [10] consists of a chip embedded in a plastic substrate that is the size of a credit card or smaller. Since the chip is very small, it has relatively limited storage and processing capabilities. A smart card communicates with the external world via metal contacts on the surface of the card or via an antenna wound into the card. The signals exchanged through the contacts or the antenna encode commands to and responses from the card. The card's memory and functionality can be accessed exclusively via these commands and responses. Smart cards are easily portable, cost-effective computing devices that can securely store and process information. Applications include authentication, banking, and telephony. Since bugs in a smart card's hardware or software may compromise its security, correctness is of paramount importance in smart cards.

Java Card [4] is a version of Java [2] for smart cards. A Java Card applet is written in a subset of Java and uses a different API from regular Java. The Java Card API provides functionality to receive commands and send responses according to certain standardized protocols, perform cryptographic computations,

etc. Java Card provides a uniform platform to develop portable smart card applications that can be also installed and updated after the card is issued to the user. In addition, the type safety of Java is a good foundation for security.

Even though Java is relatively high-level, developing Java Card applets requires the programmer to deal with fairly low-level details, thus greatly increasing the potential for bugs. For example, commands and responses must be explicitly decoded from and encoded into bytes. Furthermore, because of the severe memory limitations and the typical absence of garbage collection from Java Card implementations, often the programmer has to forgo object-oriented principles and write code that makes almost no use of inheritance, keeps nested method calls to a bare minimum, and allocates all the needed objects when the applet is installed, re-using them during the applet's normal operation. While smart card memory keeps increasing thanks to the continual advancements in semiconductor technology, the market demand for smart card functionality keeps increasing too. So, memory limitations are likely to be a problem at least for a while.

This paper overviews an ongoing project aimed at developing an automatic generator of space-efficient and time-efficient Java Card applets from higher-level spec(ification)s. The generator is being built on top of Specware [9], a system for the formal specification and refinement of software, whose mathematical foundations guarantee that the output applet code is provably correct with respect to the input applet spec. It is expected that the use of the generator will greatly increase the productivity of applet developers and the confidence that applets are correct.

2 Applet Generator

From the user's point of view, the applet generator is a box that takes an applet spec written in a domain-specific language called "SmartSlang" ("Smart Card Specification Language") and automatically produces Java Card code that implements the specified applet. Internally, there are three components that operate in sequence, as shown in Figure 1.

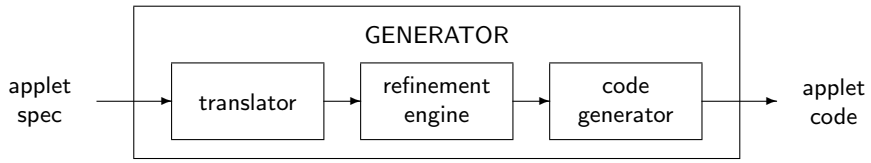

Fig. 1. Structure of the generator

2.1 SmartSlang and Translator

The design of (a first version of) SmartSlang is ongoing and not finalized yet. So, only some highlights and goals are discussed here.

The domain of the domain-specific language SmartSlang is the one of smart cards. This means that SmartSlang features high-level constructs for smart card concepts, such that an applet's functionality can be specified in a very clear and concise way, minimizing the chance of specification errors. SmartSlang includes explicit constructs for commands, responses, encoding of commands and responses as bytes forming Application Protocol Data Units (APDUs), error handling, cryptography (keys and encryption/decryption operations), personal identification numbers (PINs), challenge-response protocols, and so on. For example, commands are identified symbolically and are accompanied by declarations of their encoding as APDUs, as opposed to Java Card where APDU bytes are explicitly retrieved and decoded, dispatching to code that processes the various commands. As another example, cryptographic operations are readily accessible as pure functions operating on data and keys, as opposed to Java Card where a `Cipher` object is created, initialized with a key, and used to encrypt/decrypt data, possibly in chunks. Of course, the applet generator turns these higher-level constructs into Java Card code that implements their semantics.

SmartSlang features a richer type system than Java Card. For instance, instead of only the few Java Card integral types, arbitrary integer ranges can be used as types, to describe more precisely the semantics of certain values (e.g. a balance in an e-wallet cannot be negative). Conformance of values to integer range types, as well as all the other type safety properties of an applet spec (e.g. array indices within bounds), are meant to be statically checked by the applet generator. This requires a fancier type checker than found in typical languages. Since loops are generally problematic for this kind of analyses, the intention is to use restricted, tractable forms of loops (e.g. loop through an array) and/or to have the user add suitable annotations (e.g. invariants) to guide the type checker. The challenge is to develop a powerful but tractable type system that does not require general symbolic theorem proving (only lightweight decision procedures, e.g. for Presburger arithmetic) and does not force the user to write too many annotations. The payoff of full static checking is that more specification errors are caught (e.g. forgetting to check a balance in a debit command) and that the generated Java Card code will not throw any exception at run time.

SmartSlang has a mathematical semantics in terms of state machines: an applet is a state machine that updates its state and produces a response when it receives a command. Anyhow, SmartSlang should be simple enough to be effectively understood and used by smart card experts without a formal background.

We have used some tentative SmartSlang constructs to specify a few realistic applets, e.g. an applet that signs data with an internally stored private key after a successful PIN verification and/or challenge-response external authentication. In these preliminary experiments, the tentative SmartSlang specs are considerably shorter and clearer than their corresponding Java Card code.

The first component of the applet generator is a translator from SmartSlang to MetaSlang, the specification language of Specware. MetaSlang is a version of higher-order logic [1], similar to the languages of PVS [11], HOL [13], and other systems. A MetaSlang spec consists of some sorts (i.e. types), some op(eration)s (i.e. functions over the types), and some axioms that constrain the sorts and the ops, i.e. a spec is a theory in higher-order logic. The translation of a SmartSlang spec is a MetaSlang spec of the state machine denoted by the SmartSlang spec. In fact, this translation can be considered to formally define the semantics of SmartSlang. The translator also performs the needed type safety checks on the SmartSlang spec. The translator will be developed after the design of SmartSlang is finalized.

2.2 Refinement Engine

No substantial work has been done on this component yet. So, only high-level concepts are presented here.

The idea is that the high-level MetaSlang spec of the applet, obtained by translating the SmartSlang spec, is transformed into a low-level MetaSlang spec from which Java Card code is readily generated. The transformation consists of a series of refinement steps that are applied by means of Specware's underlying refinement machinery.

Specware provides a simple and powerful notion of refinement as a morphism between specs. A morphism is a mapping from the source spec's sorts and ops to the target spec's sorts and ops, such that all the axioms in the source spec are theorems in the target spec, once translated according to the sort/op mapping. Often a morphism maps the sorts and ops of the source spec to the same sorts and ops in the target spec, especially when the two specs describe the same entity at different levels of abstraction.

Starting from the high-level applet spec, each refinement step carried out by the refinement engine re-expresses the spec in lower-level terms, making use of Java Card concepts. For example, a state component consisting of a balance having some SmartSlang integer range type is re-expressed as having an appropriate Java Card type where the balance can fit without wasting space; e.g. for a balance between 0 and 100 a byte is sufficient, but for a balance between 0 and 1,000,000 two shorts (or just a short and a byte, if space is scarce) are needed (`int` can be used if supported by the target Java Card implementation). As another example, the application of a SmartSlang cryptographic operation is re-expressed as a series of actions where (a representation of) a `Cipher` object is created, initialized with some key, and used to encrypt/decrypt data, possibly in chunks.

2.3 Code Generator

A first version of this component has been completely designed and implemented. So, more details than for the other two components are given here.

At the end of the refinement process, the applet's functionality is entirely expressed in terms of Java Card concepts. These Java Card concepts are captured in the form of parameterized (i.e. generic) MetaSlang specs that are suitably instantiated (i.e. particularized) for the specific applet. The code generator generates Java Card code from the instantiated specs.

The parameterized MetaSlang specs are a shallow embedding of (a significant subset of) the Java Card language and API in higher-order logic. The adjective "shallow" means that Java Card expressions and statements are identified with their semantics, i.e. their syntax is not explicitly formalized but is implicitly captured by MetaSlang terms. In contrast, in a "deep" embedding Java Card expressions and statements would be defined as syntactic entities to which semantics is associated. The choice of a shallow embedding is solely motivated by it being slightly simpler and smaller than a deep embedding; a deep embedding would work as well.

Figure 2 shows some excerpts of the formalized semantics of Java Card expressions and statements. Following the Java language specification [6], an expression may complete abruptly with an exception or normally returning a result that is a value, a variable (e.g. for assignment), or nothing (i.e. a call to a `void` method); a statement may complete normally or abruptly with an exception or returning a value (e.g. `return 2;`) or nothing (i.e. `return;`); other forms of statement abrupt completion are currently excluded from the formalization. In general, executing an expression or statement causes a side effect consisting of a state change and finite sequences of input/output events. An expression or statement is defined as the set of all its possible executions, where an execution consists of a side effect and a completion (note that a value of sort `S -> Boolean` is isomorphic to a set of values of sort `S`). Expressions include calls to API methods, whose semantics is defined not in terms of Java Card constructs but directly, as if they were "macro-expressions" in the language.

Input and output events are a mildly original aspect of this formalization, compared to others. Certain API method calls cause APDU bytes to be transferred between the card and the external world. These APDU chunks are captured as input/output events. Having them explicit allows to express the input/output behavior of (the MetaSlang representation of) an applet's code in terms of command and response APDUs.

Since the semantics is relational (i.e. sets of side effects and completions), expressions and statements may be non-deterministic. Non-determinism is used to model the creation of fresh references to new objects (the axioms only say that the fresh reference must not be already present in the heap) and to model API method calls for random number generation. Because of these two features, non-determinism ripples to all the expressions and statements of the formalization. Otherwise, expressions and statements could be defined as functions from old states and input events to new states, output events, and completions. An alternative approach is to capture fresh references and generated random numbers as input events, thus keeping the semantics functional.

```
sort ExpressionResult = | var Variable
                        | val Value
                        | nothing
sort ExpressionCompletion = | normal ExpressionResult
                            | abrupt Exception
sort StatementAbruptCompletion = | retnothing
                                 | ret Value
                                 | exc Exception
sort StatementCompletion = | normal
                           | abrupt StatementAbruptCompletion
sort SideEffect = {old : State,
                   new : State,
                   in  : FSeq InputEvent,
                   out : FSeq OutputEvent}
sort Expression = SideEffect * ExpressionCompletion -> Boolean
sort Statement = SideEffect * StatementCompletion -> Boolean
```

Fig. 2. Excerpts of Java Card expression and statement semantics

Java Card constructs are formalized by means of ops that combine subexpressions and substatements into compound expressions and statements. For example, addition is captured by an op of sort `Expression * Expression -> Expression`: the resulting expression combines the side effects of the subexpressions and yields the sum of their results as result, unless either subexpression throws an exception, in which case the compound expression completes abruptly with the same exception; for lack of space, we do not show the MetaSlang axioms here.

The parameters of the parameterized MetaSlang specs of Java Card consist of sorts and ops that are suitably instantiated to represent specific programs. For instance, one of the parameters is an uninterpreted sort `UserClass`, accompanied by an axiom stating its finiteness, whose values are meant to be exactly the user-defined classes comprising the program (API classes, which are fixed for every program, are captured by an interpreted sort `APIClass`). For instance, to represent a program with three user-defined classes `C`, `D`, and `E`, the sort is defined as consisting of three values:

```
sort UserClass = | class_C | class_D | class_E
```

Similarly, there are uninterpreted sorts for user-defined fields, methods, etc., as well as uninterpreted ops that associate fields and methods to the classes where they are declared, bodies to non-abstract methods, and so on. All these sorts and ops are accompanied by axioms that constrain their possible instantiations so that the instantiated specs represent legal programs (e.g. no method is both abstract and static).

In Specware, a parameterized spec is instantiated by means of a morphism from the subspec consisting of the parameter sorts and ops along with their constraining axioms, to a spec that instantiates those sorts and ops. The validity

of the morphism ensures that the parameters are instantiated consistently with their constraints, because all the axioms in the parameter subspec are theorems in the instantiating spec. The result of instantiation is the spec obtained by replacing the parameter subspec in the parameterized spec with the instantiating spec.

The refinement engine ultimately produces an instantiated spec that represents a specific Java Card program. The program is a Java Card applet that behaves as specified by the high-level MetaSlang spec obtained by translating the SmartSlang spec. The code generator examines the instantiated spec and straightforwardly produces Java Card source code for the program. For example, from the definition of the sort `UserClass`, the code generator determines which classes are declared in the program. Fields, methods, etc. are similarly determined from the definitions of the associated sorts and ops.

Code generation is reversible: from the Java Card program it is possible to reconstruct the instantiated spec from which the program was produced. For instance, the definition of the sort `UserClass` can be reconstructed by examining the classes declared in the program. The relevance of this reversibility is explained in the next section.

3 Independent Certification

Each refinement step carried out by the refinement engine of the applet generator has an associated proof that all the axioms in the source spec of the morphism are theorems in the target spec. The sequential composition of the morphisms corresponding to all the refinement steps yields a morphism from the high-level spec obtained by translating the SmartSlang spec to the low-level spec from which Java Card code is generated. The proof for the compound morphism is obtained by composing the proofs for the component morphisms.

A future goal of this project is to have the applet generator produce, besides the code, a machine-processable proof tree of the overall refinement, as shown in Figure 3. This enables to certify the correctness of the code with respect to the spec independently from the generator, by means of an automatic checker to be also developed in this project. The applet checker, also shown in Figure 3, takes as input the SmartSlang spec, the Java Card code, and the proof. The SmartSlang spec is translated into a high-level MetaSlang spec, using the same translator used in the applet generator. The Java Card code is translated into a low-level MetaSlang spec using the code abstractor, which is the inverse of the code generator used in the applet generator. Given the high-level and the low-level MetaSlang specs, a simple proof checker for the higher-order logic of MetaSlang is used to establish whether the purported proof is a valid proof that all the axioms in the high-level spec are theorems in the low-level spec.

When a Java Card applet is produced by the applet generator, the input spec and the generated proof constitute a checkable certificate that accompanies the generated code. By using the applet checker to establish the validity of the certificate, trust is essentially shifted from the relatively large and complex

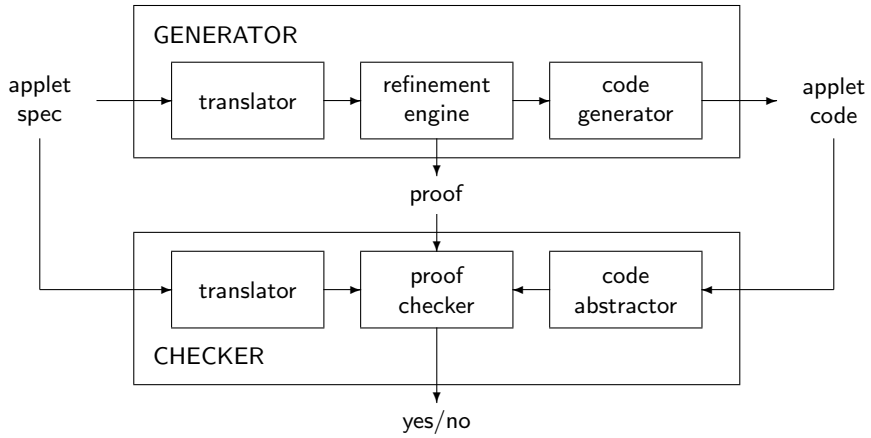

Fig. 3. Independent certification

refinement engine to the much smaller and simpler proof checker. The size and complexity of the code abstractor are about the same as the code generator, relatively modest; the size and complexity of the translator are also relatively modest. The translator and code abstractor are necessary to bring the applet spec and the applet code, which are written in different languages (i.e. SmartSlang and Java Card), into a common logical language (i.e. MetaSlang) where their relationship is formally expressed and proved.

As mentioned above, independent certification is a future goal: no work has been done on generating refinement proofs or on the applet checker yet.

4 Implementation Language

The applet generator and checker are being written in MetaSlang. This is a natural choice because Specware is itself written in MetaSlang and because the applet generator and checker are built on top of Specware (e.g. they manipulate MetaSlang specs).

More in detail, the applet generator and checker are being written in an executable subset of MetaSlang, which is like a purely functional programming language. Specware is itself written in executable MetaSlang. Non-purely functional computation such as accessing files is encapsulated in small hand-written code fragments and/or in monads. Specware can turn executable MetaSlang into LISP, C, or Java. However, C and Java code generation are not as developed as LISP code generation yet. For this reason, the code generator component of the applet generator, as well as Specware itself, are currently generated in LISP.

Eventually, we plan to write high-level MetaSlang specs for the applet generator and checker and to obtain the executable ones via refinement. This will provide higher confidence in the correctness of the two tools.

5 Related Work

Hubbers, Oostdijk, and Poll [8, 7] have built a tool that automatically translates finite state machines into Java Card skeleton code and associated Java Modeling Language (JML) annotations [5]. The generated JML annotations express pre- and post-conditions, invariants, etc., which can be checked to hold for the generated code by means of existing tools, somewhat analogously to the certificate produced by our applet generator. A difference from our approach is that finite state machines, unlike SmartSlang, have no high-level constructs specific to smart cards. Another difference is that the generated skeleton code only implements (and accordingly the JML annotations only express properties of) part of the applet's functionality, essentially its control structure; the rest of the needed functionality must be added by hand. In contrast, our applet generator is meant to generate the complete code. However, those authors plan to extend their approach to generate more of the applet functionality.

Besides the work just mentioned, the author is unaware of other research projects aimed at automatically generating Java Card applets from higher-level specs.

There is extensive work on formally specifying the Java Card platform (language, virtual machine, and API) to precisely document and validate the platform itself and to verify properties of Java Card applets. That work relates to the formalization of the Java Card language and API used in the applet generator, but our formalization has a different purpose, namely to capture Java Card concepts for refinement and code generation.

Our formalization is relatively close to, and partially inspired by, the one developed within the LOOP project [3]. In that project, properties of Java Card applets are verified by translating the code into higher-order logic (like the code abstractor in the applet checker) and then using theorem provers like PVS and Isabelle [12] to prove conjectures in the resulting theories. The conjectures are expressed as JML annotations, which are translated into higher-order logic along with the code. The semantics of the Java Card language and API and of the JML annotations is formalized via a shallow embedding in higher-order logic in a coalgebraic style. The differences between the LOOP formalization and ours are not very profound; they are motivated by subjective reasons (e.g. relational vs. coalgebraic style) or by different project needs (e.g. our explicit input/output events to express the input/output behavior of applets). Our formalization covers a smaller Java Card subset than the LOOP formalization, but our coverage will grow as this project proceeds.

References

1. Peter Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.
2. Ken Arnold, James Gosling, and David Holmes. *The Java™ Programming Language.* Addison-Wesley, third edition, 2000.

3. Bart Jacobs et al. The LOOP project. Information at http://www.cs.kun.nl/~bart/LOOP.
4. Zhiqun Chen. *Java Card™ Technology for Smart Cards*. Addison-Wesley, 2000.
5. Gary Leavens et al. The Java Modeling Language. Information at http://www.cs.iastate.edu/~leavens/JML.html.
6. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification*. Addison-Wesley, second edition, 2000.
7. Engelbert Hubbers and Martijn Oostdijk. Generating JML specifications from UML state diagrams. In *Proc. Forum on specification and Design Languages (FDL'03)*, September 2003.
8. Engelbert Hubbers, Martijn Oostdijk, and Erik Poll. From finite state machines to provably correct Java Card applets. In *Proc. IFIP SEC'03 Workshop on Small Systems Security (WG 11.2)*, May 2003.
9. Kestrel Institute and Kestrel Technology LLC. Specware™. Information at http://www.specware.org.
10. W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley and Sons, second edition, 2000.
11. SRI International. The PVS specification and verification system. Information at http://pvs.csl.sri.com.
12. Technical University of Munich and University of Cambridge. The Isabelle system. Information at http://isabelle.in.tum.de.
13. University of Cambridge. The HOL system. Information at http://www.cl.cam.ac.uk/Research/HVG/HOL.