Code Generation for High-Assurance 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

1 Introduction

1.1 Smart Cards

A *smart card* [1], also known as *integrated circuit card* or *chip card*, consists of an electronic chip embedded in a plastic substrate. The size of the plastic substrate is the same as a credit card or smaller. So, the chip is very small and hence has relatively limited processing and memory capabilities, even though these capabilities keep increasing thanks to the continual advancements in semiconductor technology. 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. So, smart cards are easily portable, cost-effective computing devices that can *securely* store and process information.

For example, a smart card can store a private key and use it to sign or encrypt/decrypt data. Since the card only recognizes and processes certain commands (e.g. to sign data provided in the command, returning the signature as a response), there is no way to extract the value of the private key as a response from the card. The value of the private key could be returned only if such a functionality had been explicitly programmed into the card, which hopefully has not. Smart cards also have built-in defenses to make it difficult (i.e. not cost-effective) to break their security via special hardware equipment. Current applications of smart cards include authentication, financial transactions, telephony, and the list is growing.

While smart cards can provide secure storage and processing of information, such a goal is fulfilled only if the smart card's software and hardware are *correct* with respect to the desired security policies. Correctness is of paramount

importance for smart cards; it is the main criterion for determining the quality of a smart card product.

1.2 Java Card

Java Card [2] is a version of Java for smart cards. A Java Card program is written in a subset of Java (features such as floating point arithmetic and multi-dimensional arrays are not part of the subset) and makes use of different libraries than the usual Java libraries. Java Card libraries provide functionality to receive commands and send responses according to certain standardized protocols, to perform cryptographic computations, etc. In addition to these standard libraries, various industry-specific libraries are being built (e.g. for telephony and banking). An application in Java Card is called a Java Card *applet*. Aside from the name, Java Card applets have very little to do with the Java applets embedded in Web pages.

A Java Card program's source code is compiled via a regular Java compiler, resulting in a number of class files (one for each class or interface declared in the program), as usual. Next, a *converter* is run on these class files. The converter checks that the class files conform to the Java Card subset of Java, and in that case it produces a number of *CAP* (Converted APplet) files (one for each package constituting the program). The CAP files are optionally checked by a *verifier* for consistency and then are loaded into the *Java Card Runtime Environment* (*JCRE*), which is a virtual machine running on the smart card. The JCRE sits on top of the smart card operating system, which in turn sits on top of the hardware. The JCRE includes an interpreter for bytecodes as well as an implementation of the Java Card libraries (mostly in native code). The CAP file format is optimized for loading into and execution by the JCRE.

The term *Java Card Virtual Machine* (*JCVM*) is an overloaded one. It is used to denote the interpreter of the JCRE, but it is also used to denote the ensemble of converter, verifier, and JCRE. The Java Virtual Machine (JVM) reads class files and executes them. Analogously, the JCVM, according to its second meaning just described, reads class files and eventually executes them, with conversion and verification taking place in the process. But while the converter and the verifier run outside the card, the JCRE runs on the card. In fact, this architecture is referred to as *split* virtual machine architecture. Given the limited memory and processing capabilities of smart cards, currently it is problematic to put converter and verifier on the card; this is the reason for the split architecture.

Java Card brings the benefits of the Java technology to the world of smart cards. There exist several smart card platforms, and until a few years ago it was difficult to develop portable smart card applications or to update the card contents after the issuance of the card. Also, the type safety of Java provides a good foundation for security. So, Java Card provides a uniform and secure platform for the development of smart card applications.

2 High Assurance and Kestrel's Work

In order for a Java Card applet to be secure (according to the desired security policies), it must be correctly implemented. In addition, the platform on which the applet runs, i.e. the JCRE, must be correctly implemented. Since the JCRE sits on top of a smart card operating system and hardware, those must be also correctly implemented. Finally, all the tools used "between" the source code and the JCRE (i.e. compiler, verifier, and converter) must be correctly implemented.

In other words, the assurance of a Java Card applet depends on several, mutually dependent critical elements. High assurance is not a black-or-white property, but rather it is "measured" in shades of gray. Improving the correctness of any of the above critical elements significantly increases the overall assurance of the applet.

Kestrel has been developing tools and techniques for the development of high-assurance software. Our Specware tool [3] provides capabilities to write formal specs, refine them, compose specs and refinements, generate running code from refined specs, and call mechanical theorem provers to prove the correctness of specs and refinements as well as putative properties of specs for validation purposes. Our Planware tool [4] is a domain-specific extension of Specware: it is a fully automatic generator of schedulers from high-level specs written in a simple and intuitive tabular form. Planware internally translates these domain-specific specs into Specware specs and makes use of the Specware machinery to refine them to extremely efficient code. Other generators have been and are being built at Kestrel (e.g. a generator of C code from Stateflow [5] diagrams).

Currently, we have projects underway to apply and extend our synthesis technology to Java Card. More precisely, we are working on the following two tasks:

1. synthesize implementations of the JCRE and related tools;
2. build an automatic *generator* of Java Card applets.

For the first task, we are building a formal spec of the JCRE in Specware. This spec will be refined first to a simulator, then to a JCRE implementation that runs on some smart card. The formal spec includes the CAP file format (which we have almost completed), the interpretation of bytecodes, and the Java Card libraries. Our formal spec can serve as an unambiguous, precise version of the informal spec provided by Sun [6, 7, 8]. The simulator is a development tool that simulates the JCRE on a desktop computer, useful for testing and debugging purposes. Its correctness is important because the developer expects it to faithfully represent the behavior of an actual card. We also plan to synthesize a converter and an off-card verifier. We are approaching this long-term task via vertical slices.

We have been previously successful at the specification and synthesis of certain security-critical components of the JVM. Since Java security is based on type safety, we have studied the components of the JVM that are responsible for enforcing type safety. We have realized a complete mathematical spec of the

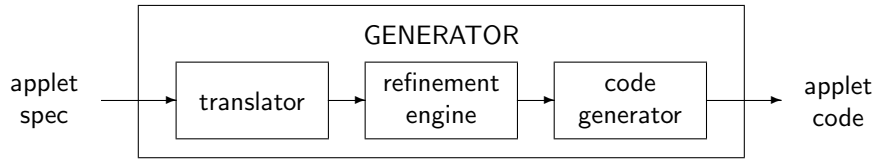

Figure 1: Architecture of the generator

Java bytecode verifier. We have written that spec in Specware and refined it to a running implementation (see [9] for a preliminary report). In this process, we have designed various improvements over Sun's spec and implementation of bytecode verification [10], in particular the treatment of subroutines [11] and subtype checks [12]. We have also identified some bugs in Sun's spec and implementation and we have proposed corrections [13]. Furthermore, we have developed a formal spec of the JVM class loading mechanisms along with their interplay with bytecode verification, and proved a type safety theorem about our formalization [12].

The second task, namely the construction of a Java Card applet generator, and in particular its code generation back end, is the main topic of this paper and is described in the sequel.

3 Applet Generator

Preliminary ideas for the architecture of the applet generator have been presented in [14]. Since then, the architecture has been slightly refined, without changing its essence. This section overviews the refined architecture.

From the user's point of view, the generator is a box that takes an applet spec as input and automatically produces Java Card code implementing the specified applet. The spec is written in a domain-specific language (DSL) that has a precise, mathematically defined semantics, while being at the same time easy and intuitive to use by smart card developers. The vocabulary of this DSL includes smart cards concepts such as commands, responses, keys, PINs, etc. The constructs of the DSL are higher-level than the Java Card code that achieves the same functionality. In other words, from the user's point of view the generator is a smart compiler that translates high-level specs written in a DSL into lower-level Java Card code.

Internally, the applet generator consists of three components that operate one after the other, as shown in Figure 1.

The first component is a *translator* that translates the applet spec written in the DSL into an applet spec written in MetaSlang, the language of Specware. MetaSlang is a version of higher-order logic, so it is mathematically precise, However, the user is not exposed to it, because Specware specs are only produced (and processed, as described below) internally. This translation formally defines the semantics of the DSL.

The second component is a *refinement engine* that makes use of the Specware machinery to apply refinements to the high-level applet spec. The result is a low-level applet spec from which Java Card code can be readily generated (see below). While this component operates on MetaSlang specs, it applies certain refinements that re-phrase (i.e. refine) the applet spec in terms of Java Card constructs and API,which are suitably represented in MetaSlang. Conceivably, some of the initial refinements applied by the refinement engine can be Java Card-independent. These refinements will be re-usable if the applet generator is extended to target other smart card platforms [15] besides Java Card.

The third component is the *code generator*, which translates the low-level Specware spec, resulting from the refinement process, into Java Card code. Details about the code generator are given in the following section.

4 Code Generator

We have devised a stylized format to represent (for now, a subset of) Java Card programs in MetaSlang. There is a bijective correspondence between Java Card programs and Specware specs in that stylized format: from a program we can generate a stylized spec and from a stylized spec we can generate a program. The refinement engine eventually produces a stylized spec, which the code generator transforms into the corresponding Java Card program.

The stylized specs are realized by means of an *embedding* of Java Card in MetaSlang, i.e. a Specware spec of the Java Card language and API. The embedding does not simply specify the syntax of Java Card, but rather its semantics. A stylized spec, produced by the refinement engine, is a refinement of a high-level applet spec that describes the semantics of the applet, i.e. all properties of the high-level spec hold in the stylized one; this implies that the stylized spec includes the semantics of the Java Card program it represents.

Two commonly used adjectives for language embeddings are *deep* and *shallow*. A deep embedding is one where the (abstract) syntax of the object language is explicitly represented in the meta language, along with semantics associated to the syntax. A shallow embedding is one where only the semantics of the object language is represented in the meta language, i.e. syntactic entities of the object language are identified with their semantics and implicitly represented as terms in the meta language.

For example, consider boolean expressions built out of boolean constants, variables, and operators; each expression evaluates to a boolean value, given values for its variables. A deep embedding consists of the (abstract) syntax of these expressions along with a function that associates to each expression its semantics. The semantics of an expression is a function that, given an environment mapping variables to values, produces the value that is the result of the expression. A shallow embedding specifies that an expression "is" its semantics, i.e. an expression is a function that, given an environment, produces a value. Each constant expression is a constant function, each variable expression returns the corresponding value from the environment, and each compound

expression (e.g. conjunction) is a higher-order function that operates on the subexpressions' functions.

Our embedding of Java Card in MetaSlang is a shallow one. The reason is that, since a shallow embedding only formalizes the semantics, it is smaller then a deep one. This choice might change in the future, e.g. if some other application required a spec of the abstract syntax of Java Card, then we could gain leverage by using the same deep embedding for that application and for the applet generator.

In our embedding, Java expressions and statements are formalized as relations over side effects and completions. Completions can be normal (e.g. an expression returns a value) or abrupt (e.g. an exception is thrown). A side effect consists of an old state, a new state, zero or more input events, and zero or more output events. The relational approach (as opposed to a functional one, i.e. functions from input events and old state to new state and output events) captures non-determinism, useful to model the Java Card API for random number generation. Input and output events capture interaction with the external world, namely the commands and responses that are received and sent via certain Java Card API methods.

The Java Card API is part of the embedding. Its methods are modeled as statements (i.e. relations over side effects and completions) that are defined not by combining statements of the language (e.g. `while`) but directly, as if they were additional, built-in, macro statements.

5 Independent Certification

An automatic generator greatly reduces human coding errors, especially when it is based on solid mathematical foundations such as Specware's. However, the question arises about the correctness of the generator itself. There may be bugs in the generator that affect the correctness of the generation process. A similar concern applies to verification, not only to synthesis. Even if a program has been verified via a theorem prover, model checker, or other tool, bugs in the verification tool may invalidate the correctness result of the program.

In verification, an approach to increase assurance is to have the verification tool produce an explicit proof (provided that the tool possesses this functionality) that can be checked with a simple proof checker. The proof checker is much simpler and smaller than the verification tool, and therefore more likely to be correct. After the proof has been checked, the correctness of the proof only depends on the correctness of the checker, regardless of the tool used to produce the proof.

The Specware refinement process can produce proofs of each refinement step. The proofs for the various refinement steps can be composed together to obtain a proof of the overall refinement process. Proofs for the individual steps are produced by the transformations that are applied to the specs; since these transformations are (designed to be) correctness-preserving, there is a proof associated to each application of a transformation. We plan to have the applet

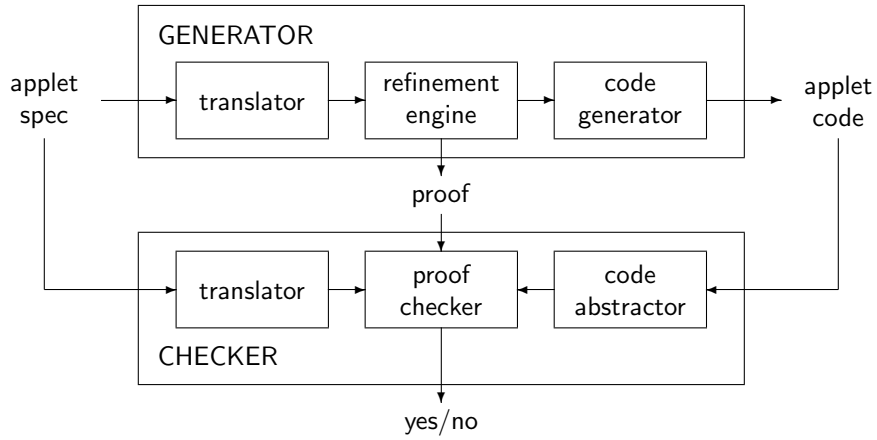

Figure 2: Independent certification

generator produce, besides the code, a proof that the code correctly implements the user's spec. The correctness of the code with respect to the spec can then be checked independently, as shown in Figure 2.

From the user's point of view, the applet *checker* is a box that takes as input the applet spec, the applet code, and a proof, and returns a yes/no answer, depending on whether the proof is valid evidence of the correctness of the code with respect to the spec.

Internally, the checker consists of three components. The translator is the same one used in the applet generator: it translates the applet spec, written in the DSL, into MetaSlang. The *code abstractor* is the inverse of the code generator used in the applet generator: it translates the Java Card program into a stylized Specware spec that represents it. The *proof checker* checks that the proof is valid evidence of the correctness of the low-level spec (produced by the code abstractor) with respect to the high-level spec (produced by the translator).

If the applet checker returns a positive answer, then the correctness of the applet code with respect to the applet spec only depends on the correctness of the translator, code abstractor, and proof checker. The code abstractor is comparable, in size and complexity, to the code generator. However, the proof checker is definitely smaller and simpler than the refinement engine, and thus easier to trust.

Translator and code abstractor are necessary to the applet checking process and cannot be eliminated. The reason is that the applet spec and code are written in two different languages; the only way to "formally compare" two entities written in different languages is to bring both into a common, mathematical language, in this case MetaSlang. Indeed, translator and code generator/abstractor merely serve to "move" across the three languages involved (DSL, Java Card,

and MetaSlang). Most of the interesting work in the applet generator is carried out by the refinement engine. The generation of the proof allows trust to be put into a much smaller and simpler component than the refinement engine.

As a general remark, the synthesis approach constructs the correctness proof during the synthesis process, by composing smaller proofs corresponding to the various refinement steps. In contrast, an after-the-fact verification approach must construct the whole proof from the code only. The search space is therefore larger. In other words, the synthesis approach can provide more scalability.

6 Current Status and Future Work

We have developed the shallow embedding of a subset of the Java Card language and API in MetaSlang. While the subset is large enough to capture non-trivial programs, we plan to extend the embedding to cover all Java Card features, eventually. We have also implemented a code generator for that subset. As the subset becomes larger, the code generator will be extended accordingly.

We are in the process of defining a first version of the DSL and related translator. This first version is sufficient to specify non-trivial applets, but future versions will allow the specification of more applets, including inter-communicating ones, extending the translator accordingly. A first version of the refinement engine, code abstractor, and proof checker will be also developed shortly.

From the user's point of view, the evolution, through successive versions, of the applet generator and checker will manifest as an evolution of the DSL and as an expansion of the set of applets that can be generated and checked. Internally, the various components of the generator and checker will be evolved in a modular way.

References

[1] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley and Sons, second edition, 2000.

[2] Zhiqun Chen. *Java Card™ Technology for Smart Cards*. Addison-Wesley, 2000.

[3] Kestrel Institute and Kestrel Technology LLC. Specware™. Information at http://www.specware.org.

[4] Lee Blaine, Li-Mei Gilham, Junbo Liu, Doug Smith, and Stephen Westfold. Planware: Domain-specific synthesis of high-performance schedulers. In *Proc. 13th Conference on Automated Software Engineering (ASE'98)*, pages 270–280, October 1998.

[5] The Math Works Inc. Stateflow user's guide, 2000.

[6] Sun Microsystems. Java Card™ 2.1.1 Runtime Environment (JCRE) Specification, May 2000. Available at http://java.sun.com/javacard.

[7] Sun Microsystems. Java Card™ 2.1.1 Virtual Machine Specification, May 2000. Available at http://java.sun.com/javacard.

[8] Sun Microsystems. Java Card™ 2.1.1 Application Programming Interface, May 2000. Available at http://java.sun.com/javacard.

[9] Alessandro Coglio, Allen Goldberg, and Zhenyu Qian. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.

[10] Alessandro Coglio. Improving the official specification of Java bytecode verification. *Concurrency and Computation: Practice and Experience*, 15(2):155–179, February 2003.

[11] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. In *Proc. 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002.

[12] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Proc. 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35, number 10 of *ACM SIGPLAN Notices*, pages 325–336, October 2000. Long version available at http://www.kestrel.edu/java.

[13] Alessandro Coglio and Allen Goldberg. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience*, 13(13):1153–1171, November 2001.

[14] Alessandro Coglio. An approach to the generation of high-assurance Java Card applets. In *Proc. 2nd NSA Conference on High Confidence Software and Systems (HCSS'02)*, pages 69–77, March 2002.

[15] MULTOS™. Information at http://www.multos.com.