A Constructive Approach to Correctness, Exemplified by a Generator for Certified Java Card Applets

Alessandro Coglio and Cordell Green

Kestrel Institute, Palo Alto, California, USA
{coglio,green}@kestrel.edu
www.kestrel.edu

1 Position

Our position is that a constructive approach (namely, generating code from specs) can be a valuable alternative to post-hoc code verification. Our goal is to have a proof of full functional correctness of the code with respect to its spec. The automated generation of such a proof, along with the code, is guided by the availability of the code generation/design process. Generated proofs are checkable by a small and simple proof checker. A specification-first approach is made more widely accessible via user-friendly domain-specific notations. Domain-specific restrictions also simplify code and proof generation. Our experience with, and user acceptance of, early versions of the generator described herein, support this position.

2 Summary

Our approach uses automated construction steps (refinements) that are proven to have the desired property, in our case functional correctness. We exemplify this approach with AutoSmart [www.kestrel.edu/jcapplets], our generator for Java Card applets [java.sun.com/javacard]. AutoSmart converts our domain-specific specification language, SmartSlang, into the Java Card language. This approach does not depend upon verifying the generator – a proof is generated and checked for each applet generated.

AutoSmart first checks consistency properties of the source specification and then applies a series of transformations to the specification. Proofs are generated automatically as the transformations are applied. Each proof can then be checked by a simple proof checker. The correctness of this approach rests upon the correctness of the checker and of the formal specification of the semantics of both source and target language, and of course the logic language used to express the semantics. Only these artifacts need be trusted.

Our Metaslang language [www.specware.org] is used to express the semantics of both source and target languages and to express the correspondence between the source and target, i.e. "correctness". Metaslang includes an executable subset that was used to implement the generator.

2.1 Progress and Outlook

The system is working and in use. The current release (August 2005) includes the automatic translation from source to target and also includes the semantics of the source and target languages. Consistency properties of about half of the axiomatization of the target language have been automatically checked by a theorem prover, Snark [`www.ai.sri.com/ stickel/snark.html`]. Autosmart now includes a proof checker.

We believe the full system will be completed early next year. We foresee no scalability problems, having sampled proofs of the various parts. Scalability results from the simplicity and restriction of the domain-specific language, and the inherent simplicity of the applets generated. Since we need not deal with legacy code, we can employ a correct-by-construction approach, which simplifies the complexity of achieving a correctness guarantee. New tasks being undertaken, which also appear to be tractable, include the automated generation of a correct Java Card run-time environment, and the automated generation of the ancillary materials necessary to achieve certification.

Will this system, Autosmart, when completed, satisfy the Grand Challenge? Well, it will indeed provide proofs of both functional correctness and also security properties of software it produces. And the proofs will be tied to the software generated. The generator itself need not be proven. It will scale to reasonable Java Card applets. However, the input specification language is domain-specific and precludes certain features found in general-purpose languages (e.g. no recursion, no concurrency).

3 The Approach

Consider the automatic translation TR of artifacts written in a source language S into corresponding artifacts written in a target language T. "Corresponding" means that a certain relationship must hold between the artifacts. Such a relationship can be formally expressed in a logical language L, including formalizations of the semantics of S and T.

An approach to ensure the correctness of TR is to formalize TR in L, and prove a theorem stating that TR translates S artifacts to T artifacts such that the desired relationship holds. A concern with this approach is the gap between the formalization of TR in L and the actual implementation of TR. If TR is hand-written, then its formalization in L is only a model of the code that implements TR, and there is the possibility that the code does not behave exactly according to its model in L.

This concern can be overcome by deriving the code of TR via a provably correct refinement process that starts from the formalization of TR in L. However, if TR is sufficiently complex, performing such a derivation can be daunting, despite the relative maturity of current correct-by-construction technology. In addition, to play devil's advocate, the correctness of the TR code would depend on the tools used to derive it from the formalization of TR in L: who ensures the correctness of those tools?

Another approach to ensure the correctness of translations operated by TR is to have TR generate, along with the translation of an S artifact SA into a T artifact TA, also a proof P in L that the desired relationship between SA and TA holds. We also need a proof checker for L and two simple "encoders" that map respectively S and T artifacts to their representations in L. The proof checker is used to verify that P is a valid proof. The encoders are used to construct the formula in L that expresses the desired relationship between SA and TA. Finally, we check that the conclusion of P is the constructed formula. We do not ensure the correctness of TR in general, but just the correctness of particular artifacts produced by TR.

In this approach, we only need to trust the following items:

1. the proof checker for L;
2. the formalizations of S and T in L;
3. the formal expression of the desired relationship between S and T artifacts;
4. the encoders of S and T artifacts into their representations in L.

The size and nature of these items makes them easier to trust than the larger, more complex TR. A proof checker is usually quite small and simple; it can be derived straightforwardly from a mathematical definition of the logic of L. The formalizations of S and T in L may be large (depending on the complexity of S and T), but they can be inspected better than code can; in addition, expected formal properties about them can be proved (this may include testing, especially if the formalizations are executable or can at least be refined to executable versions). Similar remarks apply to the formal expression of the relationship between S and T artifacts. The encoders are also small and simple.

Of course, TR may generate a proof P that fails to pass the proof checker. Such a failure would typically uncover some bug in TR. However, if a proof P is valid and proves the formula derived from the artifacts SA and TA via the encoders, then we know that TA is a correct translation of SA, no matter how many bugs TR may have.

4 The Generator of Certified Java Card Applets

The source language S is SmartSlang (= smart card specification language), a domain-specific language tailored to smart card applets. SmartSlang features high-level constructs to express smart card concepts (ranging from communication with card readers to personal identification numbers) in a concise and convenient way. It also features an expressive type system (which includes, for example, integer ranges and enumerations with optional argument), built-in cryptographic operations, global invariants, and Java-like expressions and statements. Users find the domain-specific language simple and easy to use.

The target language T is Java Card, a version of Java tailored to smart cards. Java Card features a subset of the Java language (e.g. floating point numbers and dynamic class loading are left out), a different set of library APIs than standard Java (e.g. to handle communication with the card reader and to

perform cryptographic operations), and a specialized runtime environment with its own security model. Even though Java is relatively high-level, developing Java Card applets requires the programmer to deal with fairly low-level details, greatly increasing the potential for bugs.

The translator TR is AutoSmart (= automatic generator of smart Card applets), which consists of two components that operate one after the other. The first component checks various consistency properties of the SmartSlang specification, such as type safety. For instance, expressions assigned to state variables with integer range types are statically checked to always yield results within the ranges. A linear arithmetic decision procedure and a propositional reasoner are used to check type safety. The second component of AutoSmart generates Java Card code from the checked specification. Some of the results of type checking are used by the code generator: for example, the integer range types inferred for intermediate arithmetic results are used to decide which Java Card types to use to represent those values.

Translating SmartSlang into Java Card is not trivial. While some SmartSlang constructs almost directly correspond to Java Card constructs (intentionally), others are realized by multiple constructs spread through the Java Card code. For example, Java Card allocates objects from the heap but does not support garbage collection. Since memory is a very scarce resource in smart cards, Java Card programs must not willy-nilly allocate new objects during normal computation: all objects must be allocated during applet installation, and suitably re-used during normal computation. On the other hand, SmartSlang has no notion of objects and allocation, it just deals with values. Since some of these values are represented as objects in the Java Card code, AutoSmart must figure out, for the generated Java Card code, which objects to allocate and how to re-use them.

In the example applets we have worked on so far, the expansion ratio of SmartSlang into Java Card, measured in lines, is about 3-5; we expect it to rise to about 7 with the introduction into SmartSlang of additional high-level constructs that we have planned. The generated code is quite readable and not artificially verbose, rather close to what a human developer would write. The key to generating good-quality code is domain-specificity: despite the differences between SmartSlang and Java Card (e.g. explicitly allocated objects vs. just values), the two languages are relatively close to each other (after all, that they both describe smart card applets). In addition, smart card applets are relatively small (typically, only a few hundred lines), making their analysis and manipulation tractable.

The logical language L is Metaslang, the specification language of Specware™ [`www.specware.org`], a system for the rigorous development of software from formal specifications via provably correct refinement to code. Metaslang is based on higher-order logic; its features include predicate subtypes (as in PVS), first-order polymorphism (as in HOL), pattern matching (as in ML), and quotient types. Like the languages of popular higher-order theorem provers, Metaslang can be conveniently used to formalize the semantics of other languages. The logic of Metaslang has been formally defined and a proof checker for Metaslang exists.

In order to express the desired correspondence between SmartSlang specifications and Java Card programs, we must understand which features of those artifacts are relevant at the top level. The interaction between a smart card and a card reader is a master-slave one, where the reader is the master and the card is the slave. Accordingly, a smart card applet is a passive entity that maintains an internal state. When the applet receives a command (coming from the reader through the runtime environment of the card), it processes the command, possibly updating its internal state, and produces a response (sent to the reader through the runtime environment). Thus, the semantics of a smart card applet can be expressed, in essence, as a function that maps a state and a command to a new state and a response (i.e. a state machine).

A SmartSlang specification precisely describes the state of the applet, the commands it recognizes, and how each command is processed (command processing is expressed partly operationally, partly declaratively). We have developed a formalization of the SmartSlang language in Metaslang. We have formalized the abstract syntax and associated a formal semantics to the syntax. The top-level semantics of a SmartSlang specification is a state machine of the form described above.

A Java Card program consists of a set of classes. Each program must include methods that constitute the interface with the Java Card runtime environment. When the card reader sends a command, the runtime environment invokes a certain method, supplying the content of the command as argument. The method can perform arbitrary processing, including updating the state of the objects in the heap, using the library APIs, and constructing a response. When it terminates, a response is sent to the reader. We have developed a formalization of the Java Card language and APIs in Metaslang. The formalization currently covers the constructs and APIs targeted by the AutoSmart code generator. The top-level semantics of a Java Card program is also a state machine: the objects in the heap constitute the state, and the transition determined by a command is the net effect of the method invocation with that command.

The desired relationship between a SmartSlang specification and the corresponding Java Card program generated by AutoSmart is that they exhibit the same observable behavior. The internal state is not observable; the command-response exchanges are observable. So, in our formalizations of SmartSlang and Java Card, we associate the respective state machines with the set of all their possible command-response exchange traces in time (since we are not interested in real-time properties, traces are simply sequences conveying the relative time ordering of the exchanges). The formula that expresses the correctness of a particular Java Card program with respect to a particular SmartSlang specification says that the two artifacts have the same set of traces.

As of August 2005, AutoSmart does not yet generate proofs, but we are actively working on that. The code generation component of AutoSmart consists of various sequential phases. To reduce the complexity of proof generation, we are generating a proof from each phase, and obtain the end-to-end proof by composing the sub-proofs.

Once the proof generation capability is completed, we will be in a position to check the correctness of the translations performed by AutoSmart using the proof checker and the encoders. The encoders simply map the SmartSlang spec and the Java Card program to their abstract syntax representations in Metaslang, in order to construct the formula stating correctness (i.e. that the sets of traces associated to the two artifacts are equal). The proof checker is used to check the proof, yielding a formula that is the conclusion of the proof (if successful). A simple syntactic check finally ensures that the conclusion of the proof coincides with the correctness formula.

The capability to establish the correctness of smart card applet implementations with respect to the specifications without having to trust the generator is particularly valuable for achieving third-party certification. Information technology security standards such as the Common Criteria and FIPS 140-2 (for cryptographic modules) require the developer to provide, for the highest levels of certification, proofs of correctness of the code with respect to requirements.

5 Why a Constructive Approach?

In many software situations we must deal with assurance of legacy code, which is verified by a post-hoc method of proving certain properties, or possibly functional correctness. But the combinatorial difficulty of a post-hoc approach has generally prevented the community from being able to prove full functional correctness, i.e. that the program actually does what is intended. For example, we may know with high assurance that some glue code between programs does not overflow a buffer, but not have a proof that it correctly glues components so that the ensemble is correct.

But often we need not be relegated to just analyzing legacy code, and instead are allowed to develop new software. Here the approach of applying constructive design knowledge offers advantages. Most significantly, the intrinsic problem complexity is reduced so that we can prove full functional correctness. This complexity reduction is a consequence of a synthetic versus an analytic approach. That is, the number of ways one functional specification can be implemented is large. On the other hand, while most of these implementations are redundant or differ in unimportant ways, an analytic approach must be able to capture any implementation given. But a generator need only generate a reasonable and smaller number of implementations. Then evolution and maintenance are carried out on the source specification, avoiding the need for target code analysis. Further understanding this question of verification-complexity reduction is a suggested research topic. Empirically, we typically find a factor of about 3-5 increase in complexity or size moving from spec to code.

6 Conclusion

We have described a constructive approach to correctness, in which a generator generates checkable proofs from the transformations that it performs. We have

exemplified the approach with the description of a generator of smart card applets. A key feature of the approach is that the generator need not be trusted. We need only trust the proof checker as well as the axiomatization of the semantics of the source and target language and of the correctness relationship between them, which appears to be a "minimal" set of artifacts to be trusted in order to formally establish correctness. We have also discussed potential advantages of a constructive approach over post-hoc verification, for the case where we have the opportunity to develop new code as opposed to using legacy code.