FROM FINITE STATE MACHINES TO PROVABLY CORRECT JAVA CARD APPLETS

Engelbert Hubbers, Martijn Oostdijk, and Erik Poll
Nijmegen Institute for Information and Computing Sciences, University of Nijmegen, P.O.Box 9010, 6500GL, Nijmegen, The Netherlands
{hubbers,martijno,erikpoll}@cs.kun.nl

Abstract This paper presents a systematic approach to developing Java Card applets and/or formal specifications for them, starting from descriptions in the form of finite state machines. The formal specifications are written in the specification language JML, and can be checked against Java Card source code using the static checker ESC/Java.

1. Introduction

Specification of software, let alone *formal* specification of software, is difficult, as is checking that source code actually conforms to a given specification. Often finite state machines (FSMs) provide a convenient notation for specification, for example for the life cycle of an applet or the access control it enforces. The aim of this paper is to show that, for the kind of software running on small devices such as smart cards, it is possible to automatically generate machine-checkable formal specifications from such FSMs, just as it is possible to generate (a skeleton of) the actual source code itself.

Our approach is to automatically generate prototype Java Card applets and formal specifications in JML [Leavens et al., 1999], starting from a description in the form of an FSM. Additional functionality is to be manually added to the generated applet source code afterwards. Also, more details can be added to the generated specifications.

Instead of contributing any grand new theories to the field of software engineering, this paper explores the above approach using mainly existing technologies and tools, as described in Section 1.2. How Java Card applets and JML specifications are automatically generated is described in Section 1.3. Possibilities for future work and some conclusions can be found in Section 1.4.

2. Applied Technologies

Several tools and languages are used in the transformation from FSM to Java code and JML specifications.

The Uppaal model checker [Pettersson and Larsen., 2000] is used to draw diagrams of FSMs which can be saved as XML documents. We do not use Uppaal's verification capabilities. An FSM in Uppaal consists of states (of which one is marked as initial) and transitions between these states. Every transition has a label and possibly additional attributes, such as guards and assignments.

The Java Modeling Language (JML) [Leavens et al., 1999] is a specification language tailored to Java. JML specifications consist of class invariants, global constraints, and pre- and postconditions for individual methods. These are all written as special comments in the Java source code, so that Java compilers do not notice them, but JML-aware tools can make use of them. One of the strengths of JML is that its syntax resembles normal Java syntax, so that Java programmers can easily understand it. Another strength of JML is that there are several tools available for it: the JML runtime assertion checker [Leavens et al., 1999], ESC/Java [Detlefs et al., 1998], and the LOOP tool [Van den Berg and Jacobs, 2001].

The F2J tool is a simple translator of our own creation which takes a Uppaal XML document as input and generates Java source code and JML specifications. This tool is described in Section 1.3.

The ESC/Java 'extended static checker' is a tool developed at Compaq SRC for automatically checking JML-annotated code. It detects, at compile time, possible violations of specifications by a program. ESC/Java is extremely good at spotting for example where one might potentially dereference a null pointer or access an array outside its bounds, which typically violates some postcondition or invariant. ESC/Java uses a theorem prover to verify assertions in code, without any user interaction. The tool is neither sound nor complete, which means that ESC/Java might complain about errors that will never arise when the program is run, and that ESC/Java might miss errors that may occur at runtime. However, this drawback usually only occurs in more complicated programs and specifications, whereas the applets and their properties we look at are relatively simple. Moreover, the bold step to give up on soundness and completeness is what makes ESC/Java such an effective tool, as attempting either of these two properties would probably result in a tool far too complicated and slow to be useful in practice.

3. Generating Applets and Specifications

Java Card applets have a very standard structure. They all implement a *process method* which deals with so-called *application protocol data units* (APDUs), instruction packets that are sent to the card by a terminal (and back). APDUs have a header which encodes the instruction to be performed by the applet, while the rest of the APDU is just data. Typically, the process method inspects the header, to determine which operation is to be performed, and calls dedicated helper methods to process the rest of the data. Depending on the state of the applet an instruction may be valid or invalid. A typical applet will throw an exception when it receives an invalid instruction.

In order to generate Java code from the FSM, a special `mode` field is introduced to encode the state of the FSM. The transitions in the FSM correspond to the possible instructions that are valid for the applet. The `mode` field implements a life cycle model for the applet, similar to those in [Marlet and Métayer, 2001; Mostowski, 2002]. The process method is generated as a nested switch. The outer switch is a case distinction on `mode`. The inner switches are case distinctions on the header of the incoming APDU. A dedicated helper method is generated for each possible instruction. The generated code ensures that, in case the corresponding guard is true in the FSM, `mode` is changed to the result state of the transition and all corresponding assignments are executed. Invalid instructions result in exceptions being thrown, leaving `mode` unchanged. The generated code is a legal Java Card applet with limited functionality. Typically one will want to manually add some code to the generated skeleton code. Our example in this paper is a simple purse applet whose Uppaal specification is given in Figure 1. Since Uppaal does not distinguish between normal and abnormal transitions, the labels in our diagrams have names ending with `Nor!` or `Exc!`.

The generated JML specification consists of a global *invariant*, a global *constraint*, and pre- and postconditions for all methods. The generated invariant enumerates the different possible values of `mode`.

```
/*@ invariant
  @  (mode==INIT || mode==ISSUED || mode==CHARGING || mode==LOCKED);
  @*/
```

An invariant is a predicate that has to be preserved by every method. A constraint in JML is a relation between pre- and post-states of methods that has to be respected by every method. Contrary to an invariant, a constraint can make use of JML's `\old` keyword to refer to the value of a field in the pre-state of a method. The generated constraint captures the

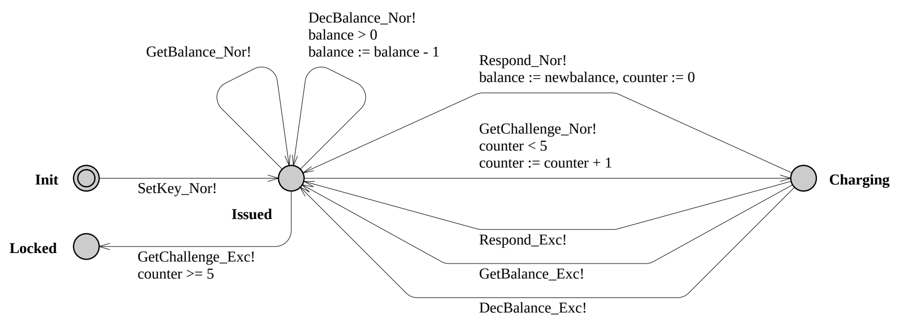

Figure 1. Life cycle model of a simple purse applet.

flow of control throughout the applet's life, by listing logical implications describing the different transitions.

```
/*@ constraint
  @  (mode==LOCKED ==> \old(mode)==ISSUED || \old(mode)==LOCKED) &&
  @  (mode==INIT ==> \old(mode)==INIT) &&
  @  (mode==ISSUED ==> \old(mode)==INIT || \old(mode)==ISSUED
  @                  || \old(mode)==CHARGING) &&
  @  (mode==CHARGING ==> \old(mode)==ISSUED || \old(mode)==CHARGING) &&
  @  (\old(mode)==LOCKED ==> mode==LOCKED) &&
  @  (\old(mode)==INIT ==> mode==ISSUED || mode==INIT) &&
  @  (\old(mode)==ISSUED ==> mode==ISSUED || mode==CHARGING
  @                         || mode==LOCKED) &&
  @  (\old(mode)==CHARGING ==> mode==ISSUED || mode==CHARGING);
  @*/
```

The generated constraint only describes the control flow. Additional constraints could be added, for instance to relate `mode` to the other fields. For example, an interesting relation between `mode` and `balance` is that 'once the applet is `ISSUED`, the `balance` can only decrease'. It should not be very difficult to generate such additional constraints based on more advanced analysis of the FSM.

Verifying the kind of properties discussed above is well within the capabilities of ESC/Java, so it can be used to check that a Java Card applet behaves conform the generated JML specification. Applets will, of course, invoke methods from the Java Card API. Fortunately, JML specifications for all these methods are available [Poll et al., 2000]. Note that control flow, even in a simple Java Card program, can be hard to follow due to all the possible exceptions. So the verification performed by ESC/Java is more complicated than it may appear.

4. Discussion

We have discussed an approach to use FSMs to automatically generate Java Card source code, that can serve as the skeleton for a smart card applet, and to generate formal JML specifications, that are well-suited to automatic verification using ESC/Java. The F2J translation described in this paper is still under development, and is currently just a prototype. The invariant and constraint generation can certainly be improved through more advanced analysis of the FSM.

One obvious next step is to use Uppaal not just as a graphical editor, but to use it as a model checker. It could be used to check some interesting properties of an FSM (for example, for our example, that the balance always remains non-negative) even before we generate source code or specifications. In addition to just modeling the smart card applet, one could also model the behavior of the terminal application that communicates with the smart card, and then check more interesting properties of the applet and the terminal interacting. Another possibility for further work is to extend the syntax allowed in FSMs, notably in the guards and assignments, so that more of the behavior of the Java applet can be described in the FSM. Finally, a more ambitious project for future work would be to extract an FSM from given Java Card source code, rather than the reverse as we do now.

References

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. (1998). Extended Static Checking. Technical Report 159, Compaq Systems Research Center.

Leavens, G. T., Baker, A. L., and Ruby, C. (1999). JML: A notation for detailed design. In Kilov, H., Rumpe, B., and Harvey, W., editors, *Behavioral Specifications for Businesses and Systems*, pages 175–188. Kluwer Academic Publishers.

Marlet, R. and Métayer, D. L. (2001). Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic.

Mostowski, W. (2002). Rigorous development of JavaCard applications. In Clarke, T., Evans, A., and Lano, K., editors, *ROOM 4, London.*

Pettersson, P. and Larsen., K. G. (2000). UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44.

Poll, E., van den Berg, J., and Jacobs, B. (2000). Specification of the Java Card API in JML. In *CARDIS'2000*, pages 135–154. Kluwer.

Van den Berg, J. and Jacobs, B. (2001). The LOOP compiler for Java and JML. In *TACAS'01*, volume 2031 of *LNCS*, pages 299–312. Springer-Verlag.