Automata-Based Programming Technology Extension for Generation of JML Annotated Java Card Code

Andrey A. Klebanov

Abstract—**This paper gives an overview of the ongoing research project which concerns generation of dependable *Java Card* code. According to the automata-based programming technology, code is generated from a high-level application behavior description which is based on finite state machines. An extra benefit from the use of such description is the possibility of generation of formal application specification in *Java Modeling Language*. Conformance of the code against its specification could be checked by different static checking and verification tools.**

Index Terms—**Finite state machines, Smart cards, Software requirements and specifications, Software verification and validation.**

I. Introduction

A smart card [1] is a card that is embedded with a chip, which allows to store and process information. In fact, smart cards are secure credit card size computers. Besides being secure, smart cards have several advantages such as mobility and simplicity of use. So, the main domains of their use are secure storage of data, business transactions, authentication, and so on.

Java Card technology [1], [2] provides a *Java* platform for smart cards. It brings all the benefits of using *Java*; it also simplifies development process, as it covers all vendor specific card features. *Java Card API* is a superset of *Java API* subset – this means that *Java Card* does not support multithreading, strings, multidimensional arrays, garbage collection, and et cetera due to the cards' limited resources, but *Java Card API* includes some extra functionality, required to handle tasks from smart cards domain of use. For example, sending and receiving special commands (*APDU* commands), working with *PIN* codes and cryptographic algorithms, and much more.

Manuscript received March 31, 2008.

A. A. Klebanov is BSc student at the Computer Technologies Department, Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (phone: 955-29-00; e-mail: klebanov.andrey@gmail.com).

The research is supervised by A. A. Shalyto, PhD, professor at the Computer Technologies Department, Saint-Petersburg State University of Information Technologies, Mechanics and Optics, Saint-Petersburg, Russia (e-mail: shalyto@mail.ifmo.ru).

There are number of reasons for *Java Card* to become an attractive field for formal methods researchers. *Java Card* applets are dependable systems which must be secure and bug-free and, in contrast with *PC* software; it could be very difficult to make updates when cards are issued. Moreover, limited resources and lack of some *Java* features guarantees, that applets will not be very complicated. So, formal verification is a feasible task for *Java Card* applications.

In this paper we introduce an automata-based programming technology extension for generation of *Java Card* applet code skeleton, which also implements application logic, with formal specification in *Java Modeling Language* (*JML*).

The rest of the paper is organized as follows. Section II introduces automata-based programming, Section III describes applied technologies. Code generation and verification details are introduced in Section IV; they are further illustrated with example in Section V. An overview of related work is given in Section VI and, finally, Section VII concludes.

II. Automata-Based Programming for Java Card

Synchronous programming [3] is believed to be one of the best approaches to application development for embedded and reactive systems. In this paper we use a sort of synchronous programming – automata-based programming [4], [5], which is also known as *Switch*-technology [6]. According to this paradigm, programs are treated as systems of automated controlled objects. Each system consists of control system (system of cooperating automata) and controlled objects. Automata-based programming technology defines two types of diagrams for application description – connectivity schema and transition graphs. Connectivity schema describes relation between event providers, state machines and controlled objects; transition graph describes behavior of corresponding finite state machine. But such complicated methodology is not only good for application design; it also helps to close the gap between specification and implementation, as there exist a formal and isomorphic way to generate a skeleton of code from the description. To prove code conformance against its specification we also generate *JML* annotations.

Let us consider some *Java Card* features which make automata-based programming use reasonable. Interaction with

a smart card is achieved by means of interface devices (card readers). Card reader powers the card and also provides communication channel between the card and a *PC* or a terminal with a host application installed on it. Communication channel is half duplex; master-slave model is used where smart card is always a slave while terminal is a master. In other words, smart cards always wait for a command from a terminal. Thus their interaction is "event driven". Commands are passed one by one from a host application to *Java Card Runtime Einvironment* where they are forwarded to the selected applet. After the command handling, card reply goes inversely the whole way back to the host application. All *Java Card* applets have the same structure – they should extend the base class `Applet`. We will only discuss one of the methods to be extended – `process` method, which is invoked every time when applet receives a command from the host application. Depending on the state of the applet, `process` method either calls dedicated helper method to handle the command or throws exception. In our approach all the logic of the `process` method is described in terms of finite state machines and a skeleton of applet code (including logic implementation) is generated. Only stubs are generated for helper methods.

III. Applied Technologies

A. JML

JML [7] is a behavioral interface specification language designed to specify *Java* classes and interfaces. It means that it both specifies behavior and syntactic interface of *Java* code. *JML* is based on design by contract [8] and model-based specification [9] approaches. *JML* was specially designed to be used by *Java* programmers, including those with just basic mathematical education. Therefore it uses slightly extended *Java* syntax. It was also intended to be independent from program design manner. Specifications written in *JML* are annotations for *Java* code (they could be also stored in a separated file) – thus *Java* compiler treats them as comments and simply ignores, but *JML*-aware tools can use them.

We shall briefly discuss some of the *JML* features which are used in our research. Key elements of *JML* specification are preconditions, postconditions, and invariants. Precondition for a method is a logical predicate which must hold for the method invocation. Postcondition for a method is a logical predicate which must hold after each execution of the method. And, finally, class invariant constrains all the methods of the class. *JML* also provides `\old(var)` expression which refers to the value of the variable `var` at the moment of entrance to the method (before its execution). It is convenient to use `constraint` keyword, `\old()` expression and logical constructions, such as `a ==> b` (implication) or `a <==> b` (equality) to constrain variable's value changing in time.

Great number of tools which support *JML* has been developed. Overview of them is given in [10]; here we mention only some of the static checking and verification tools important for our research. It is too ambitious to hope to proof all the correctness and security properties of the application. The task becomes more feasible if attention is concentrated on a subset of the properties. For example, static checking tools can guarantee that all the array indices will never get out of bounds. Such property has a great practical value. Static checking technique lies between compiler type checks and full program verification [11]. *ESC/Java2* [12] is a static checking tool which can automatically detect some common errors such as array index out of bounds or null pointer dereferencing. Several verification tools such as *KeY* [13], *LOOP* [14], and *JACK* [15] are designed to work with *JML*, they can handle more complex tasks, but user interaction is a price for that.

B. UniMod

UniMod [16] stands for *Unified Modeling*; the goal of the project is to create a methodology for the whole application development cycle – from design to implementation stage. The tool supports automata-based programming technology and adopts it to current *UML* standards. It allows to describe application and to generate skeleton of *Java* source code, but it does not support *Java Card*. Validation techniques for finite state machines are embedded in *UniMod*. Thus each state is reachable from the initial one and has a complete and consistent set of outgoing transitions.

IV. Code Generation And Verification Details

We use *UniMod* application behavior description to generate both *Java Card* code and *JML* specification for it. As it was already mentioned *Java Card* applets have standard structure. Here we will only discuss application logic which is encapsulated in the `process` method. Automata-based programming is also known as *Switch*-technology, so `process` method may consist of just one or two nested `switch` operators. For example, if two operators are used – outer one is for finite state machine states and inner one is for commands. If a command is valid for the current state a helper method is called to handle this command; otherwise an exception is thrown. Host application commands and card replies are treated as events from event providers. Moreover, according to the automata-based programming technology, transitions between states my optionally contain boolean expressions with input actions. They are transformed to *JML* preconditions for the on enter to state methods. Currently we are also able to describe control flow – specify that applet could only be in the one of predefined states (via *JML* `invariant` keyword) and describe incoming and outgoing transitions for each state (using `constraint` keyword and `\old()` expression).

As for implementation part – *UniMod* has a feature to convert application description to *XML* file. Afterwards we

use a template engine to generate code. This research is still on its early stage and some of the implementation aspects should be cleared. For example, we still discuss the way of generating *JML* specifications. On the one hand, they could be generated together with the code but this may lead to very complicated templates and on the other hand, it is possible to parse *XML* file and retrieve all the required information for specification generation. The last step in our approach is static checking and/or formal verification of the code. Variety of tools developed for *JML* provides a wide range of directions for further work. We believe that *ESC/Java2* tool will be used for static checking, as for verification tool – it is still an open question.

V. Case Study

Here we introduce a small example of a *Java Card* applet description with its *JML* specification. Connectivity schema is described on the Fig. 1. It is a way of class diagram representation, according to the automata-based programming technology. Host application commands and *Java Card* replies are treated as event providers. For example, "Verify *PIN*" command is an event from the host application. *Java Card* is a controlled object, because it handles incoming commands. Event providers and controlled object are related by finite state machine; its transition graph is shown on the Fig. 2.

Applet functions as follows. After applet's initialization phase it is required to verify *PIN* code (event `e1`). If it is correct applet state is changed to "Do something"; where wrong attempt counter is made zero via output action method `o1.z3` and something is done via method `o1.z4`. If *PIN* code is not correct and number of wrong attempts is less or equal to three (boolean expression with input action method `o1.x3<=3`) the state is remained unchanged, but value of the wrong attempts counter is incremented. If *PIN* code is not correct and the wrong attempts counter exceeds three, *SIM* card locks itself and informs host application about that.

Following *JML* specification describes control flow. Let variable `state` to describe automaton state. So specification:

```
/*@ invariant
    (state == APPLET_INITIALIZATION) ||
    (state == VERIFY_PIN) ||
    (state == DO_SOMETHING)        ||
    (state == SIM_CARD_IS_LOCKED);
@*/
```

guarantees that applet always remains in the one of predefined states. The way to constraint transitions between states is more bulky:

```
/*@ constraint
  ((state == APPLET_INITIALIZATION) ==>
  (\old(state) == APPLET_INITIALIZATION))
  &&
  ((state == VERIFY_PIN) ==> ((\old(state)
  == VERIFY_PIN) ||
  (\old(state) == APPLET_INITIALIZATION)))
  &&
  ((state == DO_SOMETHING) ==>
  ((\old(state) == VERIFY_PIN) ||
  (\old(state) == DO_SOMETHING))) &&
  ((state == SIM_CARDS_IS_LOCKED) ==>
  ((\old(state) == VERIFY_PIN) ||
  (\old(state) == SIM_CARDS_IS_LOCKED)))
  &&
  ((\old(state) == APPLET_INITIALIZATION)
  ==> ((state == VERIFY_PIN) ||
  (state == APPLET_INITIALIZATION))) &&
  ((\old(state) == VERIFY_PIN) ==>
  ((state == VERIFY_PIN) ||
  (state == DO_SOMETHING) ||
  (state == SIM_CARDS_IS_LOCKED))) &&
  ((\old(state) == DO_SOMETHING) ==>
  (state == DO_SOMETHING)) &&
  ((\old(state) == SIM_CARDS_IS_LOCKED)
  ==> (state == SIM_CARDS_IS_LOCKED));
@*/
```

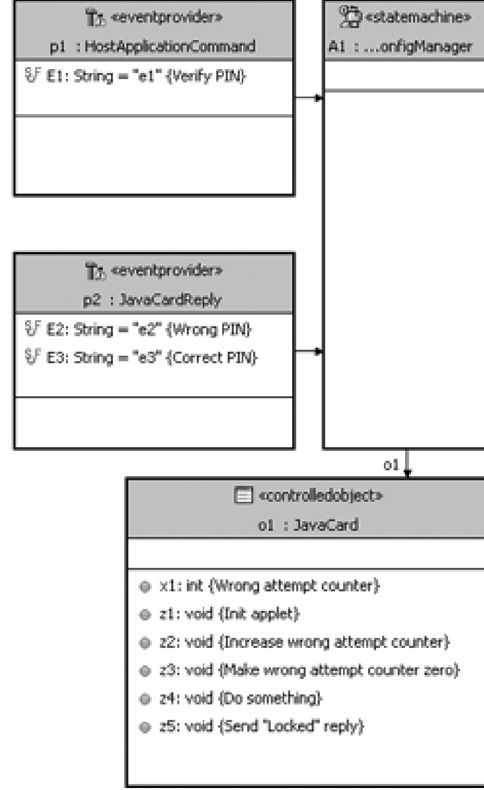

Fig. 1. Connectivity schema.

In our approach exceptions caused by illegal commands do not change automaton state. Let us denote set of automaton states by Q, hence, $\forall\ q \in Q$ we generate `\old(state) == q <==> state == q` constraint.

Finally, let us consider transition between state "Verify *PIN*" and "*SIM* card is locked". Input action method `x1` should be declared as `pure`. This is specification only property, which means that this method does not have side-effects and thus could be called in specification. In other words, specification should not change state of the application. So, precondition for on enter to state method of "*SIM* card is locked" state is `//@ requires x1 > 3;`

Though *Java* is a high-level programming language, *Java Card* has several low-level features. For example, *APDU* commands are encoded in byte arrays. In our approach

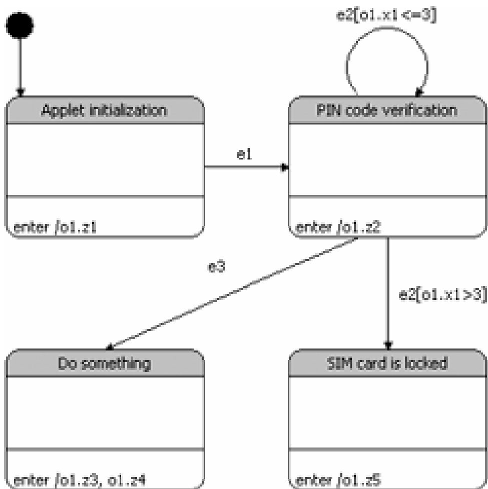

Fig. 2. Automaton transition graph

commands are treated as events, therefore their short names and natural language descriptions are used for application design and code generation. Some of the standard commands and card replies are already predefined. So, programmer uses convenient notation, but not byte arrays. By covering some of the low-level aspects, we decrease number of potential bugs and therefore increase code dependability.

VI. Related Work

There exist two research projects aimed on generating *Java Card* code from high-level specification. The first one is introduced in [17]. Special language, *SmartSlang*, is used there to describe applets with high-level constructions which are specific to smart cards. Declared benefit from those constructions is the possibility to generate more functionality, but use of a complex base system is required. Such approach does not have too much in common with ours, except that finite state machines are used during code generation.

The second one is described in [18], [19]. In [18] verification tool is used as an editor for finite state machine modeling, [19] uses *UML* to describe automata. Our work has several advantages over those researches. First, we use automata-based programming paradigm, but not only finite state machine description of applet. The main difference is that we describe application behavior rather than provide static description of application structure. Secondly, generated constraints only describe the control flow in contrast to our approach, where in addition some preconditions are also generated. Moreover, in our approach a notation which covers low-level aspects is provided to manipulate with commands and card replies. Finally, there are special model checking add-ons for *UniMod*, so it is also possible to verify application model [20]. Verification capabilities of the environment in [18] are not used.

As for *JML* – there has been done a lot of research and case studies about using and combining different tools designed to support it. *JML* is used in many areas, and is popular both in industry and academia.

VII. Conclusion

Automata-based programming technology provides a methodology for full development cycle of reliable applications. Proposed extension is intended to make applications even more correct by uniting model checking, design by contract, and verification of real implementation, but not only application model. We are planning to extend our approach to *Java ME* midlets; and create a full mobile solutions development framework, based on the *UniMod* tool.

References

[1] Z. Chen, *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*, 1st ed., Addison-Wesley, 2000.

[2] M. Baentsch, P. Buhler, T. Eirich, F. Höring, M. Oestreicher, "JavaCard – From Hype to Reality," *IEEE Concurrency*, pp. 36–42, Dec. 1999.

[3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, R. de Simone, "The synchronous languages 12 years later," in *Proc. IEEE*, vol. 91, no. 1, Jan. 2003, pp. 64–83.

[4] A. A. Shalyto, "Logic Control and "Reactive" Systems: Algorithmization and Programming," *Automation and Remote Control,* vol. 62, no. 1, pp. 1–29, 2001.

[5] A. A. Shalyto, "Software Automation Design: Algorithmization and Programming of Problems of Logical Control," *J. of Comput. and Systems Sci. Int.*, vol. 39, no. 6, pp. 899–916, 2000.

[6] A. A. Shalyto, N. I. Tukkel, "SWITCH-Technology: An Automated Approach to Developing Software for Reactive Systems," *Programming and Comput. Software*, vol. 27, no. 5, pp. 260–276, 2001.

[7] G. T. Leavens, A. L. Baker, C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," Iowa State Univ., Dept. of Comput. Sci., Tech. Rep. 98-06u, Apr. 2003.

[8] B. Meyer, "Applying design by contract," *Computer*, 25(10), pp. 40–51, Oct. 1992.

[9] I. Sommerville, "Model-based Specification," in *Software Engineering*, ed. 5, Addison Wesley, 1995.

[10] L. Burdy, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll , "An overview of JML tools and applications," *Int. J. on Software Tools for Technology Transfer (STTT)*, 7(3), Jun. 2005, pp. 212–232.

[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe, "Extended static checking," Res. Rep. 159, Compaq Systems Res. Center, Dec. 1998.

[12] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2," Nijmegen Inst. for Computing and Inform. Sci., Tech. Rep. NIII-R0413, May 2004.

[13] W. Ahrendt *et al.*, "The KeY Tool," Dept. of Computing Sci., Chalmers Univ. and Goteborg Univ., Goteborg, Sweden, Tech. Rep. no. 2003-05, May 2003.

[14] J. van den Berg, B. Jacobs, "The LOOP compiler for Java and JML," in T. Margaria and W. Yi, editors, *Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 2031 of Lecture Notes in Comput. Sci., Springer-Verlag, 2001, pp. 299–312.

[15] L. Burdy, A. Requet, "JACK: Java Applet Correctness Kit," in *GDC 2002*, Singapore, Nov. 2002.

[16] V. S. Gurov, M. A. Mazin, A. S. Narvsky, A. A. Shalyto, "Tools for Support of Automata-Based Programming," Programming and Comput. Software, vol. 33, no. 6, pp. 343–355, 2007.

[17] A. Coglio, "An Approach to the Generation of High-Assurance Java Card Applets," in *Proc. 2nd NSA Conf. on High Confidence Software and Systems (HCSS'02)*, Mar. 2002, pp. 69–77.

[18] E. Hubbers, M. Oostdijk, E. Poll, "From finite state machines to provably correct Java Card applets," in D. Gritzalis *et al.*, editors, *Proc. 18th IFIP Inform. Security Conf.*, 2003, pp. 465–479.

[19] E. Hubbers, M. Oostdijk, "Generating JML specifications from UML state diagrams," in *Proc. Forum on specification and Design Languages (FDL'03)*, Sep. 2003.

[20] E. V. Kuzmin, V. A. Sokolov, " Modeling, Specification, and Verificationof Automaton Programs," *Programming and Comput, Software*, vol. 34, no. 1, pp. 27–43, 2008.