Generating JML Specifications from UML State Diagrams

Engelbert Hubbers, Martijn Oostdijk
SoS Group, NIII, Faculty of Science, University of Nijmegen
{hubbers, martijno}@cs.kun.nl

Abstract

We describe our work on the prototype AutoJML tool which automatically derives JML (Java Modelling Language) specifications from UML state diagrams. UML is widely used for modelling Object-Oriented systems on an abstract level. It should be possible to test whether concrete program code actually implements a UML model. The generated JML specifications can be used for this. There exist tools to verify whether Java code conforms to a JML specification.

1 Introduction

Specification and verification of software is difficult and time consuming. Yet, strict compliance to specifications is becoming increasingly important for standardisation and security certification purposes. Formal verification of source code can help ascertain compliance to specifications. However, while verification of source code relative to an existing specification is already hard, getting the detailed specification right is probably even harder.

UML state diagrams or, more generally, finite automata, provide a convenient formalism for formally specifying systems. However, there is a big gap between such abstract specifications and the actual implementation. This gap leads to the concern whether the implementation really behaves according to the specification. In this paper we try to bridge this gap by generation of JML (the Java Modelling Language) specifications. JML is a language to specify Java programs at source code level. Basically JML introduces class wide invariants, and pre- and postconditions for methods. Java code can be automatically checked against JML specifications in several ways, ranging from runtime tests to formal verification. A growing collection of more or less compatible tools is becoming available for JML: the JML runtime assertion checker [LBR99], ESC/Java [F$^+$02], Daikon [ECGN01], Krakatoa [MPMU03], and the Loop tool [J$^+$98] developed in Nijmegen.

We have built a prototype tool, named *AutoJML*, which generates JML specifications based on UML state diagrams. A software engineer still needs to write the program (based on stubs generated by the UML tool), but the program can be verified against this generated specification. The generated specification may also be refined by hand. For the purposes of this paper we use ESC/Java to check that the implementation meets the generated specification.

Much research is being done towards closing the gap between high level and low level specification formalisms. Previously we presented a Uppaal [Upp] to Finite State Machines (FSM) translation tool in [HOP03a], and we did a case study of refinement of security protocols in [HOP03b].

The security community is looking for ways to enforce security properties on programs, without taking away flexibility from high level programming languages. See for example [CF00], where runtime checks for programs are generated, based on a high level specification in the form of a regular expression. In [HRS02] UML models are translated to proof obligations, resulting in a verified Java Card applet.

In the literature there are several attempts of generating code, generating specifications, or both from UML models. We mention the work of [SKM01], [LP99] and [vL01]. The common divisor of these projects is that they all generate Promela [Hol97] specifications, the input language for the SPIN model checker [Hol97]. The differences are found mainly in the UML part: which part of UML is supported as input: activity diagrams, collaborations, or state diagrams like in our case.

What is different in our approach is that we use JML as a standard specification language, so that different tools can be used to verify Java code against a specification. We do not verify properties of the UML state diagram itself. We rather generate JML specifications that assert that some piece of concrete program code implements the given UML state diagram.

Our research focuses on building concrete tools. It should be stressed, however, that this is work in progress. The AutoJML tool is applied to an example in Section 2, which yields a working, and verified, Java Card applet. The implementation details of AutoJML are discussed in Section 3. Ideas for future work are listed in Section 4.

2 General setup

We started the AutoJML project as a tool to transform a finite state machine as specified in Uppaal's diagram editor to both JML specifications as well as Java code. In the current paper, we add a new front-end in order to accept also UML state diagrams as input, as indicated in Figure 1. The choice for ArgoUML [R$^+$00] is arbitrary. Any UML tool that exports state

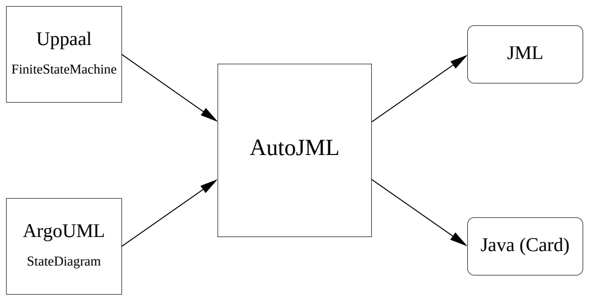

Figure 1: Abstract view of the AutoJML architecture

diagrams in the so-called XMI format should work. XMI stands for XML Meta-data Interchange, and is designed to enable interchange of meta-data between OMG-UML based modelling tools [OMG00].

To make the whole generation process more clear, this section explains the method using an example taken from the world of the Global System for Mobile communications (GSM). Details of the approach are given in Section 3.

Example *GSM uses a cryptographic protocol to authenticate the Subscriber Identity Module (SIM), which is embedded in the cell phone, to the network service provider. Although this is essentially a simple challenge-response protocol, great care must be taken to make sure the implementation is secure and in accordance with the GSM standard. So, GSM authentication is a small yet realistic case study, moreover its technical details are freely available since the ETSI GSM standard [ETS01] is made public.*

The protocol specified in the GSM standard can be summarised abstractly as follows. The network provider sends a challenge RAND *to the phone which forwards it to the SIM. The SIM computes a response* SRES *and a session key* K_C *by encrypting* RAND *with a secret key* K_I *shared* *by the network provider and the SIM. The response,* SRES*, is sent back to the cell phone, which forwards it to the network provider where it is checked for validity. If the response checks out, the session key is used to ensure confidentiality of the conversation that follows the authentication phase.*

On a more concrete level the standard describes the protocol using two commands which can be sent to the SIM: `RUN_GSM_ALGORITHM` *and* `GET_RESPONSE`*. The key* K_I *is a 16-byte number stored in the SIM, the challenge* RAND *is a 16-byte number sent to the SIM in the data field of the byte sequence representing the* `RUN_GSM_ALGORITHM` *command. The response* $(\text{SRES}, \text{K}_\text{C})$*, a 12-byte number (4 for* SRES*, 8 for* K_C*) is the result of running the Comp128 algorithm with key* K_I *and input* RAND*. This is sent back in the data field of the response byte sequence after sending the* `GET_RESPONSE` *command. The* `RUN_GSM_ALGORITHM` *command should be followed by a* `GET_RESPONSE` *command.*

As it turns out, the Comp128 encryption algorithm that was originally proposed has some technical problems: in a chosen plain text attack $150,000$ *challenges sent to the SIM will reveal all 128 bits of the key [BGW]. To overcome this problem, the service provider can implement an alternative algorithm, but for economical reasons it seems much more practical to just introduce a counter in the SIM, which is incremented with each authentication request. The SIM could lock itself after say* $10,000$ *authentication requests. This will not influence normal usage of the SIM, since that many requests are not expected in its lifetime.*

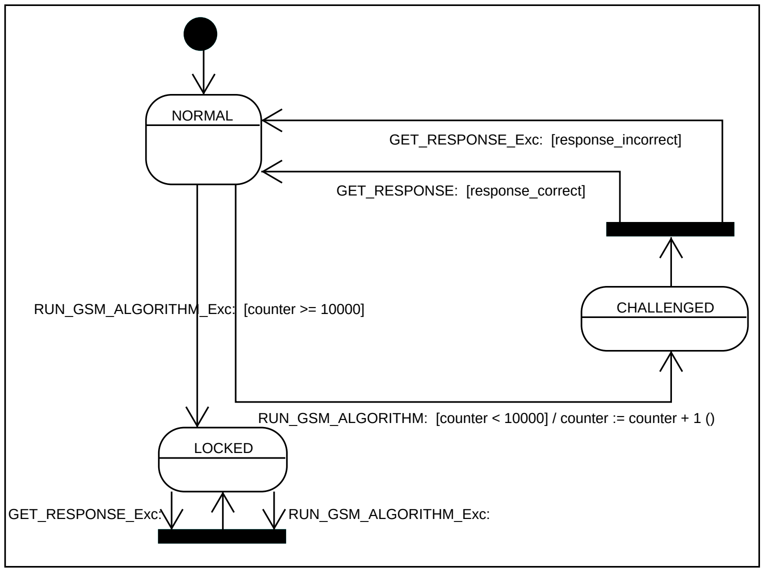
NORMAL
GET_RESPONSE_Exc: [response_incorrect]
GET_RESPONSE: [response_correct]
RUN_GSM_ALGORITHM_Exc: [counter >= 10000]
CHALLENGED
RUN_GSM_ALGORITHM: [counter < 10000] / counter := counter + 1 ()
LOCKED
GET_RESPONSE_Exc:
RUN_GSM_ALGORITHM_Exc:

Figure 2: GSM authentication state diagram

The SIM part of the protocol, including the counter mechanism, is specified in the UML state diagram[1] *in Figure 2. The state diagram consists of three states, labelled* `NORMAL`, `CHALLENGED`, *and* `LOCKED`*. The horizontal bars and the dot are UML pseudo-states. They are discussed in Section 3. For now it suffices to simply follow the transitions that pass through these objects. Initially the SIM is in* `NORMAL` *and counter equals zero. By processing a* `RUN_GSM_ALGORITHM` *command, the SIM changes to* `CHALLENGED`*, except when the counter is greater than* $10,000$*, in which case the SIM becomes locked. When in* `CHALLENGED`*, a* `GET_RESPONSE` *will get the SIM back to* `NORMAL`*, whether the response was validated or not.*

The output generated by the current version of AutoJML, when given the state diagram in Figure 2 as input, consists of the source code for a Java Card applet with JML annotations. To

[1]This diagram was generated by ArgoUML. In particular this is the reason why syntax may differ from more usual UML syntax. For example ArgoUML represents transition labels as *Label: Event [Guard] / Action* instead of the more usual *Event [Guard] / Action.*

represent the state of the applet, a global `mode` *field is generated along with constants for the three different possible states. The invariant specifies that* `mode` *must be one of three values.*

```
/*@ invariant
  mode==LOCKED || mode==CHALLENGED || mode==NORMAL;
 */
```

Although ESC/Java does not support JML constraints, we generate one anyway. The constraint describes the transitions of the state diagram.

```
/*@ constraint
  (mode==LOCKED ==> \old(mode)==NORMAL || \old(mode)==LOCKED) &&
  (mode==CHALLENGED ==> \old(mode)==NORMAL) &&
  (mode==NORMAL ==> \old(mode)==CHALLENGED) &&
  (\old(mode)==LOCKED ==> mode == LOCKED) &&
  (\old(mode)==CHALLENGED ==> mode==NORMAL) &&
  (\old(mode)==NORMAL ==> mode==CHALLENGED || mode==LOCKED);
 */
```

A Java Card applet's process method deals with incoming instructions. It just calls dedicated sub-process methods for each possible instruction. Only one (of the two) generated sub-process methods is discussed here:

```
/*@
  requires true;
  modifies mode, counter;
  ensures
    \old(mode)==NORMAL && counter < 10000 ==> mode==CHALLENGED;
  exsures (ISOException)
    \old(mode)==LOCKED ==> mode==LOCKED;
  exsures (ISOException)
    \old(mode)==NORMAL && counter >= 10000 ==> mode==LOCKED;
 */
private void processRUN_GSM_ALGORITHM(APDU apdu)
throws ISOException {
  if (mode==LOCKED) {
    mode = LOCKED;
    ISOException.throwIt(SW_UNKNOWN);
  } else if (mode==NORMAL && counter < 10000) {
    counter = (short)(counter + 1);
    mode = CHALLENGED;
  } else if (mode==NORMAL && counter >= 10000) {
    mode = LOCKED;
    ISOException.throwIt(SW_UNKNOWN);
  }
}
```

The (normal and exceptional) post-conditions (indicated by ensures and exsures) specify the different transitions labelled with `RUN_GSM_ALGORITHM`*. The corresponding assignments are performed in the body of the method. The generated applet and specification, when run through ESC/Java, generate no warnings.*

Obviously, the generated applet does not fully implement the GSM authentication protocol. Some code has to be added manually: for this example mostly calls to the cryptography API (encrypting, comparing byte arrays). It may be possible to generate this code also automatically, and include it in the UML model by using the UML features more extensively. At the moment we have not specified any so-called 'entry-actions', 'exit-actions' or 'do-activities'. It seems possible to specify Java code there, or if not, tags that can be translated automatically into code. Or we could specify more code at the transitions than we do now. For the first solution we would have to modify our `fsm.dtd`, for the last one probably not.

What is specified in JML is mostly control flow. SIMs that support Java Card do exist, yet the authentication protocol is usually implemented as native code, not as a Java Card applet. The applets that can be downloaded onto a Java Card SIM, so-called Toolkit applets, bring additional services on top of the GSM functionality. Nevertheless, loading the generated applet onto a standard Java Card smart card results in a card that implements the authentication protocol as specified in GSM standard.

3 Implementation details

The diagram in Figure 1 presents AutoJML as a black box. This section reveals, in more detail, how it works. Therefore readers not interested in these details are advised to skip this section.

One of the important ideas behind this project is that we want to use existing tools as much as possible. Therefore AutoJML consists of a few programs, each taking care of a relatively small transformation. The architecture of AutoJML is sketched in Figure 3. At the core of AutoJML

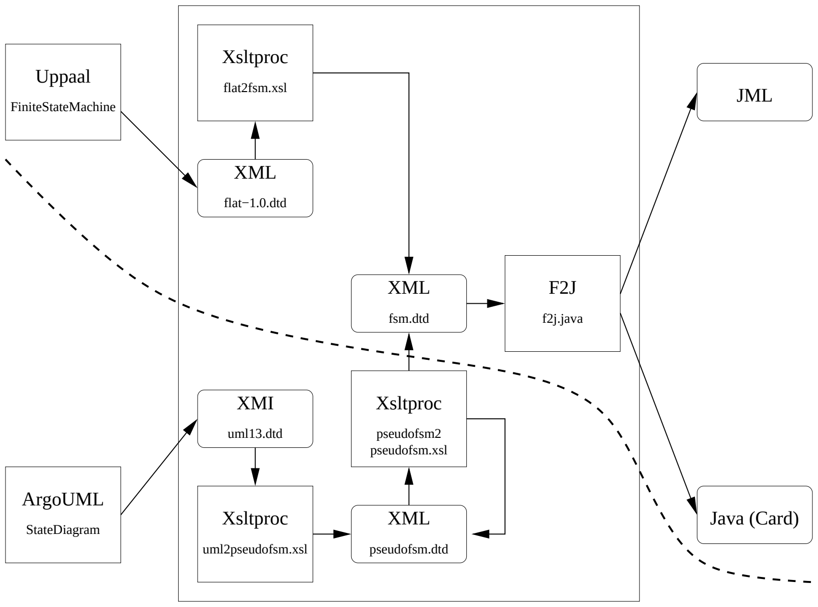

Figure 3: Detailed view of the AutoJML architecture

is a Java program called F2J, which generates the JML specifications and Java skeleton code based on an input XML document. F2J accepts XML input that conforms to the `fsm.dtd` document type definition (DTD). This is the current version of `fsm.dtd`.

```
<!ELEMENT fsm (name, initcode*, state+, init, transition*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT initcode (#PCDATA)>
   <!ATTLIST initcode language CDATA #REQUIRED>
<!ELEMENT state (name)>
   <!ATTLIST state id ID #REQUIRED>
<!ELEMENT init EMPTY>
   <!ATTLIST init ref IDREF #REQUIRED>
<!ELEMENT transition (source, target, name, (guard | assign)*)>
   <!ATTLIST transition type (nor|exc) #REQUIRED>
<!ELEMENT source EMPTY>
   <!ATTLIST source ref IDREF #REQUIRED>
<!ELEMENT target EMPTY>
```

```
    <!ATTLIST target ref IDREF #REQUIRED>
  <!ELEMENT guard (#PCDATA)>
    <!ATTLIST guard language CDATA #REQUIRED>
  <!ELEMENT assign (#PCDATA)>
    <!ATTLIST assign language CDATA #REQUIRED>
```

3.1 Previously: F2J project

The components above the dashed curve in Figure 3 form the part that has been described already in [HOP03a]. Apart from F2J, the part above the dashed curve uses the tools Uppaal and Xsltproc. The choice for Uppaal is based on its facility to export the created models into an XML file. Fortunately the one used by this tool, `flat-1.0.dtd`, is freely available. A simple stylesheet transforms Uppaal's output into `fsm.dtd` compliant XML. Perhaps it would have been possible to use stylesheets to generate JML and Java code, instead of the Java program F2J we use now. It seems that the stylesheets needed to perform F2J's task would be much more complex than the sheets we now use to transform XML files into `fsm.dtd` compliant XML files. However it might be interesting to find out whether this is really as difficult as we think.

Originally we were mainly interested in Java Card applets and since these applets all follow a very strict pattern, it is possible to generate real code. In principle we think that the generation of JML specification is more important than the generation of Java code.

In order to represent the different states listed in the F2J input, a special `mode` field is introduced into the generated code. All states in the XML document are translated into possible values of this field. This `mode` field implements a so-called life cycle model for the applet, similar to those in [MM01, Mos02]. In addition, all transition labels are translated into a list of constants representing valid instruction names for the applet. Furthermore, for each of these valid instruction names a dedicated sub-process method is generated. In Java Card the heart of each applet is always the `process` method. Each incoming instruction always enters the applet through the `process` method.

The constructor of the applet is generated so that it sets `mode` to the constant representing the initial state. The constraint describes the transitions of the state diagram. For each state and each incoming transition the possible source states are listed, and symmetrically for each state and each outgoing transition from that state the possible target states are listed.

The process method is generated to contain two nested switches. The outer one selects the state the applet is in (in the GSM example; `NORMAL`, `CHALLENGED`, or `LOCKED`). The inner switches select the type of instruction that is to be processed (in the GSM example: `RUN_GSM_ALGORITHM` or `GET_RESPONSE`). The process method just calls dedicated sub-process methods for each possible instruction.

The code generated for the body of such dedicated sub-process methods first checks the current state of the applet and the guards of the transitions starting in that state labelled with the instruction for which the method is specific. Depending on these checks, it executes the corresponding assignments and sets `mode` to the target state of that transition. If the transition is marked as exceptional, code is generated such that an exception is thrown resulting in a special error code at the terminal side. The JML specification generated for this method expresses some of this in the post-condition. Note that the level of verbosity of the JML specification is approximately the same as that of the code. Currently, the assignments information of the transitions is not used in the generation of the JML part.

3.2 Currently: AutoJML project

Although the diagram editor in Uppaal is easy to use, it is definitely not meant to be used only as a model editor. Moreover, it is not a standard industrial CASE modelling tool. Therefore we upgraded our AutoJML to also accept input from XMI compliant UML tools. The new parts of AutoJML are indicated in Figure 3 by the components below the dashed curve.

To test AutoJML we use XMI documents produced by ArgoUML, which provides XMI version 1.0 output based on UML-1.3. The XMI output of ArgoUML conforms to the `uml13.dtd` file that comes with the ArgoUML distribution. Unfortunately the transformation between `uml13.dtd` and `fsm.dtd` is not straightforward. The main problem is that `uml13.dtd` allows so-called pseudo-states which are not supported by F2J. In particular we have to take care of so-called joins and forks. The semantics we use for these two constructs basically comes down to the observation that we see them as graphical objects to create diagrams where parallel transitions are merged by taking the outer product of the incoming and outgoing transitions. Although the name of fork and join indicates something different, our system implies that there is no real difference between a fork and a join: both can be used in many-to-many situations. We return to the GSM example from Figure 2.

Example (ctd.) *As mentioned before this diagram contains three pseudo-states. The dot above the state* `NORMAL` *is an initial state, showing the entry point of this diagram. The bar between* `CHALLENGED` *and* `NORMAL` *is a fork. The bar below* `LOCKED` *is a join. The one outgoing transition from* `CHALLENGED` *to the fork should be seen as two outgoing transitions from* `CHALLENGED` *to* `NORMAL`*. Likewise for the join but then for incoming transitions.*

The main task for the transformation from `uml13.dtd` to `fsm.dtd` is the resolution of these pseudo-states into a list of transitions between true states. Fortunately, resolving one pseudo-state at a time is not that difficult: take a pseudo-state and construct the outer product of all incoming and outgoing transitions in this pseudo-state. Because of this observation we split the task into two transformations:

- `uml2pseudofsm.xsl` is used to strip all the information from the XMI document that we don't use anymore. Furthermore it makes sure that the initial pseudo-state is mapped onto its corresponding real state.

- `pseudofsm2pseudofsm.xsl` is the one that will be applied iteratively. Actually this `pseudofsm2pseudofsm.xsl` is split into two functions. Each odd time it is applied one pseudo-state is being resolved by constructing the outer product of all incoming and outgoing transitions in this pseudo-state. If there are no incoming or outgoing transitions to a pseudo-state we simply remove these pseudo-states. This typically happens for initial or end states. We only need the information on the initial state and we have already derived this in the `uml2pseudofsm.xsl` transformation.

  Unfortunately our resolving algorithm has as a side effect that we may introduce transitions whose source or target pseudo-state might no longer exist. It is also possible that several copies of the remaining pseudo-states are created. Both these transitions and pseudo-states get a special label. Therefore each even time the stylesheet is applied it basically only takes care of these labelled pseudo-states and transitions and makes sure that after this run both of these problems have been solved. Hence after each even application of this transformation stylesheet we know that the number of pseudo-states has been decreased by one. Therefore after precisely two times the number of pseudo-states in the original model all pseudo-states will have been resolved.

  The DTD `pseudofsm.dtd` is a superset of `fsm.dtd`. When this algorithm stops, the output validates against this `fsm.dtd`.

  Note that this resolution process only works if there are no transition cycles between pseudo-states without visiting a real state.

Example (ctd.) *Now if we apply our transformations to the GSM example we get the final result after seven stylesheet applications (one time* `uml2pseudofsm.xsl` *where the state* `NORMAL` *will be identified as the initial state and six times* `pseudofsm2pseudofsm.xsl`*). Here we show part of the final result.*

```
<fsm>
  <name>GSM</name>
  <state id="xmi.8">
    <name>NORMAL</name>
  </state>
  <state id="xmi.13">
    <name>CHALLENGED</name>
  </state>
  <state id="xmi.15">
    <name>LOCKED</name>
  </state>
  <init ref="xmi.8"/>
  <transition type="nor">
    <source ref="xmi.8"/>
    <target ref="xmi.13"/>
    <name>RUN_GSM_ALGORITHM</name>
    <guard language="uppaal">counter &lt; 10000</guard>
    <assign language="uppaal">counter := counter + 1</assign>
  </transition>
  <transition type="nor">
    <source ref="xmi.13"/>
    <target ref="xmi.8"/>
    <name>GET_RESPONSE</name>
    <guard language="uppaal">response_correct</guard>
  </transition>
  <transition type="exc">
    <source ref="xmi.13"/>
    <target ref="xmi.8"/>
    <name>GET_RESPONSE</name>
    <guard language="uppaal">response_incorrect</guard>
  </transition>
  ...
</fsm>
```

In order to distinguish between the transitions that model normal behaviour and the ones that model exceptional behaviour, we use a special suffix `_Exc` to model exceptional behaviour. The suffix `_Nor` can be used to model normal behaviour, but this suffix can be omitted since the normal behaviour is considered to be the default case. Furthermore, the order of the assignments and guards for the transitions that are merged by this transformation are kept in the same order. This is done because of possible side effects.

Example (ctd.) *Note that there are indeed two transitions from* `CHALLENGED` *to* `NORMAL`*. One of them is typed* `exc`*, the other* `nor`*, based on the labels* `GET_RESPONSE_Exc` *and* `GET_RESPONSE`*. Because the only assignments in this model were specified for a transition between the true states* `NORMAL` *and* `CHALLENGED`*, this example does not show that the stylesheets keep track of the order of guards and assignments during this resolution process.*

In order for this transformation to `fsm.dtd` to work there are a few constraints we put on the state diagrams created in ArgoUML.

- Transition cycles addressing only pseudo-states are not allowed.
- There should be only one `initial` pseudo-state and it should have exactly one outgoing transition. This transition is allowed to have effects, but it should not have guards.

- The pseudo-states `deepHistory`, `shallowHistory`, `branch`, `junction` and `final` should not be used since we didn't define semantics for these types.
- All true states should have a name.
- Transitions between true states should have a name.
- Transitions between pseudo-states and states do not need a name, as long as the resolved combinations of these transitions between true states do have a name.
- Names ending in `_Nor` and `_Exc` have a special meaning: these suffixes are used to indicate either normal or exceptional transitions. The string `_Nor` is optional.

4 Conclusion

The prototype tool AutoJML described in this paper generates JML specifications based on common UML state diagrams. Currently the UML model describes a Java Card applet, and some actual (skeleton) Java code is generated based on the state diagram. However, the ideas should be applicable to other domains as well. We have implemented AutoJML using established technologies (XMI, XSL, Java, JML) and tools (ArgoUML, ESC/Java). We have shown this method on a small yet realistic case study. Because of specific use of cryptographic functions our tool does not generate the complete Java Card implementation of the algorithm. One will have to add this specific code manually. However, AutoJML does generate valid Java code that is accepted by the Java compiler. Furthermore, the combination of skeleton code and JML specifications can be formally verified using ESC/Java.

The current state of AutoJML leaves quite some interesting topics that require further research. The first one is to shift the focus towards generating JML specifications without skeleton Java code. Full JML (not the limited subset accepted by ESC/Java) is powerful enough to express notions like state diagrams. Generating only JML allows programmers to make their own implementation choices. We see several possibilities here.

- AutoJML can generate a JML file which contains the method declarations for the important methods. All these declarations are annotated with pre- and postconditions. Furthermore the `mode` field indicating the state, should be defined as a model variable.
- An ESC/Java variant can be to use ghost variables instead of model variables.
- The specifications can be defined as an interface. The code written by the programmer will have to implement this interface.

We think that the first option is the most promising. However we will need to explore the appropriate JML features and no longer limit ourselves to using only ESC/Java.

The second interesting point is to find out what other kind of Java programs can be automatically specified using AutoJML. Certainly Java Card is not the only application domain. Perhaps a study and classification of design patterns as described in [GHJV94] is helpful in identifying situations where JML specifications can be automatically generated. We are studying Model Driven Architecture which may be a good technical solution to address multiple target platforms.

The third topic is application of AutoJML to security protocols. Typically security protocols are specified on a high abstract level. Finite state machines for the different principals in such protocols can be easily derived from this. In [HOP03b] we manually refine security protocols to Java code. Automation of these refinements seems interesting.

The last issue we want to look into is other CASE tools. At the moment our tool is based on XMI version 1.0 and UML version 1.3. However the OMG web page [OMG00] states that the current versions are respectively 1.2 and 1.5, and versions 2.0 are already announced to be nearly completed. The question arises what has changed in these standards. What has to be changed in our stylesheets to accept these newer standards? Furthermore, at the moment only ArgoUML has been used. We should try our machinery also on other UML modelling tools. Finally we would like to look at the available constructs in XMI/UML. Our current implementation relies on several assumptions listed in Section 3. Perhaps the size of this list can be decreased.

References

[BGW] M. Briceno, I. Goldberg, and D. Wagner. GSM cloning. Available from `http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html`.

[CF00] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66. ACM Press, January 2000.

[ECGN01] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEETSE*, 27(2):99–123, 2001.

[ETS01] Digital cellular telecommunications system (phase2+); specification of the Subscriber Identity Module - Mobile Equipment (SIM - ME) interface; 3GPP 11.11. Technical Report TS 100 977 V8.5.0, ETSI, March 2001. Available from `http://www.etsi.org/`.

[F$^+$02] C. Flanagan et al. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37 of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable Object-Oriented software.* Addison-Wesley, 1994.

[Hol97] G.J. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[HOP03a] E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct Java Card applets. In D. Gritzalis et al., editors, *Proceedings of the 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.

[HOP03b] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Proceedings of the 1st International Conference on Security in Pervasive Computing*, LNCS. Springer-Verlag, 2003. To appear.

[HRS02] D. Haneberg, W. Reif, and K. Stenzel. A method for secure smartcard applications. In *Proceedings of AMAST 2002*, volume 2422 of *LNCS*, pages 319–333. Springer-Verlag, 2002.

[J$^+$98] B. Jacobs et al. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 329–340. ACM Press, 1998.

[LBR99] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specifications for Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[LP99] J. Lilius and I. Porres Paltor. vUML: a Tool for Verifying UML Models. Technical Report 272, Turku Center for Computer Science, 1999.

[MM01] R. Marlet and D. Le Métayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 2001.

[Mos02] W. Mostowski. Rigorous development of JavaCard applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002.

[MPMU03] C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 2003. To appear.

[OMG00] OMG XML Metadata Interchange (XMI) Specification. Technical Report formal/00-06-01, Object Management Group, 2000. Available from `http://www.omg.org/`.

[R$^+$00] A. Ramirez et al. *ArgoUML User Manual: A tutorial and reference description of ArgoUML*. University of California, 2000. Available from `http://argouml.tigris.org/`.

[SKM01] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[Upp] Uppaal. An integrated tool environment for modeling, validation and verification of real-time system modeled as networks of timed automata, extended with data types. Available at `http://www.uppaal.com`.

[vL01] R. van Lierop. Model checking systems, described using uml activity diagrams, within asml. Third Dutch Model Checking Day, `http://www.win.tue.nl/dmcd/Presentation-Rick-van-Lierop.ppt`, 2001.