

Executing JML Specifications of Java Card Applications: A Case Study

Néstor Cataño
Dept. of Mathematics and Engineering
University of Madeira
Campus da Penteada, Funchal, Portugal
ncatano@uma.pt

Tim Wahls
Dept. of Mathematics and Computer Science
Dickinson College
P.O. Box 1773, Carlisle, PA 17013, USA
wahlst@dickinson.edu

ABSTRACT

Executability provides an important mechanism for validating formal specifications and allows such specifications to serve as prototypes and test oracles. In this case study, we used the `jmle` tool to execute the JML specifications of an electronic purse application written in the Java Card dialect of Java. This effort resulted in numerous improvements to the specification and to the `jmle` tool itself, as well as insight into how executability can contribute to the use of formal methods in the software development process.

Categories and Subject Descriptors

D.2.1 [Software Engineering]: Requirements/Specifications—*methodologies, tools*; D.2.4 [Software/Program Verification]: Validation

General Terms

Combined formal methods, case study

Keywords

Executable specifications, JML, Java Card

1. INTRODUCTION

Although formal methods have the potential to dramatically increase the quality of software systems, they have not been widely adopted in the software development industry. Many industrial users have the perception that formal methods are not cost effective and that the mathematical notations used are too complex for software engineers to assimilate. Executable formal specifications of software systems are one way to encourage the use of formal methods. Executable specifications can serve as prototypes and test oracles, and these uses are relatively easy to justify to clients and managers. Additionally, the ability to execute specifications makes developing them significantly easier and more natural for developers, as the correctness of the specification

can be tested in much the same way that the correctness of an ordinary program would be. In this work, we argue that executable specifications are also useful in conjunction with standard formal techniques such as static checking, and are a good way to find and remove many errors from specifications very early in the development process and in particular before applying heavier techniques such as program verification [26]. On the other hand, to avoid the problems noted in [13], executability must not overly compromise the level of abstraction used in writing specifications. That is, executable specifications must still specify only behavior (not implementation strategies etc.) so that they are still suitable for use in program verification, model checking and the like.

For this case study, we used the `jmle` tool [18] to execute the JML [20, 21] specifications of an electronic purse application written in the Java Card dialect of Java [16]. `jmle`, JML, Java Card and the electronic purse are briefly introduced in the following subsections. Our goals for the study included determining how well `jmle` works on the specification of a moderately large and complex application - what kinds of specifications can or can not be executed and why. We also examined the kinds of specification errors that can be found using executable specifications and the level of effort and expertise required to do so. As part of this examination, we compared our results using `jmle` with the results of applying three other tools to the same specifications.

1.1 The Java Modeling Language (JML)

JML is a behavioral interface specification language for Java, which means that the only correct implementation of a JML class specification is a Java class implementation with the specified behavior. JML was originally developed by Gary Leavens and his team at Iowa State University, but is now an academic community effort with many groups developing tools to support JML [11, 14, 17, 27]. In JML, methods are specified using `requires`, `modifies` and `ensures` clauses, which respectively give the precondition, the frame (what locations may change from the pre- to the poststate) and the postcondition. A method specification can also include an `exsures` or `signals` clause to specify conditions under which the method could throw an exception. The specification of a method appears immediately before its declaration. Class invariants can also be given to constrain the states of legal class instances. JML specifications use Java syntax and are embedded in Java code between special comments `/*@ ... */` or after `//@`. The reader is invited to consult [19] for a full introduction to JML.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'09 March 8-12, 2009, Honolulu, Hawaii, U.S.A.

Copyright 2009 ACM 978-1-60558-166-8/09/03 ...\$5.00.

1.2 jmle

jmle executes JML specifications by translating them to constraint programs. Because constraint programs can be written at much higher levels of abstraction than programs in a typical imperative or object-oriented language, this translation step imposes relatively little implementation bias on the kinds of specifications that can be translated. The translated specifications are executed using the Java Constraint Kit (JCK) [1], a tool for generating Java implementations of constraint solvers. As jmle itself is built on the Common JML Tools [5], all parts of the system are implemented in Java and so no external programming environments are needed to use jmle. The constraint programs generated are Java implementations of the specifications being translated, and can be called from ordinary Java code.

jmle translates each method specification to a method implementation that generates a set of constraints (a constraint store) from each of the pre-conditions of the specification. If none of the pre-conditions are satisfied (*i.e.* no pre-condition allows the corresponding constraint store to be simplified to the empty store without becoming unsatisfiable), an exception is thrown to indicate this error. Otherwise, for each satisfied pre-condition, the constraints generated from the corresponding post-condition are added to the store and solving resumes. If this store is satisfiable and can be simplified to the empty store, the method return value and any other post-state values specified are extracted from the logic variables that were used during constraint simplification. If the constraint store corresponding to a pre- or post-condition cannot be simplified to the empty store (but also can not be determined to be unsatisfiable), an exception is thrown to indicate that the specification could not be executed.

1.3 Outline of the Electronic Purse

The electronic purse is a Smart Card application, published as an advanced Smart Card programming case study by Gemplus [3]. Smart Card applications are written in Java Card [16], which is a dialect of Java designed to run in extremely resource-limited environments. Any Java Card application consists of two parts: a terminal side, implementing configuration and communication functionalities, and a card side implementing the Java Card application itself.

On the card side, the electronic purse provides the card holder with banking functionalities such as credit, debit and currency change. The card side of the purse contains three kind of applets: the loyalty applets, the card issuer applet and the purse applet, which communicate with each other by means of shared interfaces, the standard mechanism of communication between applets.

The purse implementation consists of three packages: `utils`, `purse` and `pacapinterfaces`. The `utils` package implements basic classes such as `Decimal` for representing money. The `pacapinterfaces` package declares the purse shareable interfaces. The `purse` package is the core of the purse application. It contains the class `PurseApplet`, which manages all the operations related to installation, selection and de-selection of the applet. This package also contains the class `Purse`, which keeps track of the balance of the purse, the transactions done by the purse (stored in a `TransactionRecord`), the different currency changes that have taken place and the different loyalty programs that the card holder is subscribed to (in `LoyaltiesTable`). Elements in the `TransactionRecord` are of type `Transaction`. Certain operations

can only be performed by a restricted set of users. The class `AccessCondition` defines the different access conditions. The class `AccessControl` binds different access conditions to operations. The class `AccessControlTable` declares the access controls the purse uses.

2. EXECUTING THE SPECIFICATIONS

As the electronic purse specifications were written in the specification language of the ESC/Java tool [10], which is slightly different than JML, they required a number of minor modifications. Following these changes, the specifications of 31 Java classes and interfaces were chosen for execution. These included all 28 of the classes and interfaces of the electronic purse specifications that had methods with non-trivial specifications, *i.e.* `ensures` clauses other than `ensures true;`, and three Java Card utility classes that are heavily used by the electronic purse specifications. During execution, five of these classes were found to contain erroneous invariants or field initializations, and these errors were corrected. The classes and interfaces contain 259 method specifications. Of these, 182 were executed with no changes other than those noted previously. An additional 27 method specifications were found to contain errors, and were executable after these errors were corrected. The specifications of 22 underspecified methods were strengthened to permit execution. Typically, these specifications did not constrain the object returned by the method other than asserting that it was not `null`. Another 17 methods with missing or trivial specifications were not executed. Finally, 11 method specifications were modified solely to permit execution. Examples of the kinds of errors found and corrected, and how underspecified method specifications were strengthened, are given in the following subsections. Specifications were executed by writing JUnit [23] unit tests for each method specification. These unit tests allowed regression testing as the specifications (and jmle) were modified, and could also be used to directly test the correctness of the method implementations. At least one unit test was written for each specification case of each method specification, and condition coverage was achieved for many of the specifications.

2.1 Specification Errors

Many different kinds of specification errors were found and corrected, including:

- missing or incomplete `modifies` clauses
- `modifies` clauses that reference objects that do not exist in the pre-state of the method
- array indexing errors
- incorrect redeclaration of model (specification-only) fields in subclasses
- references to post-state values in cases where pre-state values were needed

Some of the most interesting errors occurred in the `Decimal` class, which represents fixed-precision quantities such as monetary amounts. To reduce memory usage, Java Card does not include any floating point types or any ordinal types larger than `short`. Hence, the `Decimal` class has two fields `intPart` and `decPart` of type `short` representing the numbers before and after the decimal point. The class invariant

```

/*@ requires d != null;
   modifies intPart, decPart;
   ensures (intPart * PRECISION + decPart)
           * PRECISION ==
           \old(intPart * PRECISION + decPart) *
           (d.intPart * PRECISION + d.decPart);
   ensures \result == this;
   exsures (DecimalException) intPart < 0; */
public Decimal mul(Decimal d) throws DecimalException

```

Figure 1: Specification of the `mul` method of class `Decimal`.

states that the `intPart` must be nonnegative and that the `decPart` can only take values from 0 to `PRECISION` (a constant defined as 1000), limiting instances to 3 digits of precision. In addition to the kinds of errors mentioned above (for example, the `setValue` methods did not specify that `intPart` and `decPart` were modifiable), the class invariant was in a standard Java comment and not a JML `/*@ ... */` style comment. Hence, the invariant would be ignored by JML tools. Additionally, the specification of the `sub` method (for subtracting two `Decimals`) contained an arithmetic precedence error that caused it to return incorrect results. Finally, the specification of the `mul` method (for multiplying two `Decimals`) contained several errors, as shown in Figure 1. In this specification, the entire expression giving the post-state value of the `Decimal` is too large by a factor of `PRECISION`, the part after the decimal point is not truncated after the thousandths place (so that the conjunction of the `ensures` clause and the invariant is not satisfied whenever the result of the multiplication has more than three decimal places), and the exception specified by the `exsures` clause can never be thrown because the invariant specifies that `intPart` must be nonnegative (so that the conjunction of the `exsures` clause and invariant is a contradiction). The author who wrote the test cases did not discover the first two errors when reading the specification - they were found only when the unit tests for the specification failed.

All of the electronic purse specifications had previously been analyzed with the ESC/Java tool [6], although the primary intent of that effort was to check the correctness of the code in the method implementations and not the specifications themselves. This does indicate the importance of applying multiple tools, as each is likely to find different kinds of errors and weaknesses. All of these classes except for the `Transaction` class had also been analyzed with the Chase tool [7], which uses syntactic analysis to check `modifies` clauses. Chase found 43 missing or incorrect `modifies` clauses. Of these, our use of `jmle` independently rediscovered only 5 (included in the 27 erroneous method specifications previously mentioned). Of the remaining errors, 8 could be found only by analyzing method implementations (which `jmle` does not do), and the rest could potentially have been revealed by more thorough unit testing. Our use of `jmle` also found 3 missing `modifies` clauses in the `Transaction` class. `jmle` also found numerous cases in which locations that did not exist in the pre-state were listed in `modifies` clauses, which Chase does not check. A modified version of the `Decimal` class had also previously been verified with the LOOP tool [4]. The modified `Decimal` class used does correct the errors noted in the previous paragraph, but it is significant that executing the specification indepen-

```

/*@ requires true;
   modifies \nothing;
   ensures (\forallall int j; (j >= 0 && j < data.length)
           ==> data[j].methode != id) ==> \result == null;
   ensures (\exists int j; (j >= 0 && j < data.length)
           && data[j].methode == id) ==>
           \result != null && \result.methode == id;
   exsures (java.lang.Exception) false; */
public AccessControl getAccessControl(byte id)

```

Figure 2: Specification of the `getAccessControl` method of class `AccessControlTable`.

dently rediscovered these errors. The author who wrote the test cases for and corrected the original `Decimal` class was not aware of the modified version at the time that the work was done. Although the executable specification approach is incomplete and does not verify the correctness of the implementation, it requires considerably less effort and produces tests that can be run directly against the implementation.

2.2 Underspecification

As an example of a method specification that required strengthening for executability, consider the specification of the `getAccessControl` method from the `AccessControlTable` class in Figure 2. This method finds and returns the `AccessControl` object with the specified `id`, or `null` if no such `id` exists in the table. The JML keywords `\forallall` and `\exists` provide the standard universal and existential quantifiers. The specification states that if an `AccessControl` with the specified `id` exists in the table, the object returned is not `null` and has the specified `id`. In particular, the other fields of the object returned are completely unspecified. This specification was strengthened to state that the `AccessControl` object returned in the second `ensures` clause is the array element with the specified `id`.

2.3 Unexecutable Specifications

As a final example, the specification of the constructor for the `LoyaltiesTable` class, which keeps track of merchant information for customer loyalty programs, could not be executed as given in Figure 3. The `\fresh` keyword in JML specifies that the indicated objects are freshly allocated (did not exist in the pre-state), and `assignable` is a synonym for `modifies`. The `data` field is an array of `AllowedLoyalty` objects, and each `AllowedLoyalty` object contains an array field (also called `data`) which stores merchant identifiers. The invariant for the `AllowedLoyalty` class specifies that the elements of the `data` field can not be `null`. Since this is not implied by the freshness of the `AllowedLoyalty` objects themselves, the specification could not be executed without asserting that the elements of the `data` fields of the `AllowedLoyalty` objects were also fresh.

2.4 Improvements to `jmle`

In addition to the improvements to the electronic purse specifications, this case study resulted in valuable improvements to `jmle` itself. Many errors in the implementation of `jmle` were exposed by these specifications, and the specifications motivated the addition of some significant new features. The specifications in the `Decimal` class are most naturally executed using finite domain constraints (in which variables draw their values from finite sets of integers), and

```

/*@ requires true;
   assignable data[*];
   ensures \fresh(this);
   ensures (\forall int k; (k >= 0 && k < NB_MAX) ==>
            \fresh(data[k]));
   ensures nbLoyalties == 0;
   exsures (java.lang.Exception) false; */
LoyaltiesTable()

```

Figure 3: Specification of the constructor for class LoyaltiesTable.

so a solver for finite domain constraints was added to jmle. The specifications of many of the other classes use quantifier syntax that was not originally supported by jmle. The ability to treat method calls in specifications as constraints (*i.e.* to pass logic variables as parameters in calls to method specifications, rather than requiring that all parameters be ground) was a crucial addition for executing a number of specifications. Finally, the electronic purse and Java Card specifications also motivated improved handling of numerous other Java and JML features, including bit level operators, `\fresh` expressions, `\old` variables and field initializations.

3. RELATED WORK

A number of other researchers have created systems for translating formal specifications to constraint programs [12, 22] or for executing specifications using SAT solvers [15]. Of particular interest here is the `jml-tt` tool [2], which animates JML specifications. However, these systems use an external interface to animate specifications – they do not translate specifications to implementations that can be called directly from code. Additionally, `jml-tt` does not support animating specifications that use JML’s built-in classes. We are not aware of any studies on applying these systems to the specifications of sizable software systems.

In [9], J.-L. Lanet *et al.* report on a testing case study applied to a simplified version of a JML-specified Java bank application dealing with money transfer. The main goal of their work is to find inconsistencies between the JML specifications and the code, and between these and the informal requirements of the application. Unit tests are automatically generated by using the Jartege tool [25]. Our work on executing JML specifications can be thought of as part of a software engineering methodology that envisions the subsequent use of testing for validating the specifications against a particular implementation. Applying different tools at different stages of software development makes it possible to find different kinds of specification and implementation errors, and to alleviate the work carried out by other tools.

The Foundations of Software Engineering group at Microsoft Research has applied executable Abstract State Machine (ASM) specifications to the development of the Universal Plug and Play product group [24]. The authors outline a methodology for employing executable specifications at numerous points in the development process and use a case study to illustrate the advantages of this approach. Many of the benefits are similar to those that we have described. One significant difference from our work is the specification language used, as ASMs are best suited for describing how a running system responds to stimuli, while JML is designed for specifying the interfaces of software compo-

nents. Additionally, the ASM notation used in this work is much more procedural than JML.

4. CONCLUSION

Using `jmle` to execute the electronic purse specifications resulted in significant improvements to both the specifications and the `jmle` tool itself, and produced a comprehensive unit test suite that can be used to test both the specifications and the implementation. However, this effort did require a larger time investment than would be typical for applying a “lightweight” tool (that does not require human interaction while executing). Some of this time was spent debugging and adding features to `jmle`, but in numerous cases considerable effort was required to find the specification error that caused a unit test to fail. More detailed error messages would help with this problem - the runtime errors currently produced by `jmle` indicate only that the failure was in evaluating the `requires`, `modifies` or `ensures` clause. However, the implementation of `jmle` makes giving more detailed error messages difficult. It is trivial to report which constraint was being processed at the time that the constraint store was found to be unsatisfiable, but it is likely that the failure was caused by a constraint that had been solved earlier. Additionally, many intermediate constraints are produced as the store is solved, so the correspondence between constraints and parts of the specification is often indirect.

Although the unit test suites for most of the class specifications run fairly quickly (in a few minutes or less on a moderately fast PC running Linux), several require more time. The longest running test suite (for the `LoyaltiesTable` class) takes just over 3 days to execute. This is largely attributable to the number of post-state values that must be found, and the size of the constraint store that is generated. Each post-state value corresponds to a logic variable, and the running time of constraint programs is well-known to be exponential in the number of variables in the worst case. Each instance of the `LoyaltiesTable` class contains an array of `AllowedLoyalty` objects, each `AllowedLoyalty` object contains an array of merchant identifiers (instances of class `SalerID`), and each `SalerID` contains a `byte` array identifying the merchant. As the value of each element of these `byte` arrays is specified by the post-condition of many of the specifications in class `LoyaltiesTable`, the number of logic variables that must be solved for is formidable.

Even with these limitations, we have found executability (and lightweight tools in general) to be effective for analyzing specifications. Using such tools is markedly simpler and less time consuming than performing full verification, and can identify a wide range of specification and implementation errors. These benefits are compounded when a range of lightweight tools (including static and runtime checkers) are applied, as each tool will likely find different errors and weaknesses. Even when full verification is to be undertaken, using lightweight tools first can reduce the total effort required by allowing many specification problems to be found and corrected before verification begins. Writing unit tests for executable specifications should be relatively natural for developers, and has the additional advantage of producing a unit test suite for the implementation.

One important area for future work is improving the efficiency of both the constraint solvers for JML types and of the implementation of JCK itself. Excessive running times are certain to discourage developers from using exe-

cutable specifications. Given the use of constraint programming techniques, some specifications will always be expensive to execute, but improvements are possible and would help in many cases. Another aspect of jmle that requires additional attention is the handling of `\fresh` expressions and `modifies` clauses, as these were the constructs that forced changes to all 11 of the method specifications that were modified for executability. We are also investigating ways of incorporating jmle into an integrated development environment that would include tools for verification and for static and runtime checking, such as the JML4 Integrated Verification Environment [8]. Executable specifications will be most effective when combined with other tools for analyzing JML specifications, and an integrated environment is the most effective way to support this usage.

5. REFERENCES

- [1] S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java constraint kit. In M. Hanus, editor, *Electronic Notes in Theoretical Computer Science*, volume 64. Elsevier, 2002.
- [2] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. In *Proceedings of the International Conference on Formal Methods 2005 (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 75 – 90. Springer-Verlag, July 2005.
- [3] E. Bretagne, A. E. Marouani, P. Girard, and J.-L. Lanet. PACAP purse and loyalty specification. Technical Report V. 0.4, Gemplus, 2000.
- [4] C. Breunese, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for Smart Cards: an experience report. *Science of Computer Programming*, Issues 1–3, 51:55–80, March 2004.
- [5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
- [6] N. Cataño and M. Huisman. Formal specification of GEMPLUS' electronic purse case study. In L.-H. Eriksson and P. A. Lindsay, editors, *FME: Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 272–289, Copenhagen, Denmark, July 22-24 2002. Springer.
- [7] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 26–40, London, UK, 2003. Springer-Verlag.
- [8] P. Chalin, P. R. James, and G. Karabotsos. An integrated verification environment for JML: architecture and early results. In *SAVCBS '07: Proceedings of the 2007 Conference on Specification and Verification of Component-based Systems*, pages 47–53, New York, NY, USA, 2007. ACM.
- [9] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in JML-based software validation. In *The 19th IEEE International Conference on Automated Software Engineering (ASE 04)*, pages 294–297, September 2004.
- [10] Extended Static Checking for ESC/Java. <http://www.research.compaq.com/SRC/esc/Esc.html>.
- [11] The ESC/Java 2 Tool. <http://secure.ucd.ie/products/opensource/ESCJava2/>.
- [12] W. Grieskamp. A computation model for Z based on concurrent constraint resolution. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, *ZB 2000: Formal Specification and Development in Z and B, First International Conference of Z and B Users*, volume 1878 of *Lecture Notes in Computer Science*, pages 414 – 432, York, UK, September 2000. Springer-Verlag.
- [13] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *IEEE Software Engineering Journal*, 4(6):320–338, November 1989.
- [14] The JACK Tool. <http://www-sop.inria.fr/eve-rest/soft/Jack/jack.html>.
- [15] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Massachusetts Institute of Technology, 2006.
- [16] Java Card Technology. <http://java.sun.com/products/javacard/>.
- [17] The Krakatoa Tool. <http://krakatoa.lri.fr/>.
- [18] B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, editor, *Formal Methods for Industrial Critical Systems (FMICS '06)*, volume 4346 of *Lecture Notes in Computer Science*, pages 293 – 296. Springer-Verlag, August 2006.
- [19] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual. <http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman.toc.html>.
- [20] G. T. Leavens. The Java Modeling Language (JML) home page. <http://www.jmlspecs.org>.
- [21] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, Oct. 2000.
- [22] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
- [23] J. Link. *Unit Testing in Java*. Morgan Kaufmann, 2003.
- [24] Microsoft Research. Executable specifications: Creating testable, enforceable designs, February 2001. <http://research.microsoft.com/foundations/ESpecTesting.doc>.
- [25] C. Oriat. Jartege, A Tool for Random Generation of Unit Tests for Java Classes. Technical Report RR1069, 2004.
- [26] A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*. MIT Press, 2001.
- [27] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of TACAS*, number 2031 in LNCS, pages 299–312. Springer, 2001.