A Thought on Specification Reflection

Yoonsik Cheon, Yoshiki Hayashi, and Gary T. Leavens

TR #03-16
December 2003

Keywords: Specification reflection, specification introspection, reflective specification execution, specification object, specification class object, JML language.
2001 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, JML; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, class invariants, formal methods, programming by contract; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, pre- and post-conditions, specification techniques.

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Thought on Specification Reflection

Yoonsik Cheon and Yoshiki Hayashi
Department of Computer Science, University of Texas at El Paso
cheon@cs.utep.edu

Gary T. Leavens
Department of Computer Science, Iowa State University
leavens@cs.iastateedu

December 10, 2003

Abstract

In programming languages, reflection is the ability to discover and manipulate, at runtime, information about program entities, such as objects. We present our thoughts on extending the concept of reflection to behavioral interface specifications. We explain the benefits of such *specification reflection*, and discuss implementation approaches, support tools, and research problems and issues in this area.

1 Introduction

Reflection is becoming more widely used in practice, as witnessed by its adoption in the very popular object-oriented programming languages Java [1] and C# [21]. *Reflection* refers to the ability to manipulate programming language constructs as runtime data, e.g., by representing, or reifying, them in the language itself [10]. In object-oriented programming languages such as Java, this is most often done by reifying information about objects as runtime class objects or *meta objects*, and reifying information about classes as runtime *meta class objects*, etc. These meta objects and meta class class objects are accessed and manipulated through the reflective application programming interfaces (APIs), often called Meta-object Protocols (MOPs) [4, 12, 13]. By using MOPs, for instance, a debugger may access the execution stack of a program under debugging. Reflection has been studied by many researchers, in particular, in the area of programming languages. However, little work is found extending the concept of reflection to specification languages.

Formal behavioral interface specification languages, such as JML [14, 15], allow one to specify both the syntactic interface and behavior of Java program modules, such as classes and interfaces. JML does this using preconditions and postconditions for methods, as well as other features, such as class invariants. Interface specification languages such as JML provide a wide variety of support tools to manipulate specifications [3, 6, 5, 7]. However, JML's tools do not make specifications available at runtime through well-defined APIs, such as those of MOPs.

In this paper we present our thoughts on the research challenges involved in extending the concept of reflection to formal behavioral interface specification languages, and to JML in particular. We call the idea of supporting reflection on behavioral interface specifications *specification reflection*. The overall goal of specification reflection is to reify specifications as first-class objects [9] in the sense that they can be accessed and manipulated at runtime just like regular objects. For example, one may query an object about its specifications, to discover the pre- or postconditions of its methods or its invariants. The specifications discovered may be used for both runtime assertion checking or as documentation for human readers. The major engineering challenge is to support both uses with a single representation.

Specification reflection may have an impact not just on tools, but also on the semantics of specification languages. This is because, in a language, like JML or Eiffel [17], that uses expressions in its assertions, it is possible to write code that uses specification reflection inside assertions. However, we ignore this possibility in what follows.

The rest of this paper is organized as follows. In the next section we first define what we mean by specification reflection. In Section 3, we describe a simple specification reflection model, suitable for JML. Section 4 explains the benefits of specification reflection,

while Section 5 discusses some research issues. In Section 6 we briefly describe our research plan, and then we conclude in Section 7.

2 Specification Reflection

What precisely do we mean by "specification reflection"? Like code reflection in programming languages[1], we use the term specification reflection very generally to mean the ability to manipulate specifications as runtime data. The nature of specification manipulation may be introspective or may involve more than just introspection.

Specification introspection is the general term for discovering information related to specifications at runtime. Like code introspection [2] this can be used to discover descriptive information; for example, one could use specification introspection to discover the pre- or postconditions of methods. A more interesting facet of specification introspection is the ability to observe a program's specification state. The specification state of a program can involve specification-only fields of objects, such as JML's model and ghost fields, which are abstractions of the program's normal (code) fields [8].

Reflective specification execution is the general term for execution of specifications discovered through introspection. Reflective specification execution allows one to program a design by contract tool [18, 19], like a runtime assertion checker [5, 20], or to customize it.

Both kinds of specification reflection are useful. For example, using specification introspection one can build a system that can retrieve formal specifications of an object's methods in a debugger. Reflective specification execution brings even more benefits, but also carries with it several challenges. We discuss these challenges in the following sections.

Yet another possibility in manipulating specifications is to change or modify the specifications themselves at runtime. Such a capability might be useful, e.g., to dynamically debug specifications at runtime. However, we will not discuss this aspect further in this paper.

3 A Specification Reflection Model

To make our ideas more concrete, we now explore a basic plan for implementing specification reflection in JML.

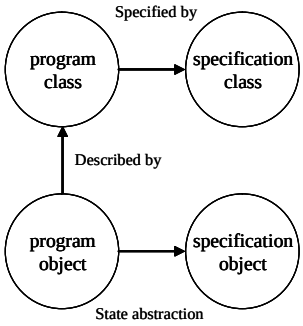

Figure 1: A specification reflection model

The information needed to support specification reflection must be present in the bytecode produced by the JML compiler (`jmlc`). This will allow it to be present at run-time. One way to do this might be to put the information needed into attributes in the bytecode file. However, to load the information at runtime then requires a special class loader. Another possibility is to add the information needed into the classes and interfaces that would be loaded in any case — for example, as extra fields, methods, or nested classes or interfaces. In this case, naming conventions would probably be necessary to prevent the names of such added members from conflicting with programmer-defined names.

In our basic model of specification reflection, as shown in Figure 1, for each code (or normal) object there may be a separate object, called a *specification object*, that represents the model and ghost fields of the code object. (The model and ghost fields are, as mentioned above, abstractions of the code object that are used in specifications.) Each code class object is also associated with a *specification class object* that represents the specifications written for that class. The specification class object contains such information as the class's invariant and pre- and postconditions of methods declared by the class. The specification class object provides introspection on the object's specification, in a similar fashion as the class object of an object provides introspection on the object. That is, in addition to the meta-object protocol, a program object is now provided with a meta-specification protocol to access the specifications and specification state represented by its specification objects. A particular specification may be queried and checked based on the program and specification states.

We expect to provide at least the following kinds of APIs for specification reflection.

[1] We often use the term "code reflection" to distinguish it from specification reflection.

- Retrieving specification objects and classes. The specification state of an object should be accessible from the object (e.g., `getSpecObject()`). In our reflection model, a specification state is represented by a specification object. In addition, a specification class should be also available from both an object and the object's class (e.g., `getSpecClass()`).

- Querying specification states. It should be possible to query the specification state represented by a specification object, e.g., to get the value of a specification-only field. A similar reflection facility as referring to a program field in object reflection may be needed for this.

- Querying about specifications. It should be possible to query about specifications on a specification class, e.g., retrieving pre- and postconditions of a method, and the invariant of a class or an interface (e.g., `getPrecondition()`, `getPostcondition()`, and `getInvariant()`).

- Executing specifications. It should be possible to execute a retrieved specification, such as a pre- or postcondition. A similar reflection facility, such as that used to invoke a program method in code reflection may be needed for this.

4 Benefits

The followings are some important benefits of specification reflection that cannot be achieved without it.

- Runtime access to specifications allows programs that understand specifications to make decisions about them. For example, multiagent software could use specification introspection to communicate details of a protocol's semantics between independently-developed agents; thus allowing them to decide if they can sensibly communicate.

- Customization of runtime assertion checking. Using reflective specification execution, one could write a customized assertion checker, that, for example, only executed assertion checks when certain conditions are true (e.g., the 10th time through a loop).

- Support for more open development of specification-based tools using the specification reflection APIs. These APIs can be seen as a development framework for tools that use specifications. Such tools, including runtime assertion checkers, browsers, and testing tools, are hard to develop from scratch. In addition, differences in parsing and representations make it difficult to share code between tools.

- Easier shipment and location of specifications by users. If the specifications are in the compiled outputs (the bytecode files in Java), then they are easy to send and locate. They can even accompany code for which sources are not provided.

5 Research Issues

In addition to its benefits, specification reflection also poses several challenges for research, which we discuss below.

5.1 Reflection for Whom?

The key research issue that we see is how to support both human readers in specification introspection and reflective specification execution. The engineering problem here is that the representation that is most convenient for execution, namely code to evaluate assertions directly, is difficult to translate into human-readable form. Conversely, if a form, such as an abstract syntax tree (AST) that is closer to the specification language's syntax is stored, then there may be considerable difficulty and expense in executing the specification.

While it is easy to see the benefits of supporting reflective specification execution, supporting human readers is also highly desirable. For example, this allows humans to browse specifications directly from bytecode. This would be particularly useful in a distributed environment where one can dynamically retrieve a reusable binary component; its specification comes automatically with the component. The approach may also have some advantages in terms of maintenance, for both specifications and bytecode are maintained in the same place and at the same time. We can imagine a wide variety of tools that take benefits of the human readable specification reflection.

The trick is thus to find a representation that is amenable to both kinds of readers. We can use some analogies with code reflection support for some ideas. For example, bytecodes typically have associated "symbol table" information that can be used to aid debuggers and provide information that is not normally present in bytecode (such as variable names). Another idea is "just in time" compilation, where specifications stored as ASTs and symbol table information might be compiled only when

needed. It might also be possible, by standardizing the compilation process, to back-translate bytecodes that are designed for executing specifications into human-readable form. Or some blend of such representations might be used. This is certainly an engineering challenge.

5.2 Specification States and Inheritance

In our reflection model, a program object may be associated with special objects called specification objects (see Section 3). The specification objects represent the specification state of the object; this includes model fields and ghost fields for the object. Such specification-only fields are used to write more abstract assertions in JML[2]. The question here is how we model and represent such specification states in the presence of specification inheritance?

In JML, a subtype inherits specifications from its supertypes[3]. In JML a class inherits pre- and postconditions, invariants, etc. from both the superclasses that it extends and the interfaces that it implements. An interface also inherits specifications from its superinterfaces that it extends. A supertype may introduce a specification state that must be inherited by a subtype, e.g., by declaring a model field. Such a specification state may be referred to by a subtype's specifications.

In programming languages such as Java, an instance of a subclass has memory slots not only for fields declared in the subclass but also for those declared in its superclasses. Thus, the state of a class instance is encapsulated in a single object, regardless of where the fields are declared. A similar approach may be employed to represent the specification state. An object's specification state may be encapsulated in a single specification object regardless of where the specification-only fields are declared. In this scheme the state of a specification object consists of all specification-only fields declared either in the corresponding object's class or in any of its supertypes. This approach, though conceptually simple, is not modular and might require a challenge for reflective specification execution due to problems such as name crashes (see Section 5.3).

Another approach is to distribute the specification state into several specification objects, e.g., one for each class or interface [5]. In this approach, an object may be associated with several specification objects, one directly and the rest indirectly. The idea is to introduce a separate specification object for each class and interface. An object is directly associated with single specification object that represent the specification state explicitly introduced by the object's class. However, the specification object is, in general, composed of other specification objects, one for each supertype of the class. Thus, the specification object graph is the same as inheritance hierarchy. This approach is modular and does not incur such problems as name crashes, but the system has to maintain specification object graphs for specification execution.

5.3 Inheritance of Specifications

In JML there are some difference between inheritance of code and specifications. In code inheritance, a subtype's method overrides (or replaces) its supertype's methods of the same signature. However, in JML's specification inheritance, a subtype and its supertype's specifications are conjoined in a way that forces behavioral subtyping [11, 16]. That is, a subtype's methods must satisfy the specifications of all methods it overrides.

How should JML's specification inheritance be supported by specification reflection? For example, evaluating a method's precondition means evaluating not only the method's precondition specified in the class where the method is declared, but also all the inherited preconditions, such as those inherited from the class's superclasses or the interfaces that the class implements. A simple monolithic approach would be to statically combine all such specifications at compile time and turn it into an assertion checking code or method. However, this approach is not modular and increases the size of bytecode, as the assertion checking code for a supertype's specification is repeated in the bytecode of every subtype. A better and modular approach would be to instrument each specification into a separate assertion checking code or method and let the subtype's bytecode call its supertype's [5]. On the other hand, during introspection, one may wish to know what part of a specification is inherited and what part is written into the class.

5.4 Evaluating Specifications that Refer to Old States

We expect that exposing the capability of checking specification assertions (such as pre- and postconditions and invariants) through reflective specification execution will greatly improve the usability of specification reflection, e.g., to such specification-based

[2] A specification may also refer to program fields (e.g., DBC specifications written in terms of program variables), thus both the object itself and its specification objects are needed to evaluate a specification.

[3] We use the term type to denote both classes and interfaces.

tools as debugging tools and testing tools. These tools can be built on top of a specification reflection facility.

An interesting issue in reflective specification execution is how to provide reflective access to previous code and specification states. In JML, as in Eiffel, postconditions can refer to both pre-state and a post-state. The pre-state value of an expression e can be referred to using the notation `\old(`e`)`. The problem is how to provide access to such pre-state values, so that post-conditions (for example) can be executed? Ideally, such a pre-state should be automatically provided by the specification reflection system. But how was the reflection system to know that it needed to save the pre-state value at an earlier time? Doing this without imposing large overheads is another engineering challenge.

5.5 Support Tools

A wide variety of tools are possible that enable and use specification reflection.

The most fundamental tool for enabling specification reflection is a compiler that translates programs annotated with JML specifications into bytecode that supports specification reflection. Also needed is a runtime library that can interpret the information the compiler puts in the bytecodes.

Once such an enabling compiler is built, other tools such as runtime assertion checkers, debugging tools, and testing tools can be built using the specification reflection facility. The research hypothesis is that building such tools has advantages over building them without specification reflection. The challenge is to understand these advantages and to demonstrate them convincingly.

6 Our Plan

We are currently investigating our idea on specification reflection by using Java and JML. Our plan is to pick a subset of JML, and define a specification reflection model and specification meta protocols. Once our model is refined, we will modify and extend the current JML tools such as JML compiler (`jmlc`) to implement our model and the specification reflection facility for Java and JML.

As a part of our research effort on specification reflection, we are currently implementing a scheme to encode some aspects of JML specifications into Java bytecode file. It will be initially used to support modular typechecking of JML specifications, but later it will be adapted and extended to support specification reflection.

The JML compiler (`jmlc`) already implements a limited form of specification reification in that specification states are represented as a part of the corresponding object or as separate specification objects, and specification assertions such pre- and postconditions are implemented as separate assertion checking methods. However, no APIs for specification reflection (i.e., specification meta protocols) are provided.

7 Summary

We believe that the ability to manipulate specifications as runtime data, that we call specification reflection, bring many benefits by extending the notion of code reflection from programming languages to specification languages. We proposed a simple specification reflection model that would be suitable for a behavioral interface specification language like JML. We then explained some of the benefits of using specification reflection and discussed research problems and issues.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[2] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in context — the shape of the design space. In Andreas Paepcke, editor, *Object-Oriented Programming: the CLOS Perspective*, chapter 2. MIT Press, Cambridge, Mass., 1993.

[3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[4] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple security-aware MOP for Java. In A. Yonezawa and S. Matsuoka, editors, *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 118–125. Springer-Verlag, Berlin, 2001.

[5] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[6] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[7] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[8] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10a, Department of Computer Science, Iowa State University, September 2003. Available from `archives.cs.iastate.edu`.

[9] Philippe Collet and Roger Rousseau. Assertions are objects too! In *White Object Oriented Nights (WOON '96), Proceedings, St-Petersburg, Russia, June 20-21, 1996*, June 1996.

[10] Francois Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: A short comparative study. University of Montreal, Montreal, Canada, 1995.

[11] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[12] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1), January 2001.

[13] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Mass., 1991.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[15] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. See `www.jmlspecs.org`.

[16] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[17] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[18] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[19] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[20] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[21] Scott Wiltamuth and Anders Hejlsberg. C# language specification. From `http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp` (Date retrieved: April 2, 2003), December 2002.