Specifying and Checking Method Call Sequences in JML

Yoonsik Cheon and Ashaveena Perumandla

TR #05-04
February 2005; revised April 2005.

Keywords: call sequences, runtime assertion checking, formal interface specification, design by contract, Java Modeling Language (JML), Java language

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, monitors, testing tools, theory; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — Denotational semantics.

Appeared in *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Las Vegas, Nevada, USA, June 27-30, 2005*, pages 511–516.

Copyright © 2005 by CSREA Press. All rights reserved.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Specifying and Checking Method Call Sequences in JML

Yoonsik Cheon and Ashaveena Perumandla
Department of Computer Science
University of Texas at El Paso
El Paso, TX 79968
cheon@cs.utep.edu, aperumandla@utep.edu

***Abstract*— In a pre and postconditions style specification, it is difficult to specify allowed sequences of method calls, often called *protocols*. However, the protocols are essential properties of reusable object-oriented classes and application frameworks, and the approaches based on the pre and postconditions, such as design by contracts (DBC) and formal behavioral interface specification languages (BISL), are being accepted as a practical and effective way of describing precise interfaces of (reusable) program modules. We propose a simple extension to JML, a BISL for Java, to specify protocol properties in an intuitive and concise manner. We also define a formal semantics of our extension and provide runtime checks. We believe that our approach can be easily adopted for other BISLs.**

I. Introduction

As many program modules are developed and reused in the forms of library classes, software components, and application frameworks, the need increases accordingly to specify the interfaces of program modules precisely and unambiguously. An interface specification describes formally a program module by specifying both the syntactic interface and the behavior of the module. The Java Modeling Language (JML) [1] is a behavioral interface specification language (BISL) for Java to describe the interfaces of Java program modules such as classes and interfaces.

In JML, the behavior of a program module is specified by writing, among others, pre and postconditions of the methods exported by the module. The pre and postconditions are viewed as a contract between the client and the implementor of the module. The client must guarantee, before calling a method m exported by the module, that m's precondition holds, and the implementor must guarantee that m's postcondition holds after such a call. The assertions in pre and postconditions are usually written in a form that can be compiled, so that violations of the contract can be detected at runtime. Checking pre and postconditions at runtime—first pioneered by Design by Contract (DBC) tools [2] [3] [4]—is useful for checking the correctness of a program with respect to its specification.

The pre and postcondition style assertions found in JML and DBC are effective for specifying the functional behavior of a program module. By a functional behavior we mean the input and output relation of the module—e.g., for an input value x a method m should produce an output value y. In addition to the functional behavior, there are other behavioral properties that clients of a program module have to know to use the module. One such a property that we call *protocols* in this paper is the order in which the methods exported by the module have to be called. The protocol properties are most often found in reusable library classes and object-oriented application frameworks. For example, methods of applets—Java classes embedded in HTML documents—should be called in a certain, predefined order. In the next section we will discuss applet protocols and specify them formally. Another common pattern that needs a protocol specification is what we call the build-and-access pattern in which some method builds or calculates derived attributes of an object and several observer methods are used to access the derived attributes once they become available. For example, most compilers represent a source code program internally as a tree, called a *parse tree* or an *abstract syntax tree*, where some of the nodes represent expressions. The type of an expression node becomes available only after a typecheck is performed on the tree. This means that methods that depend on the type, such as type access methods and code generation methods, should be called after typechecking—i.e., after the typecheck method is called and completed.

In this paper we first show that the pre and postcondition style assertions are inadequate for specifying the protocol properties of a program module, by using JML as our BISL. We then extend JML to specify the protocols in an intuitive and natural style. The essence of our extension is to separate the protocol assertions from the functional assertions of pre and postconditions. The protocol is written in a new specification clause, called `call_sequence` clause. The call sequence clause constrains the order in which methods of a class or interface should be called by clients, by specifying permissible sequences of method calls. We use a regular expression-like notation to write call sequence assertions.

We define the formal semantics of our extension to JML, i.e., call sequence assertions. The meanings of sequential

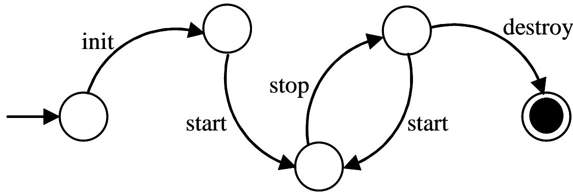

Fig. 1. The applet protocol.

Java programs are formally defined in terms of method calls and returns. We model the state of a program execution as a history of method calls and returns, and a program execution as a transition on histories. A program execution satisfies a call sequence assertion if its history is permitted by the call sequence assertion; a call sequence assertion denotes a set of histories.

The semantics provides a sound foundation for checking call sequence assertions at runtime. For runtime checks we translate call sequence assertions into finite automata. For each method call and return, we check whether such a transition is allowed by the finite automata; if the transition is not permissible, an assertion violation error is reported. As in pre and postcondition checks, the runtime checks of call sequence assertions are transparent when no assertions are violated. A prototype implementation has been completed by extending the JML compiler.

II. The Problem

Pre and postconditions are an excellent tool for specifying functional behaviors of program modules, and approaches based on the pre and postconditions, such as design by contract (DBC) [2] and behavioral interface specification languages (BISL) [1], are being accepted by programmers as a practical programming and specification methodology [5]. The precondition specifies the obligation that clients of a module has to satisfy. The clients have to call the module in a state where the precondition is satisfied; otherwise, nothing is guaranteed by the module's implementation.

However, clients often have to meet other requirements that may not be classified as functional, and thus difficult to be stated in a pre and postconditions style specification. An example of these requirements is the order in which the clients have to call the exported methods of a module. This order of method calls, often called *protocols*, is most commonly found in object-oriented application frameworks, such as graphical user interface classes [6]. For example, clients of Java applets[1] should call applet methods in a certain order (see Fig. 1). Specifically, the clients have to call the `init` method first, then the `start` and `stop` methods every time the Web page containing the applet is visited (or revisited) and left respectively, and finally the `destroy` method when the applet is not needed.

[1]Applets are Java classes that are embedded in HTML documents, and clients of applets are Web browsers and applet viewers.

```
package java.applet;

public class Applet {

  //@ public static final ghost int
  //@   PRISTINE = 1,
  //@   INIT = 2,
  //@   START = 3,
  //@   STOP = 4,
  //@   DESTROY = 5;

  //@ public ghost int state = PRISTINE;

  //@ requires state == PRISTINE;
  public void init() {
    //@ set state = INIT;
    // ...
  }

  //@ requires state == INIT
  //@   || state == STOP;
  public void start() {
    //@ set state = START;
    // ...
  }

  //@ requires state == START;
  public void stop() {
    //@ set state = STOP;
    // ...
  }

  //@ requires state == STOP;
  public void destroy() {
    //@ set state = DESTROY;
    // ...
  }

  // other fields and methods ...
}
```

Fig. 2. The applet protocol specified in JML

Fig. 2 shows an example specification of the applet protocol written in JML. In JML, specifications are typically annotated in source code as special comments, e.g., `//@` in the example. The first five annotations define constants for use in annotations, and the next annotation introduces a specification-only field named `state`. The JML modifier `ghost` indicates that the declared field is for use only in specification. The ghost field `state` keeps track of the protocol state of the applet (refer to Fig. 1). For example, the precondition of the `init` method, written in the `requires` clause, states that the `init` method should be called when the value of the `state` field is `PRISTINE`, the initial value of the field. The `init` method sets the `state` field to `INIT` to record the fact that it was called. Similarly, the precondition of the `start` method states that the method should be called after either the `init` method or the `stop` method is called, and the method

```
package java.applet;
public class Applet {
  //@ public call_sequence
  //@   init() : (start() : stop())+
  //@   : destroy();
  // member declarations ...
}
```

Fig. 3. The applet protocol specified in the extended JML

also updates the `state` field appropriately. The rest of methods are specified similarly. In short, the protocol state is explicitly modeled and manipulated in annotations by using a specification-only field and the `set` statement.

What is wrong with the above example specification? There is an impedance mismatch problem. It is hard to read and understand the intent and the meanings of the specification because the protocol is not apparent from the way the specification is written and thus has to be inferred by the reader. Similarly, it is difficult to write protocol properties in such a specification. The main problem is that the protocol cannot be directly expressed in annotations. The problem would become aggravated in practice because protocol assertions would be intermingled with functional assertions in the pre and postconditions. The approach does not work for Java interfaces because interfaces in Java cannot contain method definitions.

In the next section we present our new approach to writing protocol specifications in JML.

III. Our Specification Approach

Our approach is to separate the protocol assertions from the functional assertions. As before, the functional properties are written in the pre and postconditions. However, the protocol properties are written directly as separate annotations in a suitable notation. For this, we extend JML to introduce a new specification clause, called a `call_sequence` clause. The `call_sequence` specification constrains the order in which methods of a class or interface should be called by clients. It specifies the allowed sequences of method calls.

Fig. 3 shows the applet protocol specified with the newly introduced `call_sequence` clause. We use a regular expression-like notation to express call sequences. In the example, the infix `:` operator denotes a sequential composition of two call sequences, and the postfix + operator denotes one or more sequential compositions of a call sequence. The meanings of the specification should be apparent. The specification describes the life-cycle of applets by stating that `init` should be called first, followed by some number of calls to `start` and `stop`, and finally `destroy` should be called as the last call.

In the `call_sequence` annotation, one can also specify alternative calls and nested calls. The following call sequence states that the `start` method should call either the `repaint` method or the `paint(Graphics)` method, directly or indirectly; nested calls are enclosed in a pair of curly braces, preceded by the calling method name. The example also shows that an overloaded method can be disambiguated by specifying its parameter types, e.g., `paint(Graphics)`.

```
init()
  : (start(){repaint()
     || paint(Graphics)} : stop())+
  : destroy()
```

Our approach, with a small extension to the JML language, allows one to specify protocol aspects of program modules directly. For example, the call sequence specification of Fig. 3 is a direct description of the finite state machine shown in Fig. 1. The specification is also concise and intuitive.

In the following section we define formal semantics of the `call_sequence` clause. The formal semantics provides a foundation for checking the protocol specifications at runtime.

IV. The Semantics

In this section we first define the meanings of sequential programs in terms of method calls and returns and the meanings of call sequence specifications. We then formalize when a program satisfies a call sequence specification.

A. The Semantics of Sequential Programs

The state of a program execution is represented as the history of method calls and returns. We use the following notations to formalize the notion of histories and the semantics of sequential programs.

$a \in$ Action
$m \in$ Method
$h \in$ History $\equiv 2^{\text{seq Action}}$
$a ::= m.\texttt{begin} \mid m.\texttt{end}$

An execution of a sequential program is modeled as a sequence of actions, where an action is either a method call or a method return. A call to a method, m, is denoted as $m.\texttt{begin}$, and a return from m is denoted as $m.\texttt{end}$. Thus, a *history* is a sequence of $m.\texttt{begin}$ and $m.\texttt{end}$, where m is a method. We use a pair of angle brackets ($\langle\rangle$) to denote a sequence. For example, $\langle m.\texttt{begin}\ \ m.\texttt{end}\rangle$ denotes the state of program execution where a call to m was made and returned, and $\langle m_1.\texttt{begin}\ \ m_2.\texttt{begin}\rangle$ denotes a history where a call to m_1 initiates another call, a call to m_2—i.e., m_1 calls m_2, directly or indirectly—and both calls are not returned.

The behavior of a sequential program, i.e., program execution, is modeled as a transition on histories. This transition, T : History $\times$ History, is defined as follows.

$m \in$ Method
$s \in$ CallSequence
$s ::= m$
| m.begin
| m.end
| s || s
| s : s
| s*
| s+
| (s)

Fig. 4. Abstract syntax of method call sequences

$$h \xrightarrow{a} h^\frown\langle a\rangle$$

where the notation $\frown$ denotes concatenation of two sequences. That is, calling a method and returning from a method call is to append the corresponding action to the end of the current history.

The relation $\xrightarrow{*}$ denotes the reflexive-transitive closure of the relation $\longrightarrow$, and the relation $\Sigma_0 \xrightarrow{*} \Sigma$ denotes an execution of a sequential program starting from the initial state Σ_0.

B. The Semantics of Call Sequence Specifications

In this section we define the semantics of call sequence specifications by using the abstract syntax shown in Fig. 4. In the abstract syntax, the start and the end of a sequence of nested calls made from a method m are represented by m.begin and m.end respectively. That is, m.begin corresponds to the concrete syntax "m {" and m.end corresponds to "}", the matching ending brace.

A call sequence specification constrains a program by specifying the allowed histories of all executions of the program. A call sequence specification, therefore, denotes a set of (allowed) histories of program executions. We give the semantics of call sequence specifications by defining a mapping from specifications to sets of histories, $\mathcal{M}$: CallSequence $\rightarrow 2^{\text{History}}$. The meaning function, $\mathcal{M}$, is defined as follows.

$$\begin{array}{ll}
\mathcal{M}[\![m]\!] & \stackrel{\text{def}}{=} \{\langle m\texttt{.begin}\ m\texttt{.end}\rangle\} \\
\mathcal{M}[\![m\texttt{.begin}]\!] & \stackrel{\text{def}}{=} \{\langle m\texttt{.begin}\rangle\} \\
\mathcal{M}[\![m\texttt{.end}]\!] & \stackrel{\text{def}}{=} \{\langle m\texttt{.end}\rangle\} \\
\mathcal{M}[\![s_1 : s_2]\!] & \stackrel{\text{def}}{=} \{h_1{}^\frown h_2 \mid h_1 \in \mathcal{M}[\![s_1]\!],\ h_2 \in \mathcal{M}[\![s_2]\!]\} \\
\mathcal{M}[\![s_1 \,||\, s_2]\!] & \stackrel{\text{def}}{=} \mathcal{M}[\![s_1]\!] \cup \mathcal{M}[\![s_2]\!] \\
\mathcal{M}[\![s\texttt{*}]\!] & \stackrel{\text{def}}{=} \bigcup_{i=0..\infty} \mathcal{M}[\![s]\!]^i \\
\mathcal{M}[\![s\texttt{+}]\!] & \stackrel{\text{def}}{=} \bigcup_{i=1..\infty} \mathcal{M}[\![s]\!]^i \\
\mathcal{M}[\![\texttt{(}s\texttt{)}]\!] & \stackrel{\text{def}}{=} \mathcal{M}[\![s]\!]
\end{array}$$

where $\mathcal{M}[\![S]\!]^i$ is defined as follows.

$$\begin{array}{l}
\mathcal{M}[\![s]\!]^0 \stackrel{\text{def}}{=} \{\langle\rangle\} \\
\mathcal{M}[\![s]\!]^i \stackrel{\text{def}}{=} \{h_1{}^\frown h_2 \mid h_1 \in \mathcal{M}[\![s]\!],\ h_2 \in \mathcal{M}[\![s]\!]^{i-1}\}
\end{array}$$

The definition is straightforward and reflects our intuitive understandings of call sequence specifications. Note that the meaning function states that m is a syntactic sugar for m.begin : m.end, a sequential composition of m.begin and m.end.

C. Satisfaction Relation

When does a program execution satisfy a call sequence specification? A program execution satisfies a call specification, s, if the history of the program execution is contained in the set of sequences denoted by the specification s. We define the satisfaction relation between program executions and call sequence specifications formally as follows.

$$h \vdash s \text{ if } h \models \mathcal{M}[\![s]\!]$$

where, the $\models$ relation is defined as:

$$h \models \mathcal{M}[\![s]\!] \text{ iff } h \sqsubseteq h_1, \text{for some } h_1 \in \mathcal{M}[\![s]\!]$$

The notation $h \sqsubseteq h_1$ means that h is a prefix of h_1. That is,

$$h \sqsubseteq h_1 \text{ iff } \exists\, h_2 \text{ such that } \langle h^\frown h_2\rangle = h_1$$

A program execution meets a call sequence specification if its history is a prefix of some sequence denoted by the specification. Remember from the earlier section that a call sequence denotes a set of sequences of method calls.

The above definition is rather strong in the sense that the specification is assumed to be complete by considering all methods. A program execution that calls a method not appearing in the specification doesn't meet the specification. In practice, we would like to write call sequence specifications focusing only on a small subset of methods, without worrying about the rest of the methods. In fact, it is often impossible to consider all possible methods when writing specifications, e.g., those of other classes including future subclasses. Thus, we define a weaker version of satisfaction relations.

The loose (or weak) semantics of call sequence specifications is defined as:

$$h \restriction \alpha(s) \vdash s$$

where $h \restriction \alpha(s)$ is the projection of h over the alphabet of s. The alphabet of s, $\alpha(s)$, is the set of methods appearing in s. Thus, $h \restriction \alpha(s)$ is the sequence obtained from h by discarding any methods that do not appear in s. In the loose semantics we don't care about calls to methods that don't appear in the specification.

In the next section we use the loose semantics to provide runtime checks for call sequence specifications.

V. Runtime Checks

The next big question is how to check at runtime whether an execution of a program satisfies the call sequence specifications of the program.

Remember from Section IV-A that a program execution is modeled as a sequence of transitions on histories, and each transition has the form: $h \xrightarrow{a} h^\frown\langle a\rangle$. Thus, a natural approach to runtime checking is to ensure that, before committing a transition, such a transition is allowed by the call sequence specifications. To facilitate this check, we translate a call sequence specification into a finite state automata. The automata is an executable representation of the specification. Recall that a in a transition, $h \xrightarrow{a} h^\frown\langle a\rangle$, is either m`.begin` or m`.end` for some method m. This means that we need to check the validity of a transition only in the pre and post-states, i.e., right before a method call and right after the method return. The following code summarizes the essence of our approach to runtime checking.

```
// for trans h -a-> h⌢⟨a⟩ with spec s
if (a ∈ α(s)) {
  if (trans a possible?) {
     make trans a on the automata;
  } else {
     inform assertion violation;
  }
}
```

The first condition implements the loose semantics by checking method calls made only to those methods appearing in the specification s (see Section IV-C).

Each method is instrumented to execute the above code in the pre and post-states of every call, and a `call_sequence` clause is translated into a finite state machine injected into the bytecode of the class. A program execution should satisfy all call sequence specifications if a class has more than one `call_sequence` clause.

In sum, the essence of our approach is to translate a call sequence specification into a corresponding finite state automata and interpret the start and end of a method invocation as a transition on the automata.

A prototype of this approach has been implemented by extending the JML compiler [7]. We believe that our prototype implementation is sound and complete with respect to the semantics defined in Section IV.

VI. Evaluation and Future Work

We are currently performing two case studies to evaluate the effectiveness and practicality of our approach. The first case study is to examine existing JML specifications of Java library classes, such as various collection classes, that are shipped with the JML distribution. Our preliminary finding is that protocol assertions are totally missing from most specifications. Our plan is to extend the specifications with the newly introduced call sequence clause. The second case study is to look at source programs of various JML tools, identify protocol properties, and specify them in the extended JML. The JML tools are non-trivial software that consists of several packages and a large number of classes and interfaces built on top of an existing Java compiler. They also show the characteristics of object-oriented application frameworks, e.g., the inversion of control. We believe that the formal specification of protocols properties benefits both the beginning and the seasoned JML developers. Our initial finding is that there are many places where protocols are specified informally either as Javadoc comments or as informal descriptions in JML. This re-confirmed our belief on the need of specification facilities to specify protocol properties. Ironically we also found a similar example in our extension to the JML compiler. The parse tree node classes representing various call sequence expressions have a method called `buildFA()` to construct finite automata and several access methods, such as `states()`, `labels()`, `startState()` and `finalStart()`. This is an example of the build-and-access pattern, and thus the access methods should be called after a call to the build method, `buildFA()`. As a side product of these case studies, we hope to identify several patterns of protocol properties and document them in a catalog.

We are also in the process of refining our prototype implementation of runtime checks, e.g., optimizing the automata to improve their time and space efficiency. The current implementation does not support `static` call sequence assertions. It also doesn't support inheritance of call sequence assertions. The loose semantics facilitates the interpretation of inheritance; the inherited call sequence clauses can be thought of being conjoined (in the sense of multiple call sequence clauses) to the inheriting class or interface. That is, an execution of a subtype (class or interface) should conform to not only its own call sequence specifications but also all inherited ones. In terms of runtime checks, this means that the call sequence checking code (see Section V) of a subtype should call the corresponding code of all its supertypes.

A future work is to extend our specification approach and runtime checking framework to multithreaded programs.

VII. Related Work

Meyer pioneered design by contract (DBC) in the programming language Eiffel by integrating executable assertions into programs in the forms of pre and postconditions and class invariants [3] [4]. However, Eiffel does not provide a built-in facility to write and check protocol properties. As in JML, they have to be embedded into pre and postconditions and in-line assertions.

Eiffel contributed to the availability of similar DBC facilities in other programming languages. For example, there are several DBC tools for Java [8] [9] [10] [11]

[12]. The approaches vary widely from a simple assertion mechanism similar to the `assert` macro of C to full-fledged contract enforcement tools. However, except for Jass [8] none of the mentioned approaches supports protocol specifications.

Jass [8] inspired our work on supporting protocol specifications in DBC. In Jass, protocol properties are called *trace assertions*, and a trace assertion specifies permissible sequences of method calls in a CSP-like notation. Thus, one can also express processes, parallelism, conditional, and data exchange among processes. In our approach, we adopted a simple, regular expression-like notation instead of process algebra. As in our approach, the trace assertions are also interpreted loosely. The Jass precompiler translates the trace assertions into runtime checks. An alternative approach called Jassda [13] [14] checks trace assertions by observing the events generated by debuggers through the Java Debug Interface (JDI). A shortcoming of this alternative is that the target program must run in the debugging mode.

Ada annotation languages, such as Anna [15] and SPARK [16], do not support protocol specifications.

A more recent initiative, Spec# [17], extends C# with contract specifications. However, no construct was introduced to specify protocol properties.

Outside the DBC community, there have been several attempts to formalize protocol aspects of programs, such as frameworks (e.g., [6]), and the earliest work can be traced back to Bartussek and Parnas work on trace assertions [18].

VIII. Conclusion

We proposed and implemented a simple extension to JML to formally specify protocol properties of program modules. The extension allows one to specify the allowed sequences of method calls in an intuitive and concise manner. We believe that our extension provide a practical and effective way to specify formally the protocol aspects of reusable classes and application frameworks. The extension may be also useful for automating intra-class testing, e.g., automatically generating intra-class test data.

A prototype implementation of our extension is available from `http://www.cs.utep.edu/~cheon/download`.

Acknowledgement

The work of the authors was supported in part by The University of Texas at El Paso through URI grant 14-5078-6151. Thanks to Myoung Kim for reading an earlier draft of this paper.

References

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: A notation for detailed design," in *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston: Kluwer Academic Publishers, 1999, pp. 175–188.

[2] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[3] ——, *Eiffel: The Language*, ser. Object-Oriented Series. New York, NY: Prentice Hall, 1992.

[4] ——, *Object-oriented Software Construction*, 2nd ed. New York, NY: Prentice Hall, 1997.

[5] D. S. Rosenblum, "A practical approach to programming with assertions," *IEEE Transactions on Software Engineering*, vol. 21, no. 1, pp. 19–31, Jan. 1995.

[6] N. Soundarajan and S. Fridella, "Framework-based applications: From incremental development to incremental reasoning," in *Software Reuse: Advances in Software Reusability, 6th International Conerence, ICSR-6, Vienna, Austria, June 27-29, 2000, Proceedings*, ser. Lecture Notes in Computer Science, W. B. Frakes, Ed., vol. 1844. Springer-Verlag, 2000, pp. 100–116.

[7] Y. Cheon and G. T. Leavens, "A runtime assertion checker for the Java Modeling Language (JML)," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, H. R. Arabnia and Y. Mun, Eds. CSREA Press, June 2002, pp. 322–328. [Online]. Available: ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf

[8] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass - Java with assertions," in *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, 2001, published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

[9] A. Duncan and U. Holzle, "Adding contracts to Java with Handshake," Department of Computer Science, University of California, Santa Barbara, CA, Tech. Rep. TRCS98-32, Dec. 1998.

[10] R. B. Findler and M. Felleisen, "Contract soundness for object-oriented languages," in *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, Oct. 2001, pp. 1–15.

[11] M. Karaorman, U. Holzle, and J. Bruno, "jContractor: A reflective Java library to support design by contract," in *Meta-Level Architectures and Reflection, Second International Conference on Reflection '99, Saint-Malo, France, July 19–21, 1999, Proceedings*, ser. Lecture Notes in Computer Science, P. Cointe, Ed. Springer-Verlag, July 1999, vol. 1616, pp. 175–196.

[12] R. Kramer, "iContract – the Java design by contract tool," *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pp. 295–307, 1998.

[13] M. Brörkens and M. Möller, "Jassda trace assertions, runtime checking the dynamic of java programs," in *Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, Germany*, I. Schieferdecker, H. König, and A. Wolisz, Eds., Mar. 2002, pp. 39–48.

[14] ——, "Dynamic event generation for runtime checking using the JDI," in *Proceedings of the Federated Logic Conference Satellite Workshops, Runtime Verification, Copenhagen, Denmark*, K. Havelund and G. Rosu, Eds., July 2002, electronic Notes in Theoretical Computer Science 70(4).

[15] D. Luckham, *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs*, ser. Texts and Monographs in Computer Science. New York, NY: Springer-Verlag, 1990.

[16] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[17] M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# programming system: An overview," in *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, ser. Lecture Notes in Computer Science, 2004, to appear.

[18] W. Bartussek and D. L. Parnas, "Using assertions about traces to write abstract specifications for software modules," in *Proceedings of the Second Conference of the European Cooperation on Informatics: Information Systems Methodology, October 10-12, 1978, London, UK*, ser. Lecture Notes in Computer Science, G. Bracchi and P. C. Lockemann, Eds. Springer-Verlag, 1978, vol. 65, pp. 211–236.