11-2005

Specifying and Checking Method Call Sequences of Java Programs

Yoonsik Cheon
The University of Texas at El Paso, ycheon@utep.edu

Ashaveena Perumandla

Comments:
UTEP-CS-05-36.

Specifying and Checking Method Call Sequences of Java Programs

Yoonsik Cheon and Ashaveena Perumandla

TR #05-36
November 2005; revised April 2006

Keywords: method call sequence specification, runtime checking, assertion, pre and postconditions, programming by contract, JML language.

1998 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, validation; 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, mechanical verifications, pre- and post-conditions, specification techniques; F.3.2 [*Logics and Meanings of Programs*] Semantics of Programming Languages — Denotational semantics.

This is an extended version, submitted for journal publication, of the paper 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.

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 of Java Programs

Yoonsik Cheon and Ashaveena Perumandla

Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
{ycheon,aperumandla}@utep.edu

Abstract. In a pre and postcondition-style specification, it is difficult to specify the allowed sequences of method calls, referred to as *protocols*. 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 tool for describing precise interfaces of (reusable) program modules. We propose a simple extension to the Java Modeling Language (JML), a BISL for Java, to specify protocol properties in an intuitive and concise manner. The key idea of our approach is to separate protocol properties from functional properties written in pre and postconditions and to specify them in a regular expression-like notation. The semantics of our extension is formally defined and provides a foundation for implementing runtime checks. Case studies have been performed to show the effectiveness our approach. We believe that our approach can be adopted by other BISLs.

Keywords: method call sequence specification, runtime checking, assertion, pre and postconditions, programming by contract, JML language.

1 Introduction

As many program modules are developed and reused in the form of library classes, software components, and application frameworks, there is an increased need 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) (Burdy et al., 2005; Leavens et al., 2005, 1999) 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, for example, by writing 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 (Meyer, 1992a,b, 1997)—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 a *protocol* 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 a build-and-access pattern. In this pattern, some methods build or calculate derived attributes of an object and other methods access or observe the attributes. For example, most compilers represent a source code program internally as a tree, called a *parse tree* or an *abstract syntax tree*. Some of the nodes of a parse tree may represent expressions. The type of an expression becomes known only after typechecking has been performed on the tree. This means that those methods that depend on the type of an expression, e.g., type access methods and code generation methods, should be called after typechecking—i.e., after the typecheck method has been called and completed.

In this paper we first show that the pre and postcondition-style assertions are inadequate for specifying 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 way. 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 a *call sequence clause* (Cheon and Perumendla, 2005). The call sequence clause constrains the order in which methods of a class or interface should be called by clients, by specifying the allowed 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 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 state machines. For each method call and return, we check whether such a transition is allowed by the finite state machine; if the transition is not permitted, 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. The JML compiler (Cheon, 2003) was extended to recognize the call sequence clause and to translate it into runtime checking code.

The rest of this paper is organized as follows. In Section 2, we illustrate the problem of specifying protocol properties in pre and postconditions tangled with functional properties. In Sections 3 and 4, we explain our specification approach by describing the syntax and semantics of call sequence clauses. In particular, we define when a sequential program execution satisfies a call sequence assertion. The semantics provides a foundation for translating call sequence specifications into runtime checking code, which is discussed in Section 5. In Section 6, we present two case studies that we performed to evaluate the effectiveness of our approach. In Section 7, we describe related work, and a conclusion follows in Section 8.

2 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) (Meyer, 1992a) and behavioral interface specification languages (BISL) (Leavens et al., 1999), are being accepted by programmers as a practical programming and specification methodology (Rosenblum, 1995). The precondition specifies the obligation that clients of a module have 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 are difficult to state in a pre and postcondition-style specification. An example of these requirements is the order in which the clients have to call the exported methods of a module. These ordering dependencies among method calls, referred to as *protocols* in this paper, are most commonly found in object-oriented application frameworks, such as graphical user interface classes (Soundarajan and Fridella, 2000). 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.

Fig. 2 shows an example JML specification of the applet described in Fig. 1. In JML, specifications are annotated in source code as special comments such as `//@` and `/*@ @*/`. The first annotation defines several constants for use in

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

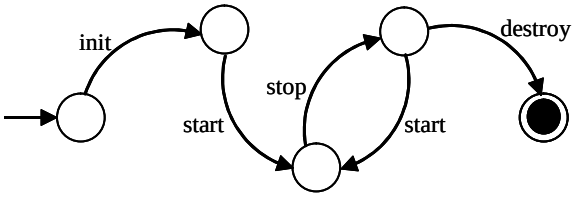

Fig. 1. The applet protocol

JML assertions, and the next annotation introduces a specification-only field `state`. The JML modifier `ghost` indicates that the declared field is for use only in specifications. 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 only 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. The JML specification statement `set` is for assigning a value to a ghost field. 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 also updates the `state` field. 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` specification statement.

What is wrong with this specification? There is an important impedance mismatch problem. The protocol—ordering dependencies among method calls—is not apparent from the specification. It is described indirectly by coding a finite state machine by hand. Writing specifications like this is cumbersome and error-prone, and worse the resulting specifications are hard to read and understand. The intent or the meaning of such a specification is hard to grasp because the protocol is not apparent from the way the specification is written and thus has to be inferred or guessed by the reader. The problem becomes aggravated in practice because protocol assertions are to be mixed up with functional assertions in pre and postconditions (see an example in Section 6). In addition, the approach does not work for Java interfaces because interfaces in Java cannot contain method definitions, and thus the `set` statement.[2]

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

[2] The `set` statement is treated as a Java statement and thus can appear only in a place where a Java statement is allowed. There are sporadic discussions in the JML community to add support for manipulating ghost fields in Java interface definitions. However, no noticeable progress has been made yet in terms of language definition and implementation.

```
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;
  //@ ensures state == INIT;
  public void init() {
    //@ set state = INIT;
    // ...
  }

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

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

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

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

Fig. 2. The applet protocol specified in JML

3 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* (Cheon and Perumendla, 2005). The call sequence clause specifies ordering dependencies among method calls. It specifies the allowed sequences of method calls and thus constrains the order in which methods of a class or interface should be called by clients.

Fig. 3 shows the applet protocol specified with the newly introduced call sequence clause. We use a regular expression-like notation to express call sequence assertions. In the example, the infix `:` operator denotes a sequential composition of two call sequence expressions, and the postfix `+` operator denotes one or more sequential compositions of a call sequence expression. 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 alternating calls to `start` and `stop`, and `destroy` should be called last.

```
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

In the call sequence assertion, one can also specify alternative calls and nested calls. The following call sequence specification 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 formal 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. This specification is also concise and intuitive. Compare it with the one in Fig. 2.

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.

4 The Semantics

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

4.1 The Semantics of Sequential Programs

The state of a sequential program execution is represented as a history of method calls and returns. We use the following notation 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 by $m.\texttt{begin}$, and a return from m is denoted by $m.\texttt{end}$. The set Method denotes the set of all methods defined for the program under consideration. Thus, a *history* of a program execution is a sequence of $m.\texttt{begin}$ and $m.\texttt{end}$, where m is a method of the program. 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 a 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.

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

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.

Note that the above definition of histories provides a global view of the program state. In object-oriented programs, however, we are often more interested in the history of an individual object, i.e., method calls to and returns from a particular object. We can obtain this per-object history (or lifetime of an object) by projecting the global history upon a particular object of the system. In fact, our current implementation is based on this per-object view of histories (see Section 5).

4.2 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 end brace.

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

Fig. 4. Abstract syntax of call sequence expressions

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, $\mathcal{M} : \text{CallSequence} \to 2^{\text{History}}$. The meaning function, $\mathcal{M}$, is defined as follows.

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

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

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

The definition is straightforward and reflects our intuitive understandings of call sequence expressions. 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`.

4.3 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 satisfies a call sequence specification if its history is a prefix of some sequence denoted by the specification. Recall from the earlier section that a call sequence specification denotes a set of method call sequences.

The above definition is rather strong in the sense that a specification is assumed to be complete by considering all the methods of a program. A consequence of this strong definition is that an execution that calls some method not appearing in a call sequence specification doesn't satisfy the specification. This has a practical implication, as in practice we would like to write call sequence specifications by considering only a small number of methods of interest, without worrying about the rest of the methods. In fact, it is often impossible to consider all the methods of a program, e.g., methods 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:

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

where $h \upharpoonright \alpha(s)$ 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, $h \upharpoonright \alpha(s)$ 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.

5 Our Implementation Approach

This section summarizes the implementation of call sequence specifications. We extended the JML compiler (Cheon, 2003) to recognize the new call sequence clause and to translate it into runtime checking code. The work of runtime checking code is transparent in that when no assertions are violated, the behavior of original code is not changed except for time and space measurements (Cheon and Leavens, 2002). We believe that our implementation is sound with respect to the semantics defined in Section 4.

We followed JML's approach to runtime assertion checking. A class or interface's call sequence specifications are translated into an assertion checking method, called a *call sequence checking method* (see Section 5.1). The call sequence checking method is responsible for ensuring that a method call to (or return from) an object of the type is permitted by the call sequence specifications. The call sequence checking method will be called at an appropriate point of control flow of program execution. In particular, it will be called before and after the execution of every method, including the constructors, of the type. As in JML, this is done by introducing a wrapper method for each method of the type (see Fig. 5). The *wrapper method* forwards client calls to the original method but wrapped with assertion checks, including calls to the call sequence checking method `checkCS$instance$T`.

```
/** wrapper method */
public void m(/* ... */) {
  // other pre-state checks, e.g., precondition and invariant
  checkCS$instance$T("m.begin" /*, ... */); // check call to m
  m$original();
  // other post-state checks, e.g., postcondition and invariant
  checkCS$instance$T("m.end" /*, ... */); // check return from m
}

/** original method */
private void m$original(/* ... */) { /* ... */ }

/** call sequence checking method */
public final void checkCS$instance$T(String a /*, ... */) { /* ... */ }
```

Fig. 5. Wrapper approach to checking call sequence specifications

5.1 Call Sequence Checking Methods

The call sequence checking method ensures that each method call and return is permitted by the call sequence specification. Recall from Section 4.1 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 way to check a call sequence specification is to ensure that, before committing a transition, the transition is indeed permitted by the specification. The call sequence checking method is responsible for this. To facilitate this check, we translate a call sequence specification into a finite state machine. The state machine is an executable representation of the specification, and each object has its own state machine. 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 transition only in the pre and post-states, i.e., right before a method call and right after the method return; this is done by the wrapper method by calling the call sequence checking method in the pre and post-states. A consequence of having a separate state machine for each object and letting the wrapper method invoke the call sequence checking method is that it only supports per-object protocols (see Section 6.1).

Fig. 6 shows skeletal code of the call sequence method. The first `if` statement implements the loose semantics by considering only those methods that appear in the call sequence specification s; recall that $\alpha(s)$ denotes the alphabet of s (see Section 4.3). If a given transition a is permitted by the finite state machine, the transition is made; otherwise, an assertion violation error is thrown. If a type has more than one call sequence specification, each call sequence specification is translated into a separate finite state machine, and a program execution should satisfy all call sequence specifications. In sum, the essence of our approach is to translate a call sequence specification into a finite state machine and to interpret the start and end of a method invocation as a transition on the state machine.

```
public final void checkCS$instance$T(String a /*, ... */) {
  boolean rac$b = true;
  // for h --a--> h^⟨a⟩ with call sequence specification s
  if (a ∈ α(s)) {
     if (transition a possible from current state?) {
         make the transition a on the state machine;
     } else {
        rac$b = false;
     }
  }
  if (!rac$b) {
     throw new JMLCallSequenceError(/* ... */);
  }
}
```

Fig. 6. Skeletal call sequence checking method of type T

5.2 Inheritance of Specifications

Like other JML assertions such as invariants and pre and postconditions, public and protected call sequence specifications are inherited by subclasses and subinterfaces. Since specifications can be attached to interfaces, JML supports multiple inheritance. There are three kinds of inheritance possible: class-class inheritance, class-interface inheritance, and interface-interface inheritance. The loose semantics facilitates the interpretation of inheritance, as the additional methods of subtypes can be ignored when interpreting the inherited call sequence specifications. The inherited call sequence specifications can be thought of as being conjoined (in the sense of multiple call sequence clauses) to the inheriting class or interface. A subtype has to satisfy both the inherited specifications and those explicitly specified in it.

As in JML (Cheon, 2003; Cheon et al., 2005), we use a *delegation approach* to support inheritance of call sequence specifications. A subtype delegates to its direct supertypes the responsibility of checking inherited specifications by calling the appropriate assertion checking methods. We use Java's reflection facility to delegate assertion checks to direct supertypes. This is to support separate compilation; the supertypes may not contain assertion checking code because they may not be compiled with the JML compiler.

Let S be a type with a call sequence specification CS and direct supertypes $T_1, \ldots, T_n$. Fig. 7 shows an extended, skeletal call sequence checking method of type S. The method first checks the call sequence specification CS, denoted by [[CS, `rac$b`]]. It then invokes the call sequence checking method of each direct supertype T_i, denoted by $\langle\langle checkCS\ T_i \rangle\rangle$. The call sequence checking methods of supertypes are invoked reflectively by the helper method `rac$check`. In sum, the explicitly specified call sequence specifications are directly checked by the call sequence checking method while the inherited ones are checked indirectly by calling reflectively the call sequence checking methods of the direct supertypes.

5.3 Example

Let's reconsider the applet protocol discussed earlier, of which the call sequence specification is written as: `init() : (start() : stop())+ : destroy()` (see Sections 2 and 3). Let's further assume that the `init` method of a particular applet implementation calls the `getParameter` method to retrieve a certain applet parameter value specified in the HTML tag.

Fig. 8 shows a sequence of method calls for the `init` method and illustrates runtime checking of the weak semantics. As shown, each method invocation incurs at least three additional method invocations: two invocations of the call sequence checking method (`checkCS$_`) and one invocation of the original method (`_$original`). The two calls to the call sequence checking method—made before and after the call to the original method, respectively—are for checking the method call and return with respect to the call sequence specification. In this particular example, all such calls will return normally, i.e., without resulting in an assertion violation error. The `init` call from the client, e.g., an applet browser,

```
public final void checkCS$instance$S(/* ... */) {
  boolean rac$b = true;
  [[CS, rac$b]]
  ⟪checkCS T1⟫
  ...
  ⟪checkCS Tn⟫
  if (!rac$b) {
    throw new JMLCallSequenceError(/* ... */);
   }
}

⟪checkCS Ti⟫ ≡
  if (rac$b) {
    try {
      rac$check("Ti", this, "checkCS$instance$Ti" /*, ... */);
    }
    catch (JMLAssertionError rac$b) {
      rac$b = false;
    }
  }
```

Fig. 7. Skeletal call sequence checking method of type S. The type S is assumed to have a call sequence specification CS and direct supertypes $T_1, \ldots, T_n$. The notation [[CS, rac$b]] denotes a translation of CS into a finite state machine and is the same as the first `if` statement of Fig. 6.

satisfies the specification, assuming that it is the first call; the call is allowed by the finite state machine. The `getParameter` call from the `init` method satisfies the specification trivially because `getParameter` doesn't belong to the alphabet of the specification, i.e., `init`, `start`, `stop`, and `destroy`; thus, the call is never checked against the finite state machine (see Section 5.1).

We believe that our implementation is sound and complete with respect to the semantics defined in Section 4, though we didn't prove it formally. The proof of our claim will consist of two parts: (1) translation of call sequence specifications into finite state machines and (2) checking every method call and return with respect to the finite state machines. The first can be proved by an induction on the structure of the call sequence specification, and the second is obvious by the way we translate methods into wrapper methods for runtime assertion checks, which also can be proved inductively on the structure of Java programs.

6 Evaluation

We performed two case studies to evaluate the effectiveness and practicality of our approach. The first case study was to examine the existing JML specifications of Java library classes, such as various collection classes, that are shipped with the JML distribution (refer to `www.jmlspecs.org`). Our finding from this case

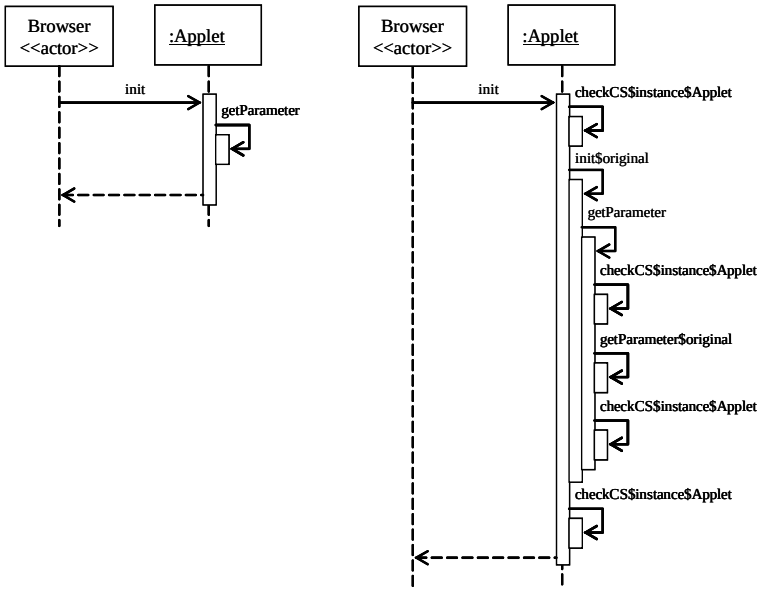

Fig. 8. Sequence of method calls without (left) and with (right) runtime checks

study is that protocol assertions are missing from most specifications. This is understandable, as they are not needed for most of the classes that we examined. Most of the specifications shipped with JML are for simple data classes (e.g., `Boolean` and `Integer`) and collection types (e.g., `Set` and `List`) which have no ordering dependencies among method calls. In a couple of places, however, we noticed explicit coding of finite state machines to specify protocol properties. In the `InputStream` class, for example, there are two model fields, `isOpen` and `wasClosed`, that are used to represent protocol states. Certain methods should be called only in a state where the model fields are true. Fig. 9 shows one such a method, `read`. The specification states that the `read` method should be called when the input stream is open, i.e., in a state where `isOpen` is true. As the model field `isOpen` is set to false only by the `close` method (this is not shown in Fig. 9), the specification really constrains the ordering of method calls between `read` and `close`. However, this protocol property is not explicitly stated in the specification, and thus one has to infer it. We can separate this protocol property specification cleanly and make it explicit by using the call sequence clause, as follows.[3]

```
public call_sequence read()* : close();
```

This separation also has a beneficial side-effect to the functional property specification. The functional specification becomes cleaner and easier to read and

[3] In Java, there is no separate open method for streams, and stream objects become open automatically when they are created.

understand, as the use of `isOpen` can now be removed from the pre and post-conditions.

```
/*@ public normal_behavior
  @   requires isOpen && (inputBytes.length() == readPosition);
  @   assignable \nothing;
  @   ensures \result == -1;
  @ also
  @   requires isOpen && (inputBytes.length() > readPosition);
  @   assignable readPosition, objectState, availableBytes;
  @   ensures isOpen;
  @   ensures \result ==
  @     \old(((JMLByte)inputBytes.itemAt(readPosition)).theByte);
  @   ensures readPosition == \old(readPosition + 1);
  @   ensures 0 <= \result && \result <= 255;
  @ also public exceptional_behavior
  @   requires !isOpen;
  @   assignable \nothing;
  @   signals_only IOException;
  @*/
 public abstract int read() throws IOException;
```

Fig. 9. Partial specification of class `InputStream`. A complete specification is available from the JML distribution (refer to `www.jmlspecs.org`).

The second case study was to look at source code of various JML tools to identify protocol properties and to specify them by using the call sequence clause. The JML tools (Burdy et al., 2005) are non-trivial software that consists of several packages and a large number of classes and interfaces, built on top of an open source Java compiler. They also show the characteristics of object-oriented application frameworks, such as 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.[4] This re-confirmed our belief that we need a specification facility to specify protocol properties formally. 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 a finite state machine 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()`.

[4] In JML, one can mix formal and informal text in a specification (Leavens and Baker, 1999).

In a separate study, we documented formally in JML the application programming interface (API) of the Java security package, the `java.security` package and its extensions (Agarwal et al., 2006). We documented 33 classes and interfaces, and at least six of them have easily noticeable protocol properties. For example, the class `Signature` allows one to compute and verify digital signatures; i.e., the same object may be used for both signing data and verifying signed data. However, the signing (or verifying) methods should be called only if the object is initialized or reinitialized for signing (or verifying), thus constraining the order of method calls. As in the `java.util` package, the percentage of classes that exhibit protocol behavior was not high, at about 18%. For those classes with protocol behaviors, the percentage of the number of lines of annotations saved by using our approach was approximately 10%, but the gains in terms of specification quality (e.g., clarity and readability) are beyond measurement.

As a side product of these case studies we identified several patterns of protocol specifications. The aforementioned build-and-access pattern is one such a pattern. The other two common patterns are the set-and-get pattern and the enter-and-exit pattern. In the set-and-get pattern some method sets the value of a certain field and other methods read the field. In the enter-and-exit pattern two methods are used in a pair, e.g., open and close, lock and unlock, and enter and exit. We expect to identify more protocol patterns as we do more case studies in the future, and these patterns will help us to recognize protocol properties and to specify them formally.

The case studies reinforced our belief that (1) protocol properties should be cleanly separated from the functional property specification of pre and postconditions and (2) they should be stated explicitly so that the reader doesn't have to guess or infer them. We found that the notational simplicity is an advantage and the loose semantics is essential. Without the loose semantics, one has to consider all the methods of a program when writing a call sequence specification, which is often impractical or impossible, especially in the presence of subclassing. We were also able to identify several limitations of our approach, which are discussed in the following subsection.

6.1 Limitations of Our Approach

There are several limitations of our approach regarding the notation, the semantic model, and the implementation. First, the current notation lacks structuring or modularization mechanisms that improve readability and writability of specifications. With a structuring mechanism, for example, one may decompose a protocol specification into a number of smaller sub-protocol specifications. A simple naming facility, such as the following, may be enough.

```
public call_sequence init() : startAndStop+ : destroy();
public call_sequence stopAndStop is start() : stop();
```

Here, the second call sequence clause gives a name, `stopAndStop`, to the specified protocol so that it can be refereed to in the first call sequence specification. The

protocol name may also be useful to give an informative message when a protocol violation is detected at runtime.

Another concern regarding the notation is the expressiveness of our specification language. Our call sequence specifications are limited to regular expressions, and thus we cannot express all possible sequences of method calls. In particular, we cannot express call sequences of the form: $a^n b^n$, where a and b are methods and n is an unknown number. Although we didn't encounter such a protocol in our case studies, there are classes with such a protocol behavior, e.g., symbol tables with scope entering and exiting capabilities. We could extend our notation and implementation to cope with this particular case, e.g., introducing state variables, but the real question is how to balance between expressiveness and notational convenience. An alternative notation would be to encode (protocol) startchart diagrams (e.g., Fig. 1) in a textual form. This might be more natural, especially for large classes with complex protocols. However, we chose regular expressions for a notational brevity and readability. We assumed that a well-designed class would have a small number of public methods and its public protocol would be simple enough to be written concisely in a regular expression. Our case studies, though limited, confirmed this. We also assumed that most programmers are familiar with a simple regular expression notation such as ours.

The other limitations are concerned with our semantic model and implementation. The semantic model that we proposed in this paper doesn't provide much help in checking protocol properties of a set of collaborating objects. In fact, the current implementation considers each object in isolation, as each object contains a separate finite state machine. In other words, a call sequence specification is interpreted as a per-object protocol for all objects of the class or interface in which the call sequence clause is written. For example, the following call sequence specification cannot be checked by the current implementation, though it is allowed syntactically.

```
public class JmlBinaryCallSequenceExpression
  extends JmlCallSequenceExpression {
  private JmlCallSequenceExpression left, right;
  /*@ private call_sequence typecheck()
    @   { left.typecheck() : right.typecheck() };
    @*/
  // other declarations such as typecheck ...
}
```

The call sequence clause specifies that the `typecheck` method of an expression should call the `typecheck` methods of left and right subexpressions. We found that this kind of specifications is common in the case studies, as an object is often composed of several component objects, and the inability to check such specifications limits the practicality of our approach. However, we note that most of such specifications describe internal (or private) protocols, as the component objects are hidden inside (or private to) the composite object. That is, the protocols constrain the implementor of the module, not its clients.

A similar problem exists between class objects and instance objects. We make a distinction between static call sequence specifications and non-static call sequence specifications. The static call sequence specifications constrain the order of static method calls, and the non-static specifications constrain the order of non-static (i.e., instance) method calls. A non-static call sequence clause may have a static method call, but such a call sequence specification cannot be checked by the current implementation, as it involves more than one object, i.e., an instance object and a class object.

For the practical use of our tool, time and space efficiency is a big concern. In the current implementation of our extension to the JML compiler, each method call incurs at least three additional method calls for checking call sequence specifications (see Section 5.3). If we consider functional specifications such as invariants and pre and postconditions, the number of additional method calls increases dramatically. This is particularly true in the presence of class hierarchy because a subclass' assertion checking methods call the superclass' assertion checking methods to inherit specifications (see Section 5.2). Worse, the JML compiler implements these delegation calls by using Java's reflection facility, and one study showed that a reflection-based delegation is 2 or 3 times slower than a non-reflection-based approach (Cheon and Leavens, 2002). Although we didn't pay much attention to the performance of our implementation or measure it accurately, our experience so far indicates that code with runtime checks is considerably slower than the same code without runtime checks.

Finally, our approach only applies to sequential programs. This is partly because JML currently only deals with sequential programs, though there are some research efforts to extend JML to support effective specification of multithreaded programs (Rodríguez et al., 2005).

Addressing aforementioned limitations is the most important future work, in addition to optimizing the finite state machine to improve its time and space efficiency. We are considering to extend our semantic model by introducing timestamps. A transition may be modeled as a tuple of an object identifier, an action, and a timestamp. A sequential program can now be modeled as a set of (collaborating) objects, where each object has its own time-stamped history. The state of a program execution is given by the histories of all objects in the system. We hope this extension facilitates checking not only inter-object protocol specifications but also multithreaded programs.

7 Related Work

Meyer pioneered design by contract (DBC) in the programming language Eiffel by integrating executable assertions into programs in the form of pre and postconditions and class invariants (Meyer, 1992a,b, 1997). However, Eiffel does not provide a built-in facility to specify and check protocol properties. As in JML, the protocol properties have to be encoded in pre and postconditions and in-line assertions. An in-line assertion is a specification statement that can appear inside a method body, such as the loop invariant and the `set` statement.

Eiffel contributed to the availability of similar DBC facilities in other programming languages. For example, there are several DBC tools for Java (Bartetzko et al., 2001; Duncan and Holzle, 1998; Findler and Felleisen, 2001; Karaorman et al., 1999; Kramer, 1998). The approaches vary widely from a simple assertion mechanism similar to the `assert` macro of C and C++ to full-fledged contract enforcement tools. Except for Jass (Bartetzko et al., 2001), however, none of the aforementioned approaches supports protocol specifications.

Jass (Bartetzko et al., 2001) 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. The trace assertions are interpreted loosely; however, no formal semantics is provided. The Jass precompiler translates the trace assertions into runtime checks. An alternative approach called Jassda (Brörkens and Möller, 2002b,a) checks trace assertions by observing the events generated by debuggers through the Java Debug Interface (JDI). An obvious shortcoming of this alternative is that the target program must run in the debugging mode.

Ada annotation languages, e.g., Anna (Luckham, 1990) and SPARK (Barnes, 2003), do not support protocol specifications.

A more recent initiative, Spec# (Barnett et al., 2005), 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 (Soundarajan and Fridella, 2000), and the earliest work can be traced back to Bartussek and Parnas work on trace assertions (Bartussek and Parnas, 1978).

8 Conclusion

The Java Modeling Language (JML) has been extended to formally specify protocol properties of Java program modules such as classes and interfaces. The extension allows one to separate cleanly the ordering dependencies among method calls and to specify them explicitly. Without this extension, one has to mix protocol properties with functional properties in pre and postconditions by encoding them as finite state machines. Writing such specifications is laborious and error-prone, and the resulting specifications are hard to read and understand, as the protocol properties have to be inferred. The separation of protocol properties also produces cleaner and easy-to-read functional property specifications. The beauty of our approach is its notational brevity and simplicity, as we adapted a familiar, regular expression-like notation for writing protocol specifications.

The JML compiler has been extended to translate protocol specifications into runtime checks. Runtime checking is transparent in that, unless a protocol violation is detected, the behavior of original program is unchanged except for time and space measurements. The inheritance of protocol specifications has also been implemented by using a delegation approach in which a subtype delegates to its

supertypes the runtime checks of inherited specifications. A subtype has to satisfy all the inherited protocol specifications. We believe that our implementation is sound with respect to the formal semantics of protocol specifications.

Case studies identified a few limitations of our implementation. However, we believe that our extension provides an effective way to specify and check protocol aspects of reusable classes and application frameworks. Our extension may also be useful for testing classes—e.g., it may be possible to automatically generate a sequence of method calls from the protocol specification to test a class.

Acknowledgement

This work was supported in part by the National Science Foundation under grant CNS-0509299 and by the University of Texas at El Paso under URI grant 14-5078-6151. Thanks to Myoung Kim and anonymous referees for comments on earlier drafts of this paper.

Bibliography

Agarwal, P., Rubio-Medrano, C. E., Cheon, Y., and Teller, P. J. (2006). A formal specification in JML of the Java security package. Technical Report 06-13, Department of Computer Science, The University of Texas at El Paso.

Barnes, J. (2003). *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, New York, NY.

Barnett, M., Leino, K. R. M., and Schulte, W. (2005). The Spec# programming system: An overview. In Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., and Muntean, T., editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag.

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

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

Brörkens, M. and Möller, M. (2002a). Dynamic event generation for runtime checking using the JDI. In Havelund, K. and Rosu, G., editors, *Proceedings of the Federated Logic Conference Satellite Workshops, Runtime Verification, Copenhagen, Denmark*. Electronic Notes in Theoretical Computer Science 70(4).

Brörkens, M. and Möller, M. (2002b). Jassda trace assertions, runtime checking the dynamic of java programs. In Schieferdecker, I., König, H., and Wolisz, A., editors, *Trends in Testing Communicating Systems, International Conference on Testing of Communicating Systems, Berlin, Germany*, pages 39–48.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232.

Cheon, Y. (2003). A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA. The author's Ph.D. dissertation.

Cheon, Y. and Leavens, G. T. (2002). A runtime assertion checker for the Java Modeling Language (JML). In Arabnia, H. R. and Mun, Y., 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.

Cheon, Y., Leavens, G. T., Sitaraman, M., and Edwards, S. (2005). Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice & Experience*, 35(6):583–599.

Cheon, Y. and Perumendla, A. (2005). Specifying and checking method call sequences in JML. In Arabnia, H. R. and Reza, H., editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume II, Las Vegas, Nevada, June 27-29, 2005*, pages 511–516. CSREA Press.

Duncan, A. and Holzle, U. (1998). Adding contracts to Java with Handshake. Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara, CA.

Findler, R. B. and Felleisen, M. (2001). 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*, pages 1–15.

Karaorman, M., Holzle, U., and Bruno, J. (1999). jContractor: A reflective Java library to support design by contract. In Cointe, P., editor, *Meta-Level Architectures and Reflection, Second International Conference on Reflection '99, Saint-Malo, France, July 19–21, 1999, Proceedings*, volume 1616 of *Lecture Notes in Computer Science*, pages 175–196. Springer-Verlag.

Kramer, R. (1998). iContract – the Java^{TM} design by contract^{TM} tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE Computer Society Press.

Leavens, G. T. and Baker, A. L. (1999). Enhancing the pre- and postcondition technique for more expressive specifications. In Wing, J. M., Woodcock, J., and Davies, J., editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag.

Leavens, G. T., Baker, A. L., and Ruby, C. (1999). JML: A notation for detailed design. In Kilov, H., Rumpe, B., and Simmonds, I., editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston.

Leavens, G. T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., and Kiniry, J. (2005). The JML reference manual. Available from `www.jmlspecs.org` (Date retrieved: October 31, 2005).

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

Meyer, B. (1992a). Applying "design by contract". *Computer*, 25(10):40–51.

Meyer, B. (1992b). *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY.

Meyer, B. (1997). *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition.

Rodríguez, E., Dwyer, M. B., Flanagan, C., Hatcliff, J., Leavens, G. T., and Robby (2005). Extending JML for modular specification and verification of

multi-threaded programs. In Black, A. P., editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer-Verlag, Berlin.

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

Soundarajan, N. and Fridella, S. (2000). Framework-based applications: From incremental development to incremental reasoning. In Frakes, W. B., editor, *Software Reuse: Advances in Software Reusability, 6th International Conference, ICSR-6, Vienna, Austria, June 27-29, 2000, Proceedings*, volume 1844 of *Lecture Notes in Computer Science*, pages 100–116. Springer-Verlag.