Compiler and Tool Support for Debugging Object Protocols

Technical Report OSU-CISRC-3/00-TR10

Sergey Butkevich* Marco Renedo* Gerald Baumgartner* Michal Young**

* Dept. of Computer and Information Science
The Ohio State University
395 Dreese Lab., 2015 Neil Ave.
Columbus, OH 43210-1277
{butkevic, rened, gb}@cis.ohio-state.edu

** Dept. of Computer and Information Science
University of Oregon
120 Deschutes Hall
Eugene, OR 97403-1202
michal@cs.uoregon.edu

March 17, 2000

ABSTRACT

We describe an extension to the Java programming language that supports static conformance checking and dynamic debugging of object “protocols,” i.e., sequencing constraints on the order in which methods may be called. Our Java protocols have a statically checkable subset embedded in richer descriptions that can be checked at run-time. The statically checkable subtype conformance relation is based on Nierstrasz’ proposal for regular (finite-state) process types, and is also very close to the conformance relation for architectural connectors in the Wright architectural description language by Allen and Garlan. Richer sequencing properties, which cannot be expressed by regular types alone, can be specified and checked at run-time by associating predicates with object states. We describe the language extensions and their rationale, and the design of tool support for static and dynamic checking and debugging. Tool support for static (compile-time) protocol conformance checking has been implemented as an extension to the compiler of Sun Microsystems’ Java Development Kit, Release 1.1.7, and the dynamic checking and debugging is currently in progress.

1 INTRODUCTION

A repeated pattern in the history of software engineering research is development of underlying principles for specifying certain properties, then development of specification formalisms and automated checks for some part of those properties, and then migration of some efficiently checkable part of those specifications to programming languages. This pattern can be seen in abstract data types, eventually (but only partially) realized in module and class constructs of modern languages, and in module interconnection specifications which likewise were developed first as extrinsic specifications but are now at least partly internalized in the “package” constructs of Java and Ada. Since there is a long thread of research in specifying the sequences of operations accepted at module interfaces [8], and more recently development of extrinsic specifications of operation sequence protocols in architecture description languages [2, 3, 11] as well as the StateChart part of UML [4, 17], it is natural to consider whether and to what extent such protocols can be incorporated
directly into programming languages and checked routinely as a part of normal compilation. Recent research in programming language design and semantics has greatly widened the class of interface properties that can be captured as part of type compatibility, and Nierstrasz has shown in principle how operation sequencing can be treated in a type system [12], but to date investigations of protocols as object types have been limited to pencil-and-paper exercises. In this paper, we describe an extension to the Java programming language which supports static protocol conformance checking and dynamic checking of compatibility between actual and declared behavior. The static checking has been implemented as part of an extended Java compiler.

1.1 Protocols as Part of Types

The interface specifications described here combine concepts of access-right expressions, originally described by Kieburtz and Silberschatz [8] with the regular object types of Nierstrasz [12]. They are interface specifications, distinct and independent from mechanisms such as path expressions [3], which might be used to implement the synchronization to enforce a particular pattern of operations. Similar to Liskov and Wing’s notion of behavioral subtyping [10], we extend the subtype relationship with behavioral information. Interface specifications are related to architectural description languages (ADLs) such as Wright [2, 3] and Darwin [11]. But while ADLs are language independent and capture higher-level architectural structures, our interface specifications are language specific, which allows some static checking and enables the compiler to generate code for dynamic monitoring. Our approach is partly based on Nierstrasz’s regular types for active objects [12]. Similar formal models have been developed for concurrent objects with asynchronous message passing [15].

The type or interface of a class specifies a set of operations or methods provided by a class. Often these methods can be called only in a particular order, but the order is not part of the interface and cannot be checked by a Java compiler. (The situation is similar for other strongly-typed, object-oriented languages.) The well-known benefits of static type checking are thus available for such properties as the number and order of arguments to each method, but not for the sequencing of method calls. Protocols add this sequencing information to class and interface declarations and allow a compiler to check whether the declared intent of an object making method calls is compatible with the sequences supported by the object being called.

An extension of Ada that employs behavioral subtyping and is similar in some ways to our approach has been proposed by Puntigam [16]. Our approach differs in two fundamental ways: First, we treat a protocol as a contract between individual objects, whereas Puntigam’s behavioral types specify what a set of objects may do collectively. Second, Puntigam’s proposal is for static verification of actual behavior through program analysis; our more modest and, we think, more practical approach combines static verification of declarations with dynamic checking of actual behavior. In addition, our protocol specifications are somewhat more expressive, supporting non-determinism that cannot be expressed in Puntigam’s behavior specifications.

1.2 Debugging Support for Protocols

Since compile-time checks are limited to checking regular (finite-state) specifications and are not, in general, capable of determining whether actual run-time behavior is consistent with these declarations of intent, additional checking is necessary at run-time. To allow this run-time checking, the compiler instruments the generated code such that the run-time behavior is communicated to a debugging tool that compares the dynamic behavior with the declared behavior and either logs protocol violations or generates run-time errors. For checking the method call sequence, the debugging tool employs a labeled transition system, in which each method call triggers a state transition.

2 Language Design

2.1 Formulation of the Problem

Assume we are given a class RandomAccess implementing some interfaces DataOutput and DataInput.

    class RandomAccess implements DataOutput, DataInput {  
        // ...
Now assume that a client of class RandomAccess contains the following piece of code.

```java
// ...
DataInput file = new RandomAccess();
file.open();
x = file.read();
file.close();
y = file.read();
// ...
```

This code will compile without errors or warnings. However, it is clearly not what was meant by the author of class RandomAccess. A client should not read from a file after it has been closed. What is missing in the source code is a description of the order in which the methods of a class or an interface must be called.

### 2.2 Protocol Declarations

We introduce a new language construct\(^1\), a protocol declaration, or, briefly, a protocol. A protocol declaration can appear in an interface or in a class. Syntaxically, a protocol is introduced by the keyword "protocol" and contains a block of protocol statements. (We are using double quotes to denote literals and symbols.) Unlike methods, classes, and interfaces, a protocol does not have a name but is associated with its enclosing class or interface.

In the simplest case, a protocol contains just a single regular expression over the alphabet of all public method names. For the interface DataInput the protocol might be:

```java
interface DataInput {
    protocol { open, read*, close; }
    // ...
}
```

This means that an object of a class that implements this interface is allowed to call the method open once, then call the method read zero or more times, and then call the method close before being destroyed (garbage-collected).

A reasonable protocol for the class RandomAccess would be the following:

```java
class RandomAccess implements DataOutput, DataInput {
    protocol { open, (read|write)*, close; }
    // ...
}
```

The latter protocol allows more functionality than the former. We say that the protocol of class RandomAccess conforms to that of interface DataInput. An object \( X \) conforms to an object \( Y \), if \( X \) is \textit{request substitutable} for \( Y \). I.e., if a client of \( Y \) expects \( Y \) to accept a sequence of requests \( s \), and we substitute \( X \) for \( Y \), then \( X \) will accept the same sequence \( s \). (A more formal definition of the notion of conformance will be given later, when we describe more general types of protocols.)

The conformance relation is a partial ordering among types. It has to be consistent with the subtype relation, i.e., if a class or interface \( X \) is a subtype of another class or interface \( Y \), then the protocol of \( X \) must conform to the protocol of \( Y \). Otherwise, the compiler should generate an error. If an interface or class \( X \) has no protocol declaration, the default protocol is assumed — i.e., methods of such a class can be called in any order. Such a protocol represents a minimal element with respect to our conformance relation, i.e., it conforms to any other protocol. If we use the symbol \( \prec \) to mean "conforms to," then we have:

\[
\text{Default} \prec \text{RandomAccess} \prec \text{DataInput}
\]

\(^1\)In the initial version we chose to extend the syntax directly. A future version may encapsulate the construct in a formal JavaDoc comment, as done in iContract [9].
where Default denotes a default protocol.

Can the allowed sequences of operations always be expressed as a single regular expression? The following example shows that, unfortunately, this is not possible. Consider a simple interface for a stack.

```java
interface Stack {
    public void push(int i);
    public int pop();
}
```

We would like to write a protocol for this class that would allow sequences of requests such as \(\text{push, pop}\) or \(\text{push, push, pop}\), but disallow, for example, the sequence of requests \(\text{push, pop, pop}\). A regular expression or a deterministic finite automaton (DFA) cannot do that since it cannot keep track of the number of elements on the stack. Other finite-state specifications share the same fundamental limitation in expressiveness. Thus, we would need a richer language, such as a context-free grammar. The conformance check for context-free languages, however, is undecidable. Following the idea introduced by Nierstrasz [12], we use a labeled transition system (LTS) over the alphabet of all public methods of a class or interface to describe the protocols, which, in general can be non-deterministic. This allows writing protocols that represent a reasonable approximation for possible object behaviors and, at the same time, are simple enough such that the conformance check can be performed at compile time.

Using this approach, we can write the protocol for the interface Stack as follows:

```java
protocol {
    start final state e;
    final state ne;
    <*> push <ne>;
    <ne> pop <*>;
}
```

Figure 1 shows the LTS defined by this protocol. Clearly, this protocol is only an approximation for the stack behavior. Internal non-determinism (as opposed to external non-determinism) is introduced as an artifact of modeling, i.e., deterministic choices of the service are modeled as arbitrary choices. It is for this reason that the protocol must be modeled as a labeled transition system (LTS) with failure semantics, and not as a language acceptor in which non-determinism can be removed by transformation to a deterministic finite-state acceptor using the subset construction. Because internal non-determinism is an artifact of abstraction in the finite-state model and not a feature of the actual system, these same internal choices are interpreted differently in run-time checks (described below).

A formal protocol syntax specification is given in Figure 2. Below, we outline its main features. Some details were intentionally left out for the sake of brevity.

A protocol declaration consists of a series of protocol statements. Each protocol statement is either a state declaration, a regular expression declaration, or a sequencing statement.

### 2.3 State Declarations

A state declaration declares one or more state identifiers that subsequently can be used in sequencing statements. Final states can be identified with the modifier "final". The start state can be identified with the modifier "start". Each state identifier is followed by an optional "=" sign followed by a boolean expression, which represents a state predicate. Its meaning will be explained later. There are two implicitly defined states — the default start state and the default final states that are represented by empty state expressions on the left side and on the right side of a sequencing statement, respectively.
\begin{verbatim}
(ProtocolDeclaration) ::= "protocol" "{" (ProtocolStatement) "}" \\
(ProtocolStatement) ::= (StateDec) | (RegExpDec) | (SeqStatement) \\
(StateDec) ::= ["start"] | ["final"] | "state" (JavaId) ["=" (JavaBoolExp)] \\
["," (JavaId) ["=" (JavaBoolExp)]] \\
(RegExpDec) ::= "regexp" (JavaId) ["=" (RegExp)] {"," (JavaId) ["=" (RegExp)] \\
(SeqStatement) ::= ["<" (StateList) ">"] (RegExp) ["<" (StateList) ">"] \\
(StateList) ::= "*" | (JavaId) {"," (JavaId)} \\
(RegExp) ::= (MethodCallPattern) \\
["~"] (RegExp) ["~"] (MethodCallPattern) ["~"] (RegExp) \\
(MethodCallPattern) ::= (JavaId) ["(" (PatternArgumentList) ")"] \\
(PatternArgumentList) ::= (PatternArgument) {"," (PatternArgument)} \\
(PatternArgument) ::= "*" | (JavaType)
\end{verbatim}

Figure 2: Protocol Grammar Definitions

In the Stack example above, we defined two states — e and ne (corresponding to empty and non-empty states of the stack). Both states are final, which means that an object implementing this interface is allowed to be destroyed at every state. In the example of DataInput there are no explicitly defined states.

2.4 Regular Expression Declarations

A regular expression declaration defines one or more names for regular expressions. This might be thought of as a macro definition and might be useful when writing complex protocols.

2.5 State Lists

A state list is either the literal "*" or a list of one or more identifiers separated by commas. The literal "*" is interpreted as the list of all explicitly declared state identifiers.

2.6 Sequencing Statements

In the simplest case, a sequencing statement is just a regular expression (as in the DataInput example).

More generally, a sequencing statement consists of an optional state list, followed by a regular expression over the alphabet of public method names, another optional state list, and a semicolon.

A sequencing statement defines state transition in the LTS defining the protocol. An individual regular expression describes a language of allowed sequences, and the appropriate semantics for this is language acceptance (also called trace semantics). There is no internal non-determinism, so we can use the standard subset construction [1, p. 117] to represent each individual regular sequencing statement as a deterministic acceptor, while still maintaining the failure semantics of the protocol LTS as a whole.

If the left-hand side (LHS) and right-hand side (RHS) state lists specify only one state each, the start state of the DFA is the LHS state, and the final state of the DFA is the RHS state. An empty LHS represents the default start state. An empty RHS represents the default final state. If there are multiple LHS states, multiple RHS states, or both, the sequencing statement is equivalent to a series of sequencing statements with the same regular expression and all possible LHS-RHS state pairs. The protocol LTS is constructed by connecting the DFAs resulting from individual sequencing statements.

If a public method of a class or interface is not mentioned in any sequencing statement or regular expression definition, it is assumed that no restrictions are imposed on its use. In other words, not mentioning
a public method \texttt{foo} in the protocol declaration is equivalent to including the following sequencing statement for each state \( s \):

\[
<s> \texttt{foo} <s>;
\]

### 2.7 Regular Expressions

We use a conventional syntax for regular expressions except that the comma operator corresponds to concatenation in lex-style regular expressions. A vertical bar represents a choice between two subprotocols. The operators "\*", "\+", and "\?" denote zero or more, one or more, and zero or one occurrences of the regular factor, respectively. A list of method call patterns between brackets is equivalent to the same list of patterns separated with vertical bars. A bracketed list of patterns with the literal "\~" in front denotes the list of all possible method call patterns of any public method of the class or interface except the ones listed. Parentheses are used to group terms together.

The simplest method call pattern, an identifier \texttt{foo}, indicates that any public method \texttt{foo} can be called by a client. By providing types as arguments, a smaller set of methods out of the set of all overloaded methods can be selected. The literal "\*" inside a method call pattern acts as a wild card.

### 2.8 Conformance

How should a conformance relation between two protocols be formally defined? We employ the principle of request substitutability introduced in [12]. Protocol \( X \) conforms to protocol \( Y \) if all sequences of requests supported by \( Y \) will be also supported by \( X \) and, moreover, that any request refused by \( X \) after accepting one of those sequences might also have been refused by \( Y \). More formally, \( X \prec Y \) if

1. \( \text{traces}(Y) \subseteq \text{traces}(X) \)
2. \( \text{failures}_Y(X) \subseteq \text{failures}(Y) \).

Without non-determinism, the condition (2) is redundant, but if internal non-determinism is present, as it is the case in our \texttt{Stack} example, then it is necessary to check both conditions.

As an example, assume we have an interface for an uninitialized variable that has two public methods, which we also call \texttt{push} and \texttt{pop}.

```java
interface Var {
    protocol {
        final start state e;
        final state ne;
        <e> push <ne>;
        <ne> pop | push <ne>;
    }
    public void push(int i);
    public int pop();
}
```

The protocol of interface \texttt{Var} allows more freedom than that of \texttt{Stack}, and we would expect that \texttt{Var} conforms to \texttt{Stack}, but not vice versa. Note though that \( \text{traces(Var)} = \text{traces(Stack)} \), so we can not distinguish between the two protocols by their traces only. However, if we compare the failure sequences, the difference between the two protocols becomes clear: The protocol of \texttt{Var} will always accept the call sequence \texttt{push, pop, pop}, whereas the protocol of \texttt{Stack} \emph{might} not. This means that the set of the relative failures of \texttt{Stack} with respect to \texttt{Var} is not a subset of the failure set of \texttt{Var}. Hence, \texttt{Var} \( \prec \) \texttt{Stack}, but \texttt{Stack} \( \not\prec \) \texttt{Var}.

![Figure 3: The LTS for interface \texttt{Var}](image)
In [12], an algorithm for conformance checking between two LTSs was given. We use this algorithm as part of the type checking phase of the compiler. If a class/interface Y extends/implements class/interface X, then the protocol of Y must conform to that of X. Otherwise, a compilation error is reported. For example, if we declare interface Stack as extending interface Var with the protocols described above, we will receive a compilation error saying that the protocol of Stack does not conform to that of Var.

There is a serious deficiency in describing protocols with finite-state LTSs — they are only approximations of real protocols, as seen in the Stack protocol. This protocol does not rule out the sequence of calls push, pop, pop. It only tells that it might fail. As we noted earlier, if we tried to specify protocols more precisely, we would not be able to perform the conformance check during compile time and their language would become too complicated for them to be useful. However, we can do better at run time by attaching predicates to the states, choosing among branches of the LTS at run-time.

Part or all of the internal non-determinism in protocol specifications, which plays a role in static conformance checking, is removed by evaluating the predicates at run-time. The remaining non-determinism is interpreted as external choice. While we use failure semantics for static checks of the conformance relation between declared protocols, language (trace) acceptance is the appropriate semantics for run-time checks of consistency between actual behavior and declared intent.

2.9 State Predicates

A state predicate is a Java boolean expression that is an optional part of a state declaration and is associated with a state. It is stored in the LTS and is evaluated at run-time to choose between several non-deterministic transitions in the LTS. A state predicate has class scope (syntactically this is the same as an initializer of a class field).

As an example, we can add another method isEmpty() to our Stack interface and rewrite the interface as follows:

```java
interface Stack {
    protocol {
        start final state e = isEmpty();
        final state ne = !isEmpty();
        <ne> pop <*>;
        <+> push <ne>;
    }
    public void push(int i);
    public int pop();
    public boolean isEmpty();
}
```

The state predicates are only used at run-time. They have no effect on the compile-time conformance check. Moreover, it is not possible for the compiler to check whether they are reasonably implemented and not self-contradictory. However, they provide essential information for debugging.

2.10 Debugging

For demonstrating the use of debugging, consider the following example (from some file foo.java) in which the protocol of interface Stack is violated by calling the method pop on an empty Stack:

```java
Stack a = new StackImplementation();   // line 12
a.push(3);
int x = a.pop();
int y = a.pop();   // line 15
```

Since, in general, the compiler cannot detect protocol violations as on line 15, we provide run-time debugging support to detect such protocol violations.

There are two main design issues involving debugging. First there is the problem of how to implement the LTS tracing so it can be used in already existing code, and second what action should be performed when a protocol violation is detected.
With respect to the first problem, one alternative would be to modify the Java Virtual Machine so it traces the LTS. A second alternative would be to modify the compiler so it inlines additional functionality in the client code. The approach we adopted is to introduce a Wrapper (as in the Decorator design pattern [6]) in line 12, between reference a and the StackImplementation object. In this way, any time there is a method call to object a, the wrapper object can trace the protocol (perform an LTS transition) as a side effect, while calling the same method on the object of class StackImplementation. We can automate this modification and make it invisible to the user by modifying the compiler. For every assignment in which the left-hand side type is an interface type with a protocol, the compiler inserts a Wrapper constructor call on the right hand side. This approach has significant run-time overhead but the user does not need a special Java Virtual Machine or compiler to take advantage of this functionality. In any case, this aspect of the implementation strategy is mostly independent of the overall approach to specifying and checking object protocols.

With respect to the second problem, only the user knows exactly what to do in case of a protocol error. For maximum flexibility, we provide a mechanism for selecting the error handling behavior. Following the Strategy design pattern [6], we provide an interface ErrorHandler, in which each method corresponds to a possible type of protocol violation, and allow the user to select an appropriate implementation of this interface. We provide standard error handler implementations for logging protocol violations and for raising run-time exceptions. Using a simple API users can write custom error handlers.

Run-time tracing of the protocol can serve at least four purposes. It is up to the user to decide what role protocols should play in the debugging process:

- **Finding errors in the client's implementation.** This is potentially the most useful application of the run time checking of protocols. If there is a misuse of a class or interface, it will be reflected in a violation of the protocol of that class or interface and the user will be able to detect this violation by tracing its protocol.

- **Debugging the protocol.** The user can write test harnesses and check if the protocol behaves as intended via inspection of the LTS at run time.

- **Using exceptions to control the application.** By using an error handler that throws exceptions in case of a protocol violation and by catching these exceptions in the client, the LTS can be (mis)used as part of the control of the application.

- **Finding errors in the server's implementation.** Using an interface with the same protocol as the class, it is possible to build a test harness for the class such that protocol violations indicate errors in the class.

3 Implementation

This section describes the changes we made or plan to make to the Java Development Kit, Release 1.1.7 [18] for implementing compile-time and run-time support for protocols. The compile-time conformance check is implemented and fully functional, also in the case of separate compilation. In particular, all the examples that we consider in the text would compile with our modified Java compiler. However, we did not yet implement support for method signatures in method call patterns, negation in regular expressions, and regular expression (macro) definitions. The run-time debugging support is also fully functional, but currently the user must manually write the code for defining and instantiating the wrapper classes. We are working on extending the compiler so the code is generated when compiling the interface and the client, respectively.

3.1 Compile-Time Implementation

The compile-time implementation consists of four main parts

1. **Parsing protocols.** We modified the javac compiler so that it recognizes the new keyword protocol, parses protocols, creates a parse tree for each protocol, and reports syntax errors.
2. **Semantic analysis.** In this stage we generate an LTS from the parse tree. Each regular expression is first translated into an NFA using Thompson’s construction and then this NFA is translated into a DFA using the subset construction [1, pp. 122, 117]. Then a protocol LTS is built by connecting these individual DFAs. This automaton is non-deterministic, in general. We do not convert it to a DFA, since the conformance relation that we use is not preserved under such conversion. The protocol LTS has two types of states — the states that were generated as a result of conversion of regular expressions to DFAs and the states that were explicitly defined by the programmer in the protocol. The data structure representing the latter type of states contains extra information, such as state predicates and source file line and position number.

3. **Conformance checking.** If a class or interface extends or inherits another class or interface and both of them have protocols, the compiler performs the conformance check between the protocols. For this conformance check, we use the algorithm proposed by O. Nienstrasz [12]. Since in [12] it is assumed that all states of the LTS are final, we made a simple modification to the algorithm so that it can compare two LTSs that possibly have non-final states. The running time of this algorithm is exponential over the number of states that have non-deterministic edges in the worst case. However, in a typical case, it is much faster. Note that we check the conformance of the protocol LTSs without taking state predicates into account (since the task of conformance checking would become undecidable otherwise). If the protocol of the subtype does not conform to the protocol of the supertype, a type error is reported.

4. **Storing protocols in binary code.** If the conformance check was successful, then a representation of the protocol LTS is stored in the class file as a user-defined class attribute. This allows performing the conformance check between protocols from different source files, which is necessary for separate compilation. The class file so created is readable by a standard Java compiler and by a standard Java virtual machine. However, only a modified compiler is able to read the protocol LTS back from the class file.

### 3.2 Run-Time Implementation

When a piece of code (the *client code*) assigns an object (the *server*) to a reference, and this reference has as type an interface with a declared protocol, the tool initializes an LTS that corresponds to that protocol. For example, if file *foo.java* contains the line:

```java
I a = new C();
```

then an LTS is initialized when the client code *foo.java* assigns the server *new C()* to the reference *a*. The initialized LTS corresponds to the protocol of interface *I*. The simulation of the LTS is stopped when the reference is garbage collected.

*The main purpose of the run-time tool is to detect violations of the protocol specified in the interface when calling a method on a variable of the interface type.*

Every time the client code calls a server method through the reference, the tool checks if any of the possible current states of the LTS allows that method call. If so, a state transition is performed in the LTS and the server method is executed. After the server returns from executing the method, the LTS states for which the corresponding state predicate is false are removed from the set of possible states.

**Implementation Overview**

A *Wrapper* object is inserted between the reference and the server in the statement where the assignment occurs. The client code of the previous example is modified as follows:

```java
I a = new Wrapper(new C(),...);
```

---

2Recall that the regular expression parts of protocol specifications describe language acceptance, and involve no internal non-determinism; converting them (individually) to deterministic automata thus gives the correct semantics in the overall LTS representation.
In this way, any call to a method of the server through the reference first has to go through a method call on the Wrapper object. So far this insertion has to be done manually but we are working on modifying the compiler so the insertion will be done automatically when the client code is compiled. The Wrapper class is tailored to the interface, that is, for each interface I there is an IWrapper class. We are working on modifying the compiler so the wrapper class will be created automatically when compiling the interface. The common code among the wrappers is contained in the superclass Tracer. Additional data structures are the ProtocolInformation class, used to store the specification of an LTS, and the TraceState class, used to store the state of an LTS. Sometimes reporting is desirable after the reference has been garbage collected, so a list of TraceStates is kept separately for that purpose. See Figure 4 for a UML diagram of the class hierarchy.

Below we explain in more detail the components of the run-time implementation.

Data Structures

The data structures involved in the run-time implementation are:

- Several classes IWrapper, one for each interface I that contains a protocol declaration. The class IWrapper will be created by the compiler when compiling the interface I.
- Class Tracer: an instance of this class simulates an LTS. It is a superclass for the Wrapper classes.
- Interface ErrorHandler, which can be implemented by the user to control protocol error handling and reporting. Some standard error handler classes are provided, which can be extended by the user.
- Class TraceState: an instance of this class holds the current state and the history of an LTS.
- Interface TraceFilter, which can be implemented by the user to select TraceStates currently in memory that satisfy specified conditions.
- Class ProtocolInformation: an instance of this class holds the specification of an LTS at run time.

Class Wrapper

The main role of this class is to enrich any method call to the server object with operations to help trace the states of the LTS. A Wrapper contains code specific to a particular interface. Everything else is implemented in class Tracer, which is a superclass of the Wrappers.

The Wrapper also creates the run-time description of the protocol as a static object of type ProtocolInformation.

Assume that class StackImpl implements the interface Stack defined above, and that line 15 of file foo.java contains the following assignment:

```java
Stack a = new StackImpl();
```

When compiling file foo.java, our compiler will (eventually) replace the assignment by

```java
Stack a = new Stack_Wrapper(new StackImpl(), this, "foo.java: line 15.");
```

The class Stack_Wrapper would have been generated previously by the compiler when the interface Stack was compiled.
public class Stack_Wrapper extends Tracer {
    public Stack_Wrapper(Stack server, Object client, String lineAndFile) {...
    public void push(int i) {...
    public int pop() {...
    public boolean isEmpty() {...
    public boolean verify() {...
}

The I_Wrapper class for an interface I implements every method of that interface as a sequence of three method calls:

1. A call to the method announce (implemented in class Tracer) to check if the method is valid, i.e., if there are any edges with that method's name out of any of the current states of the LTS.

2. A call to the corresponding method of the server.

3. A call to method advance (also implemented in the Tracer) to perform the transition in the LTS to the new states, and to check state predicates.

Method push() of the previous example is implemented as

    public void push(int i) {
        announce(METHOD_PUSH); // push is being called
        server.push(i); // call method on server
        advance(METHOD_PUSH); // advance to next states
    }

The method verify() removes all the states that do not satisfy their state predicates from the list of current states. This method is interface dependent because it evaluates the state predicates.

Class Tracer

Class Tracer is the simulator of the LTS. The algorithm used to simulate the LTS involves three phases:

- **Announce phase.** The Wrapper announces to the Tracer that a method is going to be called. The Tracer checks if any of the current states allows that method call. If there are none we say that the method is invalid.

- **Advance phase.** After the method has been called the Tracer computes the list of new current states by finding the states that we can jump to from the old states by calling that method. Then the Tracer takes off this list all the states that do not satisfy their state predicate. If the list is now empty we say that the state is invalid.

- **Finalize phase.** This phase occurs when the Wrapper is garbage-collected. One would like to detect if the protocol terminated in a final state or not. According to the Java Language Specification [7] any method finalize() implemented in a class is always called by the virtual machine when an object of this class is about to be garbage collected. We employ this feature and insert an appropriate algorithm in the method finalize() of the Tracer so that when the Wrapper is garbage-collected, it checks if any of the current states is a final state. If none of the current states is a final state we say that the protocol is not in a final state.

To allow users to query the state of a protocol at run time, class Tracer maintains a static list of protocol states, with an object of class TraceState per wrapper, and provides a general mechanism to collect and filter information from this list (see Figure 5).
Class **TraceState**

A **TraceState** object must contain the information necessary to provide a full report of the state of a protocol even after the **Wrapper** and its **Tracer** part have been garbage-collected. It should contain, for example, the last method calls performed, a reference to the protocol specification, error flags and the class names of the server and client, as well as the current states the LTS might be in.

The current states of the LTS are represented as an array of type `boolean`: if the `boolean` at index `i` has value `true` then the LTS might be in state number `i`. A **TraceState** object also contains a reference to the **ErrorHandler** object that the **Tracer** uses any time it finds an error.

The information in a **TraceState** object is updated with each LTS transition. The information is queried by error handlers or if the users traverses the static list of trace states to dump the information.

Class **ErrorHandler**

The tool is designed to give maximum flexibility as far as error handling is concerned. We apply the **Strategy** design pattern [6]d in this situation. When the **Tracer** encounters an error it defers the error to an object that implements the **ErrorHandler** interface. The user can create classes that implement the interface **ErrorHandler** and pass an error handler object to the **Tracer** through the static method `Tracer.SetDefaultOptions(...)`. All Wrappers that are created from then on will defer errors to that **ErrorHandler** implementation until the next call to that method.

There are three kinds of errors that can be found by the **Tracer**:

- **Invalid method.** This error can happen during the announce phase. Typically it will by caused by an error in the interface protocol or in the client code.

- **Invalid state.** This error can happen during the advance phase. Typically it will be caused by an error in the server code.

- **Not in final state.** This error can happen in the finalize phase. There are no current states that are final states. Typically some method calls on the client code are missing to bring the protocol to a closure.

Class **ProtocolInformation**

A **ProtocolInformation** object stores the run-time description of a protocol. There is only one protocol information object per interface with a protocol. It is created statically by the corresponding **Wrapper**. This data structure also has to contain information about the interface and the protocol necessary for reporting errors: it contains the name of the interface, the names of the methods and the names of the states. For each state declared in the protocol, the declared name of that state is stored. States that were not declared are named with the line number and character position of the regular expression from which the state originates.
4 CONCLUSIONS

We have described an extension of Java with a protocol construct for specifying sequencing constraints on the order in which methods may be called. Protocols can be specified as part of a class definition or an interface declaration. We have extended the compiler of Sun Microsystem’s Java Development Kit, Release 1.1.7, to check the conformance of a class protocol to an interface protocol as part of the interface conformance type check.

We have also described the design of a debugging tool for testing the conformance of a client’s code to the protocol declared in an interface. The tool runs a labeled transition system (LTS) generated by the compiler from the protocol declaration. For every method call by the client, the LTS checks whether the method call is allowed according to the protocol. For specifying sequencing constraints that cannot be captured by an LTS, we allow associating predicates with states of the LTS. By testing these predicates at run time, object states can be mapped to LTS states.

The run-time debugging tool as we have described only allows testing the conformance of the client’s code to the protocol specified in the interface. The conformance of the class protocol to the interface protocol is checked at compile time. What is missing is testing the conformance of the class’s code to the declared class protocol. In future research we will explore the automatic generation of a test harness from the class protocol for testing this latter conformance.

Currently we combine static checking of protocols declarations with run-time checking of actual behaviors; there is no static analysis of code. While we believe that complete verification of protocol conformance through code analysis, as proposed by Puntigam [16], is likely to be too computationally expensive and too conservative to be useful, it is possible that program analysis may play a useful role in combination with dynamic checking. We will explore how data flow analyses, such as those described by Olender and Osterweil for checking sequencing constraints [13, 14], can be used to find some violations at compile time and reduce the amount of checking left for run time.

Acknowledgments

The effort of Michal Young was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0034. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References


