2012

Object protocols as a tool for debugging method call sequencing constraints

Ronald William Gilkey
Louisiana State University and Agricultural and Mechanical College

OBJECT PROTOCOLS AS A TOOL FOR DEBUGGING METHOD CALL SEQUENCING CONSTRAINTS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science in Systems Science

in

The Department of Computer Science

by
Ronald William Gilkey Jr
B.S., Louisiana State University, 2007
May 2012

Acknowledgements

I would like to give my sincere thanks to my advisor Dr. Gerald Baumgartner for his patience and guidance throughout the course of my research. His knowledge and experience in the field was a great resource during my research. I benefited from his instructive comments and his mentorship in the scientific approach.

My utmost gratitude goes to team members of The Brew Project for providing me with their prototype code for the tracing system.

I would like to thank my wife, Kate, for her constant support and encouragement throughout my graduate studies. I would also like to thank my parents – Diane, Sandy, and Ron – for their efforts is helping me to educate and better myself.

Table of Contents

List of Tables

List of Figures

Abstract

Clearly conveying and enforcing the proper ordering of method calls on objects has become a common problem among developers and interface designers. Without the ability of the compilation environment to enforce these constraints, programmers must rely on clear documentation being provided and diligence in programming to ensure that a proper sequence of operations is performed. Commonly, though, type-checking becomes the only tool to help support the correctness of operation sequences as API documentation rarely describes inter-object communications. Thus, the likeliness of producing erroneous and buggy software increases.

Object protocols provide a simple and straight-forward approach to solving this problem. They do so by providing simple grammatical constructs for expressing valid call sequences on objects. These constructs can be parsed by a compiler and then used for conformance analysis on subclasses and objects that implement interfaces. Once parsing is complete, the compiler will implement code in the target binaries for run-time enforcement of sequencing constraints.

This thesis builds upon the foundation provided by Butkevich et al. to provide the design of a comprehensive tool for utilizing object protocols within the Java compilation and runtime environments. In addition to the existing work, we provide language constructs that allow the enforcement of object protocols on classes and the designation of objects and interfaces to be traced programmatically. We will consider the problems that are introduced when protocols are enforced upon classes and discuss the strategy for resolving these issues.

We present the results of testing the object tracing framework to determine the impact of protocols on the performance of software. Baseline timings are drawn on the operations of data

types and then protocols are manually added to the code in the same manner that would be done by the compiler in a fully working implementation; including the implementation of the tracing framework. We show that the overhead introduced is minimal, making the run-time monitoring of protocols practical.

Along with support for basic sequencing constraints, we also show how other constraints, such as timing constraints, can be added to the framework with minimal adjustments. Such applications of constraints can help debug different environments, such as embedded, real-time, and near real-time environments.

1. Introduction

When Java emerged, it broke ground not only as a platform-independent language, but as a programming language that eliminated several classes of programming errors as well [7]. It handles common programming issues – such as strong type-checking, checks for invalid memory access requests, and garbage collection – on behalf of the programmer. Java provides developers with the tools to write more reliable, stable software. In order to support these features Java employs both static and run-time analysis. Static analyses are often comprised of type-checking, tree traversals, and attribute grammars paired with dataflow analysis to verify conditions at compile time. Some examples of the run-time analysis methods utilized are: checking null pointer references, implementing array bounds checks, and garbage collection of objects [2, 4].

What is common to the resolution to each of these issues is the practice of programmatically enforcing the formalisms of the problem. For example, ensuring that unused memory is freed and ensuring only certain data types can be used for certain operations are formalisms that require attention in software development to ensure they are followed. The ability to enforce these formalisms in the compiler and run-time environment is what allows for the prevention of many bugs in Java software development. Object protocols implement the formalism of the required sequence of operations performed on classes and interfaces, and several benefits arise from such functionality as noted by Butkevich et al. [12]:

- Errors can be found in client implementations
    - Clients that do not obey the corresponding server's protocol
    - Protocols that do not properly conform to those of their parent class and/or interface

- Protocols can be tested to ensure proper run-time behavior
- Protocol violations can be handled as prescribed by the implementer

1.1 Problem Definition

The RandomAccessFile class in the Java I/O package provides us with a simple, yet clear example of the problem. The class implements both the DataInput and DataOutput interfaces to support I/O operations that may occur at random positions in the file/byte stream. The logical (and intended) ordering of operations is to open the file (done during object construction), perform some number of – if any – a read and/or write operations, and then close the file. The compiler, however, will compile code like that shown in Figure 1 below into bytecode, leaving the error to be handled by the Java Virtual Machine (JVM) at run-time. A read operation should not occur after a close operation has been performed on the file (as noted by the documentation); however the compiler is unable to enforce this constraint.

```
/* ... */
RandomAccessFile rf = new RandomAccessFile("/tmp/protocols.txt");
byte datum = rf.read();
rf.write(datum << 2);
rf.close();
datum = rf.read();
/* ... */
```

Figure 1 – Sample problematic Java code

Compilers are unable to detect and report these problems because there are no mechanisms for representing the states of an object throughout its lifetime. Type systems allow for the detection of potentially meaningless code or invalid code. Such checks are generally based upon argument patterns, return types, and dynamic type resolution. While compilers leverage type systems as much as possible, there are still classes of errors that cannot be detected. Monitoring and reacting to method call sequences is an interesting class of problems that has not been solved by

current type system implementations. Object protocols provide a solution for the method call sequencing problem by offering developers the grammatical constructs and run-time mechanisms to trace objects throughout their execution.

In Java, object interactions can be seen in a client-server model. Client objects send requests to server objects in the form of method calls. This interface is clearly defined in terms of method signatures (the name, return type and number, order and type of arguments); however there is no determination as to whether the call being made is appropriate. This is where object protocols come into play. They intercept the method calls made on class and interface objects and ensure that the execution of the call would not violate the object's protocol.

Object protocols are presented as a member of class and interface types, originally described by Butkevich et al [12]. The protocol specification is extracted during the compilation phase and checked for conformance any interfaces implemented or classes that are derived from. The necessary code to intercept method calls is put in to the bytecode during the generation process. We adapt this work along with notions on tracing requests depicted by DeLine and Fähndrich [14] to construct a set of enhancements to the Java language that provide a rich debugging environment for monitoring the interactions objects in a Java program.

In this thesis, we demonstrate that:

- protocols are useful as a method of enforcing and debugging object method call sequences
- and the run-time monitoring introduced by protocols is efficient.

2. Language Support

Section 1.1 shows a clear example of the need for sequencing constraints on method calls. What is needed is a solution to the problem. Regular languages can provide us with the ability to define the structure of the expression needed to represent such sequences. We can see that the regular expression (read|write)*,close represents our acceptable sequence of operations on a RandomAccessFile object. Protocols leverage the use of REs and a labeled transition system (LTS) to represent the state of the object and govern access to method execution. This chapter is dedicated to describing the proposed implementation of protocols in Java. We will use a simple stack interface in examples and figures. The Java definition of our example interface is illustrated in Figure 2.

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

Figure 2 – Stack interface declaration

Protocols are implemented as a language construct in Java, the protocol declaration. Protocol declarations can be defined as a component of a class or interface. Using our stack example, we need to write a protocol that can accept valid call sequences, like (push, pop) or (push, push, pop) and can identify invalid call sequences, like (pop, push) or (push, pop, pop). This cannot be accomplished with a regular expression or deterministic finite automata (DFA). They cannot count the number of items on a stack. As suggested by Nierstrasz [19], an LTS is used whose alphabet is the public method signatures of the class or interface being described by the protocol. This structure in general is non-deterministic, which allows able us to check the conformance with other protocols during compilation [12].

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

Figure 3 – Stack protocol declaration

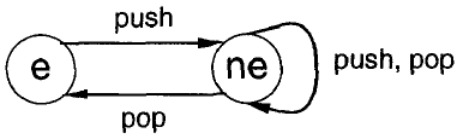

Figure 4 – LTS for Stack interface

Figures 3 and 4 illustrate the protocol declaration for our stack example and the corresponding LTS, respectively. These figures illustrate that non-determinism can arise with the LTS. Consider the call sequence (push, push, pop, pop). While the sequence is a valid sequence, the LTS cannot know for sure whether a call to pop will leave it in the empty state (e) or the non-empty state (ne) [12]. Section 2.4 will illustrate how this problem is resolved.

2.1 Parsing

Protocol support will require modifications to the javac compiler. The key modification is the introduction of the protocol construct in class and interface declarations. When a protocol declaration is found, the compiler is generates a parse tree, attaches it to the corresponding class or interface object, and reports any syntax errors that are found in the declaration. The protocol declaration grammar is comprised of three types of statements – state declarations, regular expression declarations, and sequencing statements.

State declarations declare one or more states that can be used later in sequencing statements. A state represents the current overall state of the object being represented. There is a default start and final state that can reach and be reached by, respectively, any state. A protocol can have only one start state declared, overriding the default start state, via the “start” modifier. Protocols can have multiple final states; they are identified by use of the “final” modifier. Each state can also contain a predicate, which is a Boolean expression used check the validity of the state. In

Section 2.4 we show how state predicates are utilized to validate state conditions at run-time that cannot be checked statically due to non-determinism.

Regular expression declarations are aliases for regular expressions. This shorthand notation is useful when writing more complex protocols.

Sequencing statements represent the transitions in the protocol's LTS. Statements are composed of an optional state list of starting states, a regular expression of method calls that can be made, and an optional state list of ending states. If omitted then the default state of the DFA is used as the start or ending state, respectively. A single state value is found in the sequence statements starting or ending state list, then that state is set as the start or ending state of the DFA, respectively. If the starting or ending state list has multiple values, then the DFA can be seen as a Cartesian product of the starting state list and ending state list. [12]

2.1.1 Wildcard State

The wildcard state, "*", represents the list of all states declared in the protocol. The default state is not included in the wildcard state. [12]

2.1.2 Regular Expressions

Protocols employ conventional regular expressions for expressing method call sequences. The pattern described by a regular expression, using the set of all method calls defined for the object as the vocabulary, can clearly describe a sequence of method calls. In our regular expressions, the comma operator corresponds to concatenation. [12]

2.1.3 Grammar Modifications

The grammar originally proposed by Butkevich et al. [12] provides all the groundwork needed to support protocols in Java. It only requires a few small modifications to work in javac. Since javac is a LL(1) parser, the RegExp portion of the grammar must be rewritten so that the left-

recursion is removed. This is accomplished by breaking down the clauses into groups of terms and factors and utilizing right-recursion. Operator precedence is then put into place to ensure that the proper associativity is maintained.

Additionally, the RegExpDec clause contains an ambiguity, since the component { "," <JavaId> "=" <RegExp> } ";" conflicts with the <RegExp> "," <RegExp> component of the RegExp clause. This ambiguity in regular expressions is corrected by removing the { "," <JavaId> "=" <RegExp> } construct from the RegExpDec clause. In doing so the ability for defining multiple regular expressions in one statement is taken away from the user, but this is only a trivial loss.

Figure 5 illustrates the adjusted grammar.

2.2 Automata Generation

During semantic analysis, protocols are translated into their corresponding LTS. Translation takes 3 steps. First, each regular expression in the protocol is translated into a non-deterministic finite automata (NFA) using Thompson's construction. Next, each NFA generated in the previous step is translated into a DFA using the subset construction. Finally, each DFA generated by the subset construction is connected corresponding to the transitions described in the sequencing statements of the protocol. After generation, the LTS is stored in the class or interface object for later reference. [12]

```
<ProtocolDeclaration>        ::= "protocol" "{" <ProtocolStatementList> "}"
<ProtocolStatementList>      ::=
                                 | <ProtocolStatement> <ProtocolStatementList>
<ProtocolStatement>          ::= <StateDec> | <RegExpDec> | <SeqStatement>
<StateDec>                   ::= [ "start" ] [ "final" ] "state" <StateDecList>
<StateDecList>               ::= <JavaId> <StateDecListAssign> <StateDecListTail> ";"
<StateDecListAssign>         ::=
                                 | "=" <JavaBoolExp>
<StateDecListTail>           ::=
                                 | "," <StateDecList>

<SeqStatement>               ::= [ "<" <StateList> ">" ] <RegExpDec> [ "<" <StateList> ">" ] ";"
<StateList>                  ::= "*"
                                 | <JavaId> <JavaIdList>
<JavaIdList>                 ::=
                                 | "," <JavaId> <JavaIdList>
<RegExpDec>                  ::= <RegExpTerm> <RegExpTail>
<RegExpTail>                 ::=
                                 | "|" <RegExpDec>
<RegExpTerm>                 ::= <RegExpFactor> <RegExpTermTail>
<RegExpTermTail>             ::=
                                 | "," <RegExpTerm>
<RegExpFactor>               ::= <MethodCallPattern>
                                 | [ "~" ] "[" <MethodCallPatternList> "]"
                                 | "(" <RegExpDec> ")" <RegExpFactorTail>
<RegExpFactorTail>           ::=
                                 | "*" <RegExpDec>
                                 | "+" <RegExpDec>
                                 | "?" <RegExpDec>
<MethodCallPattern>          ::= <JavaId> "(" <PatternArgumentList> ")"
<MethodCallPatternList>      ::= <MethodCallPattern> "," <MethodCallPatternList>
<PatternArgumentList>        ::= <PatternArgument> <PatternArgumentListTail>
<PatternArgument>            ::= "*"
                                 | <JavaId>
<PatternArgumentListTail>    ::=
                                 | "," <PatternArgumentList>
```

Figure 5 – Modified grammar for protocols

2.3 Conformance Check

The compiler performs a conformance check when a class or interface extends or inherits another class or interface and both objects have protocols associated with them. The LTSs of the objects are compared using a slightly modified version of Nierstrasz's algorithm [19] during the type-checking phase of compilation. Nierstrasz's algorithm assumes that all states in the LTS are final states. A designation of non-final states allows us to detect a protocol violation when an object is garbage-collected in a non-final state. To support this, the algorithm is modified to

support non-final states. If the subtype's protocol does not conform to the supertype, then a type error is reported and compilation will fail. Otherwise, the protocol information is stored in the class file as an attribute of the object [12].

2.4 State Predicates

State predicates are Boolean expressions that can be used to validate the state of an object. Figure 6 illustrates a protocol declaration for our stack example. These expressions allow designers and developers to provide richer descriptions of states. They also allow for the resolution of the internal non-determinism in the LTS. Consider the call sequence (push, pop, pop). The sequence is invalid, but without state predicates the erroneous condition cannot be identified. State predicates allow for the protocol to determine which state is appropriate for the object. Using the predicates in Figure 6, the protocol can identify that after the first pop, the object is in the empty state. When the next call to pop is made, the protocol will raise the error condition and be handled as prescribed [12].

```
protocol {
        start final state e = isEmpty();
        final state ne = !isEmpty();
        <*> push <ne>
        <ne> pop <*>
}
```

Figure 6 – Stack protocol declaration with state predicates

2.5 Bytecode Generation

After a successful conformance check, the protocol must be stored in bytecode. To do so, the parse tree of the code being compiled must be altered. In order to do this, the wrapper class is generated as described next in Sections 2.5.1. Once the necessary wrapper classes have been generated, the parse tree can be altered. The abstract syntax tree (AST) object representing the

class being instantiated is replaced with the appropriate tree node for the wrapper class. This ensures that during bytecode generation the wrapper class is executed by the JVM enforcing the protocol upon the interface.

2.5.1 Wrapper Classes

Wrapper classes are generated by the compiler when an interface with a protocol is defined. The class is generated dynamically by the compiler to accomplish in order to:

- Generate the run-time representation of the LTS (as an instance of ProtocolInformation)
- Surround the wrapped object's methods with tracing statements.
- Implement the verify method for the protocol

Each method in the wrapped class is implemented as a sequence of three calls in the wrapper class:

1. A call to announce() to check if the request is valid (i.e., the transition can be made)
2. A call to the corresponding method of the server interface
3. A call to advance() to transition the LTS and verify state predicates.

The verify() method removes any states whose predicates are not satisfied from the current state set. This method must be defined in the wrapper since no knowledge of the protocol itself is known to the underlying tracing system. [12]

2.6 Run-time Tracing

Run-time tools are needed to trace the sequence of calls on objects and ensure the conformance to their protocol when applicable. In order to accomplish this, wrapper classes are utilized to invoke an LTS corresponding to the object's protocols and manage conformance checks. Figure 7 shows the UML diagram of the tracing system for our stack example. Butkevitch et al.

illustrate how this modeling can be successful for interfaces, but limit their research to the study of tracing interfaces. We will show in Section 3.1 how this framework can be adapted for supporting class tracing.

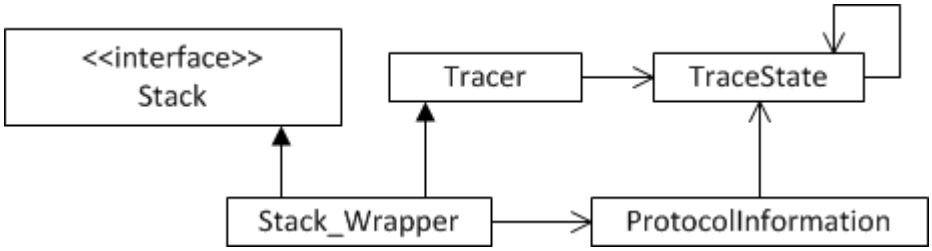

Figure 7 – UML diagram of tracing system for Stack

2.6.1 Tracer Class

The Tracer class is invoked in three methods – announce, advance, and finalize – throughout the lifecycle of the object to manage the LTS. The announce method is used to verify that an upcoming transition is appropriate. To do so it must iterate over the current state list and see if any of the states have an outgoing transition that allows that method call. The advance method is called after the method has been invoked on the wrapped object. It updates the list of current states to be that of the new set of states which the server may be in. An empty current state list indicates that an error has occurred. The finalize method leverages Java's garbage collection system to perform the check to ensure that the protocol ends in a final state. If the object is being in a non-final state when the garbage collector calls the wrapper objects finalize method, then the error handler is invoked [12].

2.6.2 TraceState Class

TraceState objects contain the information necessary to capture the state of a protocol. As the LTS is updated, the TraceState object must be as well. TraceState objects are kept in a static list in memory throughout the execution of the program. This allows the querying of state information by the user [12].

2.6.3 TraceFilter Interface

The TraceFilter interface can be used to query the set of TraceStates stored based on specified conditions. An enumerable object is returned to the user for parsing/processing [12].

2.6.4 ProtocolInformation Class

The ProtocolInformation object is the representation of the LTS in the tracing system, and are generated during compilation and tied to the wrapper classes generated during run-time. ProtocolInformation objects store and operate on the following data

- Names of methods
- Names of states (declared and internal)
- Name of interface
- Starting state
- Final states
- Transition matrix

The transition matrix allows for fast querying of the LTS to find the set of states that can be transitioned to from. This is done by structuring the matrix in a 2D array where each element in the array is a set of states that can be transitioned to. The rows of the array represent the state to be transitioned from and the columns represent the method name on the outgoing transition [12].

2.6.5 Error Handling

The error handling system provided allows users to create their own classes for handling the processing of errors. They can be silently ignored, logged, handled with exceptions, or in any way the user desires. This is done by the ErrorHandler interface, which follows the Strategy design pattern [17]. There are 3 kinds of errors that can be detected by the tracing system:

- Invalid method errors – these errors occur when a method is requested that has no outgoing transition in the LTS and are discovered in the announce method.
- Invalid state – these errors occur after a method has been executed on the server and none of the resulting states can be verified; this occurs in the advance method.
- Non-final state – these errors occur when a wrapper object has been garbage collected and is not in a final state.

Whenever an erroneous condition is detected, the corresponding method on the object instantiating the ErrorHandler interface is invoked [12].

3 Framework Modifications

As described in Chapter 2, Butkevich et al. [12] have provided a framework for the tracing of object protocols when these protocols are applied to interfaces in the Java language. This chapter outlines several modifications/enhancements that can be made to the framework to provide additional features to the framework or to enhance those features already provided by the framework.

3.1 Support for Classes

While there are variances between the structures of classes, abstract class, and interfaces [3, 4, 7], there are advantages that can be gained from the enforcement of protocols on class objects. Let us consider abstract classes. They may or may not include abstract methods and they cannot be instantiated directly, thus they are always subclassed. This proves to be a prime opportunity to use protocols as a debugging tool. Abstract classes rely on the programmer to implement those pieces of code in which behaviors are application specific. As with interfaces, when using abstract class and/or traditional class hierarchies it is not possible to guarantee that the proper sequence of method calls is followed. Thus, protocols can be leveraged to do so, acting on behalf of the developer/API designer.

3.1.1 Type Casting

Figure 8 shows the basic form of the assignment of an object of type C to an interface reference a, of type I. We will assume that both I and C have protocols and the protocol of C conforms to the protocol of I. Section 2.5 defines how the wrapper class will be inserted into the parse tree and executed by the JVM.

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

Figure 8 – Simple assignment of a class to an interface

What are not clearly defined are the mechanisms for handling casting of objects that have been replaced by protocol wrappers. Such a situation is illustrated by the code listed in Figure 9.

```
public interface I {
   protocol { ... }
   ...
}
...
public interface IPlus extends I {
   protocol { ... }
   ...
}
...
public class C implements IPlus {
   protocol { ... }
   ...
}
...
IPlus a = new C();
...
I b = (I)a;
...
IPlus d;
if (a instanceof IPlus)
  d = (IPlus)a;
else
  d = new C();
```

Figure 9 – Casting objects bound by protocols

The outcome of the casts from objects of C to IPlus or from IPlus to I, with respect to protocols, are not defined – what should be done to their protocols? In order for the run-time environment to understand and properly handle the issues, some modifications will need to be made to the architecture of the compiler [10, 16]. Type casts are base-checked during compilation for those that cannot possibly succeed (like comparing a double to a boolean). However, the actual subtype checks for reference types occur at run-time [21].

When casting an object with a protocol to an object of a superclass, the resolution is straight-forward. We will call the object receiving the cast on the left-hand side of the expression LHS, and similarly we will call the object being cast on the right-hand side of the expression RHS. If the type of the LHS also has a protocol, then the current state set of the LHS is set to the subset of states from the RHS that are also found in the LHS protocol. Afterwards, the wrapper around the RHS should be discarded as the state of the protocol can no longer be traced. Downward casts to subclasses can be handled in the same manner. If the LHS has no protocol, then the wrapper around the RHS is discarded as mentioned above.

3.2 On-demand Tracing

In DeLine and Fähndrich's paper on protocols in low level software [14] they introduce the notion of tracked types and objects. We adapt this notion to our implementation of protocols in Java. As mentioned above when discussing supporting classes with protocols, a large number of objects being traced could potentially degrade system performance. In large systems where the number of objects being traced could be in the hundreds or thousands, the increased execution time could introduce significant lag in the overall performance of the system. By providing the functionality for programmers to choose which objects will be traced, the overhead costs of protocols can be kept under control. We introduce the keyword traced to the grammar of class creation statements and array creation statements [5] using the Tracedopt construct, whose value is either "traced" or null. Figure 10 outlines the new grammar for class and array creation statements. While some variable declarations may contain creation expressions, Figure 11 demonstrates several class and array declarations which do not instantiate objects [5]. Since no objects are instantiated in these cases, protocols are not utilized unless a traced objects is assigned to one of the variables.

ClassInstanceCreationExpression:

new Tracedopt TypeArgumentsopt TypeDeclSpecifier TypeArgumentsOrDiamondopt
(ArgumentListopt) ClassBodyopt

Primary . new Tracedopt TypeArgumentsopt Identifier TypeArgumentsOrDiamondopt
(ArgumentListopt) ClassBodyopt

ArrayCreationExpression:

new PrimitiveType DimExprs Dimsopt

new Tracedopt ClassOrInterfaceType DimExprs Dimsopt

new PrimitiveType Dims ArrayInitializer

new Tracedopt ClassOrInterfaceType Dims ArrayInitializer

Tracedopt:

traced

Figure 10 – Modified creation expression for classes

```
int[]     ai;              // array of int
short[][] as;              // array of array of short
Object[]  ao;              // array of Object
Collection<?>[] ca;        // array of Collection of unknown type
```

Figure 11 – Array and class declarations that do not instantiate objects

The traced option will be identified during parsing and leveraged during semantic analysis and code generation to generate and run the protocols at run time. In order to prevent misuse of the statement, a new check will have to be implemented in the type-checking. This check will identify when a traced statement is in use and ensure that it is being used on a class object.

Primitive data types and enumerations are not supported by protocols and use of a declaration such as “new traced int[11];” are considered erroneous expressions.

Allowing for the explicit declaration of objects to be traced forces some changes on the original framework. Section 3.1 demonstrated how protocols are enforced upon assignment of an object to an interface when the object has a protocol conforming to the interface’s protocol. The conformance checking phase of compilation described in Section 2.3 will need a minor modification to support this functionality. The conformance analysis algorithm will need to be adjusted so that instead of searching for classes or interfaces that extend or inherit another class or interface, it simply checks the flag set by the scanner as to whether or not the traced option is enabled. If the option is enabled, then the conformance analysis will be performed and upon successful completion the wrapper classes will be added as they normally would.

3.3 Timing Constraints

We have shown how object protocols provide the capability for exercising sequencing constraints upon objects. The framework described provides a utility that can be utilized for more than originally intended. It provides developers with the ability to define more comprehensive descriptions of object behavior within the language itself, and to leverage the compiler and run-time environment to enforce those behaviors. In this section, we consider another property of software that can be used for debugging other types of problems in software engineering – execution time. The execution time of software plays a critical role in a variety of software systems, such as real-time computing and embedded systems. The ability to ensure that an action occurs within the correct time frame and detecting when it does not is critical in such systems. Protocols can be enhanced to provide developers with the ability to define such timing constraints and detect when the constraints are not satisfied. Using the error handling

mechanisms provided by the framework, the error can be dispatched and handled as prescribed by the implementer.

Deng [15] provides one example of a grammar for timing constraints in automata. His implementation uses only one clock, which represents the length of time the automaton has been in its current state. Supporting such an environment in the tracing framework would require some modifications, namely the introduction of a clocking mechanism to support multiple named clocks. Using such a clock mechanism, the generated wrapper objects would call the clocking mechanisms as necessary to enforce the defined timing constraints. The existing error handling mechanisms in the framework can be utilized to support reporting the errors that can occur when using a timed automaton.

3.4 Optimizations

By allowing a flexible representation of the automata, several optimizations can be leveraged. The manner in which a protocol is defined can allow the compiler to determine what type of automata will be used to represent the protocol's transition system. Protocols that do not utilize state predicates can be optimized to a DFA implementation rather than an NFA. The removal of the run-time checks allows the transformation to the DFA since all choices are known in advance. This increases the speed of the algorithm since the run-time checks can be discarded and a transition matrix can be used for lookups in O(1) time.

Optimizations are found for the NFA representation of the automata as well. Some protocols may not introduce epsilon transitions. During the announce() phase of a transition, an epsilon closure is performed to add any state that can be reached via an epsilon transition to the list of

possible states the automata can be in after the transition. If no epsilon transitions are found in the automata, the compiler will utilize an NFA structure that does not perform an epsilon closure.

Augmenting the underlying storage mechanism of the automata yielded excellent results. The tracing framework was originally implemented using 3-dimensional arrays to store the representation of available states on a transition from another state. The Java BitSet library [10] is used instead. The BitSet library, while implemented uniquely to each target platform, is specially crafted to make efficient use of storage and indexing to represent large binary strings as well as providing efficient manipulation of the data – even when the number of binary digits in the string is unknown and greater than 32. The BitSet storage implementation aides tremendously in the reduction of the tracing framework's execution times. One such example is calculating an epsilon closure or the set of states that can be reached on an input. The array implementation required several nested loops to calculate this information. Using BitSets, the operation can be performed as a union operation on BitSets representing the transitions on a state. This yields one loop over the number of states, and a union operation is performed during each iteration of the loop. The union operation on the BitSet can be performed as a logical or – which is a much faster operation than the array operations.

In Section 4.2 we will see the timing measurements and the improvements made by the optimizations described. The optimized code can be found in Appendix C.

4. Implementation

4.1 OpenJDK

An implementation of protocols in Java was begun in the OpenJDK 7 b147 compiler [8]. The grammar modified to support javac in Figure 5 was used in the implementation. Currently, the implementation of the parser has been completed. This work includes:

- Command-line options for protocol handling
- The building of parse trees for object protocols which requires the additional methods for recursive descent and tree node definitions.
- The implementation of the corresponding tree traversal mechanisms
    - Pretty printing
    - TreeScanner
    - TreeCopier
    - TreeMaker
    - TreeVisitor

The work for building the corresponding automata has begun and is nearing completion. Once that has been completed the conformance checks and wrapper generation will begin.

4.2 Efficiency Analysis

4.2.1 Experimental Setup

The experiments were performed on a system with the following hardware and software configurations:

- Intel(R) Core(TM)2 CPU 6320 @ 1.86GHz
- 2 GB RAM

- Debian GNU/Linux v. 6.0.3
    - Base install, plus SSH server
- Java environment is jdk1.7.0_02

During each test run, the method call being tested is executed several times prior to being timed. This ensures that the code has been loaded in the cache and provides more accurate results since the time spent on loading the cache is not reflected in the later calls.

4.2.2 Experiment Results

Utilizing the prototype code for the tracing and protocol system provided by the The Brew Project [1], performance tests were performed. In Section 2.6, Figure 7 illustrated the architecture of the tracing system. The Stack_Wrapper subclasses the abstract class Tracer and extends the same interface that the wrapped class extends. The wrapper is designed so that the constructor takes the client and server objects as references. This design was leveraged to substitute various implementations of a Stack for testing purposes. In addition to the base framework, the optimizations described in Section 3.4 are tested as well.

The three types of stack implementations used in testing are:

1. Stacks with empty method stubs – these method calls simply return right away.
2. Stacks with an array storage scheme – stack elements are stored in an array and manipulated based on an index number.
3. Stacks with the java.util.Stack library – the stack is implemented using the stack implementation from the Java core.

Each of these implementations is listed in Appendix B. Table 1 shows the results of the base tests. These tests time the method calls on each implementation of the stack interface without protocols and the tracing framework.

Table 1 – Control test results (in nanoseconds)

Implementation	Iterations	Method	Min Time	Average Time	Max Time	Standard Deviation	Error
Empty stubs	500	push	823.00	875.71	947.00	19.10	1.85
		pop	835.00	885.61	944.00	20.43	1.66
	1000	push	815.00	875.79	951.00	20.37	1.24
		pop	827.00	883.17	966.00	21.47	1.26
Array base	500	push	1064.00	1128.80	1203.00	26.57	2.50
		pop	1030.00	1091.82	1158.00	23.66	2.24
	1000	push	1045.00	1128.04	1237.00	28.50	1.72
		pop	1015.00	1093.29	1192.00	25.36	1.54
java.util.Stack	500	push	2019.00	2129.90	2294.00	42.00	3.65
		pop	2853.00	2958.32	3117.00	43.04	6.58
	1000	push	2011.00	2132.37	2289.00	41.27	2.41
		pop	2812.00	2957.49	3120.00	41.43	2.62

Next, the implementations were tested again using the framework without any optimizations. This time protocols were enabled and the operations executed conformed to the desired sequence of operations. The results of these tests are shown in Table 2. In Table 3 we show the results of the protocol tests when run on a protocol that requires an NFA with epsilon closures. This configuration is the least efficient of all of the automata configurations available.

Table 2 – Unmodified framework test results (in nanoseconds)

Implementation	Iterations	Method	Min Time	Average Time	Max Time	Standard Deviation	Error
Empty stubs	500	push	4105.00	4315.89	4594.00	81.01	7.14
		pop	4267.00	4489.30	4729.00	94.89	9.06
	1000	push	4105.00	4322.85	4598.00	77.07	4.79
		pop	4263.00	4481.33	4736.00	90.53	5.98
Array base	500	push	4511.00	4702.36	4944.00	72.33	7.63
		pop	4586.00	4783.68	5071.00	80.54	6.67
	1000	push	4500.00	4695.43	4902.00	71.42	5.47
		pop	4564.00	4788.93	5139.00	82.24	5.82
java.util.Stack	500	push	5542.00	5774.43	6143.00	90.62	7.05
		pop	6545.00	6851.10	7260.00	96.59	9.13
	1000	push	5497.00	5769.87	6091.00	79.89	4.81
		pop	6545.00	6854.02	7267.00	101.55	7.08

Table 3 – Framework results using an NFA with epsilon closures

Implementation	Iterations	Method	Min Time	Average Time	Max Time	Standard Deviation	Error
Empty stubs	500	push	1207.00	1289.26	1436.00	41.98	3.70
		pop	1181.00	1280.88	1394.00	33.45	2.96
	1000	push	1196.00	1289.51	1440.00	38.27	2.38
		pop	1199.00	1282.16	1406.00	33.54	2.09
Array base	500	push	1410.00	1512.68	1688.00	44.56	3.94
		pop	1354.00	1442.08	1564.00	36.04	3.18
	1000	push	1406.00	1512.16	1677.00	41.28	2.57
		pop	1364.00	1442.09	1575.00	34.48	2.14
java.util.Stack	500	push	2000.00	2159.25	2422.00	64.81	5.74
		pop	2707.00	2898.61	3087.00	62.45	5.52
	1000	push	2004.00	2166.96	2425.00	63.89	3.97
		pop	2639.00	2898.94	3161.00	63.96	3.98

Next, we take a look at the results of the tracing framework using a protocol where the NFA has been implemented without epsilon closures, and finally the protocol optimized to a DFA. The testing results for these configurations are show in Tables 4 and 5 respectively.

Table 4 – Framework results using an NFA without epsilon closures

Implementation	Iterations	Method	Min Time	Average Time	Max Time	Standard Deviation	Error
Empty stubs	500	push	1195.00	1283.43	1444.00	42.59	3.76
		pop	1176.00	1265.57	1399.00	37.09	3.27
	1000	push	1196.00	1284.04	1444.00	41.46	2.58
		pop	1176.00	1267.00	1421.00	36.38	2.26
Array base	500	push	1414.00	1510.18	1665.00	43.09	3.81
		pop	1357.00	1434.47	1568.00	37.05	3.27
	1000	push	1417.00	1511.31	1658.00	44.50	2.76
		pop	1342.00	1437.21	1591.00	38.32	2.38
java.util.Stack	500	push	1981.00	2158.49	2380.00	59.33	5.23
		pop	2665.00	2895.69	3090.00	61.27	5.41
	1000	push	1970.00	2160.76	2391.00	63.34	3.94
		pop	2665.00	2899.60	3154.00	64.13	3.98

Looking at the test results we can see that the operational overhead of our optimized tracing framework is approximately 400 nanoseconds. Based on the timing results shown in Table 1 this is approximately one-half of the time required to execute an empty method. Compared to the original work by Renedo which took approximately 3500 nanoseconds, this is a decrease in

execution time by approximately 88%. With such a small operational impact, the results of using protocols would be minimal in a live operating environment.

Table 5 – Framework results using a DFA

Implementation	Iterations	Method	Min Time	Average Time	Max Time	Standard Deviation	Error
Empty stubs	500	push	1203.00	1286.44	1474.00	42.66	3.77
		pop	1185.00	1274.09	1353.00	32.37	2.85
	1000	push	1188.00	1283.45	1455.00	42.70	2.66
		pop	1177.00	1273.44	1399.00	33.83	2.10
Array base	500	push	1365.00	1510.49	1722.00	51.10	4.52
		pop	1353.00	1451.39	1564.00	37.03	3.27
	1000	push	1387.00	1503.94	1696.00	47.78	2.97
		pop	1364.00	1450.36	1587.00	37.12	2.30
java.util.Stack	500	push	2038.00	2173.21	2395.00	58.33	5.15
		pop	2654.00	2823.71	3068.00	60.15	5.33
	1000	push	1996.00	2172.45	2410.00	56.65	3.71
		pop	2654.00	2822.62	3083.00	63.29	3.93

Another key observation is that the complexity of the underlying objects will have an impact on the performance of the tracing system. This can be seen by looking at the difference in execution times from the array and Java stack implementation. The array implementation is strictly memory based and has short, efficient implementations of the interface methods. The Java stack implementation has a bit more overhead. The java.util.Stack class has several ancestors in the class hierarchy, implements 6 interfaces and supports generics as well. Such overhead certainly slows down the overall run-time of the method; however the impact on protocol performance originates from the state predicates. In the worst case, each predicate is executed at each transition in the protocol to verify the states. Combined with more complex protocols, predicates could have a more serious impact on performance.

Overall, the impact of protocols on the execution time of a method call is on the order of approximately three quarters of a microsecond. We plan to do more extensive testing on more

complex protocols to see how the complexity of the protocol affects the performance of the tracing system.

5. Conclusion

In this thesis, we studied a framework for enforcing the sequence of method calls on classes that implement interfaces in Java programs. This framework enables software developers to conceptualize the desired behavior of an object and define it within the definition of the class or interface. We show how the Java compiler (javac) and run-time environment (JVM) can be extended to analyze and enforce these behaviors. We present an extension to this framework that would allow for class objects to be traced as well, and consider the subtyping needs required by such an addition to the framework. To support these constructs, we provide details on how they can be implemented using the Tracer interface of the supplied framework. These methodologies strive to evolve the capabilities provided by traditional compilation environments by providing verification methodologies for object behaviors that reach beyond those available in a traditional type system.

To evaluate the efficiency of the environment, we used an implementation of the tracing framework to analyze the overhead introduced by the use of protocols. This data shows that the management of automata used in transitioning object states can be done in time on the order of nanoseconds to microseconds, with the majority of time being spent on the actual handling of protocol violations, not within the LTS itself.

Our future work includes the completion of the integration of the tracing framework into OpenJDK, the introduction and implementation of timing constraints into the tracing framework, continued performance testing on more complex protocols and data structures, as well as the methodologies needed to perform more analysis during compilation to reduce the impact on execution times.

References

[1] The Brew Project, http://www.csc.lsu.edu/~gb/Brew/index.html.

[2] Fundamentals of Java Security, http://java.sun.com/developer/onlineTraining/Security/Fundamentals/Security.html.

[3] Interfaces and Inheritance, http://docs.oracle.com/javase/tutorial/java/IandI/index.html.

[4] Java, http://www.oracle.com/technetwork/java/index.html.

[5] The Java™ Language Specification, http://docs.oracle.com/javase/specs/jls/se7/html/.

[6] Java Timeline, http://oracle.com.edgesuite.net/timeline/java/.

[7] OpenJDK, http://openjdk.java.net/.

[8] OpenJDK 7 b147, http://www.java.net/download/openjdk/jdk7/promoted/b147/openjdk-7-fcs-src-b147-27_jun_2011.zip.

[9] RandomAccessFile (Java Platform SE 7), http://docs.oracle.com/javase/7/docs/api/java/io/RandomAccessFile.html.

[10] Class BitSet (Java Platform SE 7) http://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html

[11] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple Dispatch and Retroactive Abstraction for Java™. Technical Report No. OSU-CISRC-5/01-TR08, Dept. of Computer and Information Science, The Ohio State University, March 2002.

[12] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 50–59, New York, NY, USA, 2000. ACM.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge Massachusetts, 2001.

[14] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pages 59–69, New York, NY, USA, 2001. ACM.

[15] Guohui Deng. Generating Embedded System Analyzer. Master's Project. Louisiana State University, A & M, 2007.

[16] Madhu Gopinathan and Sriram K. Rajamani. Enforcing object protocols by combining static and runtime analysis. *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, pages 245–260, New York, NY, USA, 2008. ACM.

[17] Erich Gamma, Richard Helm, Ralph E. Johnson, and John. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[18] Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An Analysis of Constrained Polymorphism for Generic Programming. *Multiparadigm Programming 2003: Proceedings of the MPOOL Workshop at OOPSLA'03*, 87 – 107.

[19] Oscar Nierstrasz. Regular types for active objects. *Proceedings of the OOPSLA '93 Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1-15. Association for Computing Machinery, 1993. ACM SIGPLAN Notices, 28(10), October 1993.

[20] Ganesan Ramalingam, Alex Warshavsky, John Field, Deepak Goyal, and Mooly Sagiv. Deriving specialized program analyses for certifying component-client conformance. *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 83–94, New York, NY, USA, 2002. ACM.

[21] Peter Sestoft. *Java Precisely*. MIT Press, Cambridge, Massachusetts, 2005.

[22] Jan Van Den Bos and Chris Laffra. Procol: a parallel object language with protocols. *OOPSLA '89: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 95–102, New York, NY, USA, 1989. ACM.

Appendix A: Original Protocol Grammar

```
<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>
                              | [ "~" ] "[" [ <MethodCallPattern> { "," <MethodCallPattern> } ] "]"
                              | <RegExp> "*" | <RegExp> "+" | <RegExp> "?"
                              | <RegExp> "|" <RegExp> | <RegExp> "," <RegExp> | "(" <RegExp> ")"

<MethodCallPattern>       ::= <JavaId> [ "(" [ <PatternArgumentList> ] ")" ]

<PatternArgumentList>     ::= <PatternArgument> { "," <PatternArgument> }

<PatternArgument>         ::= "*" | <JavaType>
```

(From Butkevitch et al. [12].)

Appendix B: Testing Implementations of the Stack Interface

B.1 Stack with Stub Methods

```
public class StackImpl implements TestStack{
   public int pop() { return 0; }
   public void push(int x) {}
   public boolean isEmpty() { return true; }
}
```

B.2 Stack with Array Back-end

```
public class StackArrayImpl implements TestStack{
   int sp = 0;
   int [] stack = new int[100];
   public int pop() {
      if (sp <= 0)
         return 0;
      sp--;
      return stack[sp];
   }
   public void push(int x) {
      if (sp < 100) {
         stack[sp] = x;
         sp++;
      }
   }
   public boolean isEmpty() { return (sp == 0); }
}
```

B.3 Stack with java.util.Stack Back-end

```
import java.util.Stack;

public class StackUtilImpl implements TestStack{
   private Stack<Integer> stack = new Stack<Integer>();
   public int pop() {
            int val = 0;
            try {
                  val = stack.pop();
            }
            catch(Exception e) { ; }
            return val;
   }
   public void push(int x) { stack.push(x); }
   public boolean isEmpty() { return stack.empty(); }
}
```

Appendix C: Optimized Tracing Framework Code

C.1 ProtocolInformationDFA.java

This code represents the protocol automaton optimized to a DFA.

```
package edu.ohio_state.cis.lts;

import java.util.BitSet;

public class ProtocolInformationDFA {

    /** The name of the interface in which this protocol is defined. */

    public String interfaceName;


    /** An array contatining all public method names of the interface

     * (used in the protocol?).

     */

    public String[] methodNames;


    /** An array containing the names of the states, both defined by the user

     *  and generated internally.

     */

    public String[] stateNames;


    /** Specifies which states are final. */

    public boolean[] finalStates;


    public int[][] tMatrix;


    /** Returns the name of the interface in which this protocol is defined. */
```

```
public String getInterfaceName() { return interfaceName; }

/** Returns the name of the method with the given index.
 *  @param m the index of the method.
 */
public String getMethodName(int m) { return methodNames[m]; }

/** Returns the name of the state with the given index.
 *  @param s the index of the state.
 */
public String getStateName(int s) {
        return stateNames[s];
}

/** Returns the number of states used in this protocol.
 */
public int nStates() {
        return stateNames.length;
}

/** Returns true if the state is final and false otherwise.
 *  @param s the index of the state.
 */
public boolean isFinalState(int s) { return finalStates[s]; }
```

```
    public ProtocolInformationDFA() {}


    public ProtocolInformationDFA(String interfaceName, String[] methodNames, String[] stateNames, boolean[] finalStates, int[][] tMatrix) {
        this.interfaceName = interfaceName;
        this.methodNames = methodNames;
        this.stateNames = stateNames;
        this.finalStates = finalStates;
        this.tMatrix = tMatrix;
    }
}
```

C.2 ProtocolInformationNFA.java

This code represents the protocol automaton as an NFA. When testing for NFA without epsilon closure, the calls to the epsilon closure function were simply commented out.

```
package edu.ohio_state.cis.lts;

import java.util.BitSet;

public class ProtocolInformationNFA {

    /** The name of the interface in which this protocol is defined. */
    public String interfaceName;


    /** An array contatining all public method names of the interface
     * (used in the protocol?).
     */
    public String[] methodNames;


    /** An array containing the names of the states, both defined by the user
```

```
 *  and generated internally.
 */
public String[] stateNames;

/** Specifies which states are final. */
public boolean[] finalStates;

public BitSet[][] tMatrix;

/** Epsilon transitions from one state to another */
public BitSet[] eTransitions;

/** Returns the name of the interface in which this protocol is defined. */
public String getInterfaceName() { return interfaceName; }

/** Returns the name of the method with the given index.
 *  @param m the index of the method.
 */
public String getMethodName(int m) { return methodNames[m]; }

/** Returns the name of the state with the given index.
 *  @param s the index of the state.
 */
public String getStateName(int s) {
      return stateNames[s];
```

```
    }

    /** Returns the number of states used in this protocol.
     */
    public int nStates() {
        return stateNames.length;
    }

    /** Returns true if the state is final and false otherwise.
     *  @param s the index of the state.
     */
    public boolean isFinalState(int s) { return finalStates[s]; }

    public BitSet reachableStates(int method, int state) { return tMatrix[method][state]; }

    public BitSet epsilonTransitions(int state) { return eTransitions[state]; }

    public ProtocolInformationNFA() {}

    public ProtocolInformationNFA(String interfaceName, String[] methodNames, String[]
stateNames, boolean[] finalStates, BitSet[][] tMatrix, BitSet[] eTransitions) {
        this.interfaceName = interfaceName;
        this.methodNames = methodNames;
        this.stateNames = stateNames;
        this.finalStates = finalStates;
```

```
        this.tMatrix = tMatrix;
        this.eTransitions = eTransitions;
    }
}
```

C.3 TracerDFA.java

This code represents the tracer class adjusted to use the ProtocolInformationDFA class.

```
package edu.ohio_state.cis.lts;

import java.util.BitSet;

public abstract class TracerDFA {

    /** This references points to the head of the TraceState list
        stored statically in the Tracer class. */
    final static TraceState head = new TraceState();

    /** This reference points to an empty TraceState after the last
        element of the TraceState list stored statically in the Tracer
        class. Very helpful to add new elements to the list without
        having to run over the whole list. */
    final static TraceState tail = new TraceState();

    static { head.next = tail; tail.prev = head; }

    /** This is the default error handler object. When a wrapper is
```

```
     *  constructed we assign this error handler to its
     *  TraceState.
     */
    static private ErrorHandler defaultEh = new DebugLevel0(System.out);

    /** This is the TraceState that is updated by this Tracer object. */
    private TraceState ts;

    private int curState;

    /** Sets the error handler that is used when a new
     *  protocol is started.
     */
    static public final void setDefaultErrorHandler(ErrorHandler eh) {
        defaultEh = eh;
    }

    /** This method is used by the user to modify the error handler of
       *  some of the TraceStates in the TraceState list. It has two
       *  arguments: the New ErrorHandler object and a TraceFilter that
       *  selects the TraceStates to be modified.
       */
    static public final void setFilteredErrorHandler (ErrorHandler eh, TraceFilter tf) {
        for(TraceState t = head.next; t != tail; t = t.next) // if t selected reset ErrorHandler to
eh.
```

```
            if (tf.selected(t))
                t.eh = eh;
    }

    /** Constructor of the Tracer. This constructor cannot be called
     *  by the user. It is called only by the Wrapper.
     *  @param pInfo the protocol information to be used in the tracer
     *  @param fileAndLine the string containing the name of the file and the number of the line where the communication link has been established.
     *  @client the reference to the object in which the communication link is established
     *  @server server object whose protocol we will be tracing.
     */
    protected TracerDFA (ProtocolInformationDFA pInfo, String fileAndLine, Object client, Object server) {
        ts = new TraceState(pInfo, defaultEh, fileAndLine, client, server);
        // add it to the list if necessary
        if (ts.eh.insertInTraceList()) {
            tail.prev.next = ts;
            ts.prev = tail.prev;
            ts.next = tail;
            tail.prev = ts;
        }
    }

    /** Called by the wrapper to announce that a method is about to be called.
     * Does checks whether this method is allowed to be called in the current
```

```
     * lts state. Does not advance the lts.
     * @param m the indexe of the method that is about to be called
     * @see advance()
     */
    protected final void announce(int method) {
        if (ts.error != TraceState.NO_ERROR)
            return;
        if (ts.pInfo.tMatrix[curState][method] == -1)
            ts.eh.invalidMethod(ts);
    }

/** Called from the wrapper after a method is called on the server.
    *  Advances the lts.
    */
    protected final void advance(int method) {
        // tracing disconnected if there was an error already
        if (ts.error != TraceState.NO_ERROR)
            return;
        if ((curState = ts.pInfo.tMatrix[curState][method]) == -1)
            ts.eh.invalidState(ts);
    }

    /** Checks the predicate conditions associated to each possible
     *  state. It rules out the ones that aren't satisfied.
     */
```

```
protected abstract void verify();


/** Is called when the Wrapper that extend this Tracer is garbage collocted.
 *  Makes sure that the client and server can be garbage-collected as well by
 *  replacing reference to them in the TraceState object to the references to their
 *  correspondng classes. Checks whether the lst is in a final state. If not --
 *  sends a signal to the error handler.
 */
final public void finalize() {
        /* replace objects references by their classes so they can be
           garbage collected. */
        ts.server = ts.server.getClass();
        ts.client = ts.client.getClass();
        // check if in final state
        if (!ts.pInfo.isFinalState(curState))
                ts.eh.notInFinal(ts);
        return;
}


/** Dumps information about the states of all objects whose lts's are being
 *  traced using the given default error handler and trace filter.
 *  eh the error handler to be used for dumping the info
 *  tf the trave filter that determines which states will be dumped and which will be not.
 */
final static public void dump(ErrorHandler eh, TraceFilter tf) {
```

```
        head.next.recDump(eh, tf);
        return;
    }

    /** Dumps information about the states of all objects whose lts's are being
     *  traced using the default error handler and trace filter.
     */
    final static public void dump() {
        head.next.recDump();
        return;
    }
}
```

Vita

Ronnie Gilkey was born in Delaware in 1981, and moved to Baton Rouge, Louisiana, in 1988. He earned his Bachelor of Science in Computer Science at Louisiana State University in May 2007. In August of 2009 he came back to Louisiana State University to pursue graduate studies in systems science. He is currently a candidate for the degree of Master of Science in Systems Science, which will be awarded in May 2012.