UNIVERSITY OF CALIFORNIA
Santa Barbara

Interface Grammars for Modular Software Verification

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Graham Hughes

Committee in Charge:

Professor T. Bultan, Chair
Professor C. Krintz
Professor J. Su

March 2009

The dissertation of
Graham Hughes is approved:

Professor C. Krintz

Professor J. Su

Professor T. Bultan, Committee Chairperson

December 2008

Interface Grammars for Modular Software Verification

To my parents, Noreen and Richard, for your continuous support.

Curriculum Vitæ

Graham Hughes

Education

1995	Bachelor of Arts in Computer Science and Math, University of California at Santa Barbara

Experience

2001 – 2002	Teaching Assistant, University of California, Santa Barbara.
2002 – 2008	Graduate Research Assistant, University of California, Santa Barbara

Selected Publications

G. Hughes and T. Bultan. “Automated Verification of Access Control Policies Using a SAT Solver,” In *International Journal on Software Tools for Technology Transfer (STTT)*, special issue on selected papers from the Workshop on Web Quality, Verification and Validation (WQVV 2007) vol. 10, no. 6, pp. 473 534, December 2008.

G. Hughes, T. Bultan and M. Alkhalaf. “Client and Server Verification for Web Services Using Interface Grammars,” In *Proceedings of the Workshop on Testing, Analysis and Verification of Web Software (TAV-WEB 2008)*, pp. 40-46, Seattle, Washington, July 21, 2008.

G. Hughes and T. Bultan. “Interface Grammars for Modular Software Model Checking,” To appear in *IEEE Transactions on Software Engineering*, special issue on selected papers from the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007).

G. Hughes and T. Bultan. “Automated verification of XACML policies using a SAT solver,” In *Proceedings of the Workshop on Web Quality, Verification and Validation (WQVV ’07)*, 2007.

G. Hughes and T. Bultan. Extended interface grammars for automated stub generation. In *Proceedings of the Automated Formal Methods Workshop (AFM 2007)*, New York, New York, United States, 2007. ACM.

G. Hughes and T. Bultan. “Interface grammars for modular software model checking,” In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA ’07)*, pages 39–49, 2007.

G. Hughes, S. P. Rajan, T. Sidle, and K. Swenson. “Error detection in concurrent java programs,” In *Proceedings of the Workshop on Software Model Checking (SoftMC 2005)*, volume 144, pages 45–58. Electronic Notes in Theoretical Computer Science, Feb. 2006. Issue 3.

Abstract

Interface Grammars for Modular Software Verification

Graham Hughes

Applying model checking techniques directly to programs has shown extensive promise; however, two related problems hinder applicability of model checking to software on a wider scale. First, the state space explosion problem (i.e., an exponential increase in the search space by increasing number of variables and concurrent components) limits the scalability of model checking techniques and second, the environment generation problem (i.e., finding models for parts of software that are outside the scope of the model checker) limits the applicability of model checking to the domains where such environment models are available. I propose a semi-automated approach to attack the above mentioned problems. Specifically, I propose an interface specification language and require the users to write interface specifications for components of a program that are outside the scope of the current verification effort. My interface specification language allows a user to write an *interface grammar* for a component to specify the constraints on the ordering of calls made by the program to that component. This approach enables modeling of nested call structures that cannot be expressed by interfaces based on finite state machines. I built an interface compiler that takes the interface grammar for a component as input and generates a stub for that component. The stub generated from the interface grammar of a component is used to replace that component during state space exploration, either to assuage the state space explosion, or to provide an executable environment for the component that is being verified.

Professor T. Bultan
Dissertation Committee Chair

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Applying model checking techniques directly to programs [5, 20, 56] has shown promise for specific verification tasks, such as checking for concurrency errors [56] or checking device drivers for interface violations [5]. Indeed, I have demonstrated success with these techniques in previous work [29]. However, there are two related problems that hinder applicability of model checking to software in a wider scale: first, the state space explosion problem (i.e., an exponential increase in the search space by increasing number of variables and concurrent components) limits the scalability of model checking techniques and second, the environment generation problem (i.e., finding models for parts of software that are outside the scope of the model checker) limits the applicability of model checking to the domains where such environment models are available.

I have focused on model checking Java programs. I have specifically focused on the Java Path Finder (JPF) [56] model checker, which can model check Java programs directly. However, performing this task is limited by the two problems cited above. JPF cannot handle native calls in Java programs, and it has difficulty model checking large programs without some form of isolating parts of the program from each other. Hence, in order to use JPF for verification of Java programs, one has to write extensive and sometimes intrusive environment models, which is a daunting task.

Inability to handle native code is not only a limitation specific to JPF, but also the sign of an inherent problem in model checking. In order to search the state space of a program exhaustively (as most model checkers attempt to do), one needs a representation of that state space. JPF chooses to model the state space of a Java program by recording configurations of the Java Virtual Machine (JVM). JPF has its own JVM which keeps track of different configurations that are visited during the execution of the program that is being verified. However, execution of native code, by definition, moves the program execution outside the scope of the JVM and hence cannot be observed by JPF. Even if one tries to keep track of program execution at a lower level of abstraction, perhaps by keeping track of the physical memory and processor state, a similar problem will arise if one tries to analyze a distributed program which involves interactions among multiple machines or a program that interacts with a database server, etc. Eventually, this will require keeping track of the state of each and every component that the program interacts with. This is unlikely to be a scalable approach due to the state space explosion. Moreover, in many (if not the majority) of cases, the developer who is trying to check the correctness of a program may not have access to the code of all the components that the program interacts with.

I propose a semi-automated approach to attack the above mentioned problems. Specifically, I propose an interface specification language and require the users to write interface specifications for components of a program that are outside the scope of the current verification effort. My interface specification language allows a user to write an *interface grammar* for a component to specify the constraints on the ordering of calls made by the program to that component. This approach enables modeling of nested call structures that cannot be expressed by interfaces based on finite state machines. Moreover, in order to provide a flexible approach that can handle complex interface constraints, my interface specification language allows the users to escape to Java and write semantic predicates or actions in Java or in the Java Modeling Language (JML), specifying the behavior of the component (similar to the approach used by parser generators such as Yacc [36]). I believe that my approach provides a balance between two extreme alternatives, i.e., writing stubs completely manually or automatically extracting simple abstract models such as finite state machines.

In addition to specifying this language, I present a compiler that translates interface grammars to the Java programming language. Further, I extend the compiler to allow expressing portions of the interface grammar using JML expressions, and apply this tool to the task of verifying two large systems.

In chapter 2 on the following page, I present my analysis of a large software system using manual environment generation, to motivate the development of interface grammars. In chapter 3 on page 25, I present a high level overview of my approach. In chapter 4 on page 29, I define interface grammars. In chapter 5 on page 48, I describe the semantics of interface grammars. In chapter 6 on page 79, I describe technical details required to translate the algorithms from the previous section into the Java language. In chapter 7 on page 87, I describe an extension to interface grammars to make specifying bidirectional interface grammars possible. In chapter 8 on page 126, I present a case study applying interface grammars to the task of verifying clients using the Enterprise Java Beans specification. In chapter 9 on page 141, I present another case study applying interface grammars to the task of verifying clients and servers implementing a web service. Finally, in chapter 10 on page 169 I present related work and in chapter 11 on page 175 I present directions for future research in interface grammars.

Chapter 2

Model Checking with Manual Environment Generation

First, I will discuss why one might desire to apply model checking techniques to programs. When model checking was first developed they were applied to extracted models and protocols, not real code. Applying model checking techniques to real code is nontrivial.

As it happens, analyzing concurrent programs is very useful, due to the difficulty of repeatably exhibiting most sorts of concurrent errors. The most common technique for detecting errors in large commercial systems is testing, of course; however, creating good test cases can be very difficult independent of any concurrency concerns, and testing for concurrency errors can be extremely frustrating. I present here a case study in which a specialized model checker was used to discover concurrency errors in a large preexisting code base. Although this depicts a largely manual effort, I also discuss ways to reduce the labor required. I discuss the general problem of applying a model checker to a code base in Section 2.1, the code base I examined in Section 2.2, the results I achieved by applying model checking to that code base in Section 2.3, and reflections upon the methodology I used in Section 2.5.

Model checking large programs is uncommon but not entirely unheard of. Yang, Twohey et al. [66], applied the CMC model checker [41] to the Linux kernel, specifically on three filesystems, and found numerous errors in the filesystems that could potentially cause data corruption or loss. Ball, Majumdar et al. [4], presented a method for performing predicate abstraction on C programs as a part of the SLAM toolkit, with the intent of model checking the resulting Boolean programs; they mention using this technique on NT device drivers, among other programs. The Java PathFinder team was asked to find an error in NASA's Remote Agent system; they describe their success in finding the deadlock that nearly crippled the mission 60,000 miles into space in [25, 55].

2.1 Model Checking Programs

As a technique, applying model checking to programs warrants explanation. Model checking was originally applied to hardware verification, and then to small, self contained, deliberately simplified models of a program or protocol's behavior. Here, every feasible state would be examined—either exhaustively or with the aid of mathematical formalisms that permit examining multiple states at once—and various properties about the model's behavior could be verified. It has recently become feasible to run model checkers directly on the program itself, rather than some extracted artifact; checkers like Verisoft [21], CMC [41], and Java PathFinder [54] are examples of model checkers designed to run on code. In this work, I used Java PathFinder, which I abbreviate as JPF.

In model checking programs, our goal is a systematic exploration of every feasible execution path from an initial state. This of course includes exploring all the branches in the program, but also exploring every possible interleaving of the program's threads. This latter property means that if a program is run properly, if a deadlock, data race, or other concurrent error can ever occur it should be found.

There are two major problems in applying this technique. The chief problem is referred to as the *state explosion problem*; there can be an exponentially large number of execution paths, presuming there is even a finite number at all. Numerous techniques such as program slicing, partial order reduction, partial evaluation, and others—which are discussed in [55]—can mitigate this penalty. However the underlying problem is fundamental, and the most successful technique for dealing with it is still limiting the scope of the analysis.

The other major problem is the *environment generation problem*. That is to say, model checking as a technique can only be applied to closed systems, whereas most programs require input from the user and the like. The system must be closed, but it must be closed in a way that does not exclude important behavior, it must be closed in a way that is simple enough so that state explosion does not occur, and frequently parts of the codebase being examined must be replaced by simpler versions.

As such, environment generation is really two problems. First, expensive, irrelevant or difficult to analyze portions of the system must be replaced by equivalent but simpler versions. Second, the entry points to the system must be invoked so that all interesting execution paths are covered. This second half is conceptually very close to generation of test cases.

An example of a difficult to analyze portion of the system might be code that uses JDBC, a Java interface to relational databases. The actual database cannot be analyzed unless it, too, is written in Java. Even if it were, analyzing the database would introduce a vast number of largely superfluous states, exacerbating the state explosion problem. Instead, by replacing JDBC with a lightweight layer that simulates a database, one can successfully analyze the system. This database simulation is deliberately low fidelity and only does what is absolutely necessary for the code to work. This technique is similar in practice to Mock Objects [30], and the goal is similar.

2.1.1 Comparison with Other Styles of Analysis

Before I go into the details of the model checker I used, it is worth noting how model checking relates to more familiar program checking techniques. Accordingly, I briefly contrast dynamic analyses, static analyses, and model checking.

Dynamic analysis is at heart a very simple idea. By instrumenting the program appropriately, one can check important properties while the program is running. This instrumentation has a number of attractive qualities: knowledge of control flow and program state is usually precise, paths that the program cannot take will not be examined, and the results can be very precise. It has one major flaw, which is that only one execution path has been exercised and so if anomalies occur on other execution paths, the analysis may not be able to detect the problem.

Static analyses, by contrast, compute properties without running the program. Most static analyses will examine all possible execution paths. Precise analysis can frequently be infeasible or even impossible, and so almost all static analyses will produce false alarms, or fail to warn about some legitimate issues.

Model checking can be viewed as an extension of dynamic analysis where the execution path issue is resolved by running the analysis through every possible execution path. Viewed in this manner the source of the state explosion problem is very clear; as well, the similarity between environment generation and good test suites is made evident.

2.1.2 Java PathFinder

As alluded previously, Java PathFinder or JPF is a specialized model checker for performing model checking of programs, specifically Java programs. JPF implements a

Java virtual machine, and as such is capable of checking almost any pure Java program. A great deal of effort has gone into enhancing JPF's ability to explore a large number of Java machine states; for more details see [55].

As a practical tool, JPF can be viewed as a nondeterministic Java virtual machine. All control nondeterminism must be made explicit, but concurrent nondeterminism is automatically taken care of. As such, while JPF is not as efficient as dedicated model checkers like SPIN [26] at the problem of actual model checking, it can be used in a very natural way to check that properties in concurrent programs are upheld along any feasible execution path.

JPF handles pure Java code. It cannot model check native methods, although the user may provide pure Java versions for the simulation; in particular this means programs that include file input/output or network traffic are problematic. These holes in JPF's ability to simulate a program can be patched using simplified versions of the relevant classes. This replacement can be done without modifying the code under examination, which is useful.

I found JPF useful because of its ability to run the target software, which is written in Java. Because I was investigating concurrency errors, its ability to try every serialization of the concurrent code was very attractive. If one exercises the code correctly, any deadlocks, races, or other errors that are present—even if they are hard to trigger—should be noticed.

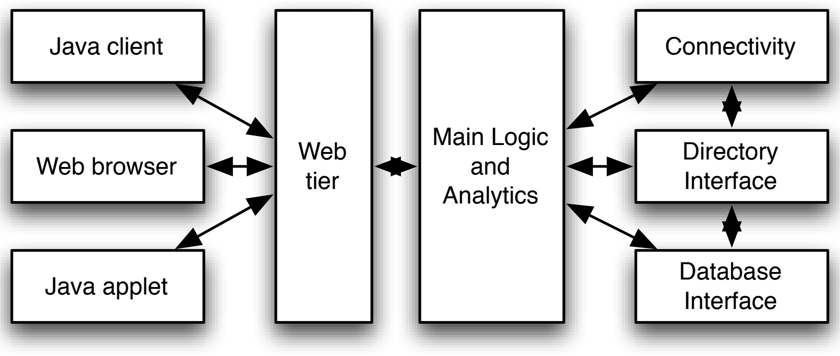


Figure 2.1: Codebase Structure

2.2 Codebase Overview

I applied the techniques discussed in Section 2.1 to the development tree of a large client server codebase. This system possesses a number of different user interfaces and must communicate with several third party database products. The user interface can be implemented either as a Java based thick client, Java applets, or as web pages in a browser. The browser is served by the web tier, which comprises several JSP and servlet components running in a web server. The main process logic and the analytics engine reside in a third tier. The fourth tier contains the underlying repositories such as database, directory, and document management as well as connectivity to other systems using a variety of mechanisms. This structure is shown graphically in Figure 2.1. The various modules communicate through Java RMI [59].

The version of the system I examined comprised well over a thousand classes and a little over 470,000 lines of Java code. Because of its modular design, as well as the fact that the interfaces between the modules are well described through the RMI interfaces, there were very clear divisions within the design. I used these divisions to control how much of the software I would be forced to examine while model checking.

The system's use of RMI had an additional effect. The version of RMI that is shipped with Sun's Java Development Kit, which is the version which was used by the codebase, is implicitly concurrent. The default RMI server creates a new thread for every client. Any software built upon it must be capable of handling that level of concurrency. The system under examination exploits this concurrency to achieve better stability.

This concurrency created a problem during development; debugging complex concurrency code is notoriously difficult, and reproducing deadlocks and race conditions can be very involved. This is true even though in many cases only a handful of threads are required to provoke the problem.

2.3 Applying Model Checking

The stage is now set for analysis; I have a large concurrent program with some difficult to resolve concurrency errors, and a tool that should be good at reliably reproducing these concurrency errors.

When I began my analysis, the development team informed me that there was a deadlock that cropped up rarely under stress tests, and that they had been unable to isolate. Accordingly they were interested if my techniques could prove useful here. I had neither seen or worked with the system prior to this analysis. The size of the project was intimidating, and I did not at the time believe I would be able to analyze substantial portions of the code base due to the state explosion problem.

I believed that the deadlock was probably related to the database interface code. Because of the structure of the codebase, the database interface ran behind an RMI server, entirely separate from the main code. These RMI interfaces seemed a logical place to begin, and we manually wrote the minimal environment required to bring the database server up and ready to serve requests. This environment was developed through an iterative process: the database server would be run, it would immediately fail (in our case, frequently due to some unset global variable), the environment would be modified accordingly, and then rerun. At the time, I believed that this would be a good way to explore the system and to find which parts of it I would have to simulate by hand to avoid state explosion.

It was necessary to replace some Java system classes so that the database server could be brought up under JPF. These replacements included:

- the system RMI package was replaced with one that would not attempt to connect to a network;

- the Date class, which was used by the system's logging mechanism, was modified to always give exactly the same time (thus avoiding a proliferation of program states that were exactly identical save for a timestamp);
- the localization classes were deemed superfluous and stubbed out;
- the JDBC and related database access classes were replaced with a version that was sufficiently high fidelity so the code would work correctly, but that did not store any unnecessary data.

I found that this phase required the most time investment, in large part because I was unfamiliar with the codebase.

2.3.1 Database Concurrency

When the environment generation was complete, I started driving the database server using its RMI interface. The relevant portion of the nominal protocol is as follows:

```
Client acquires DbAdapter interface using RMI
loop
    Client calls DbAdapter.getConnection() to retrieve a DbConnection
    Client uses the database through the DbConnection
    Client releases the DbConnection
```

This protocol is written as a state machine in Figure 2.2.

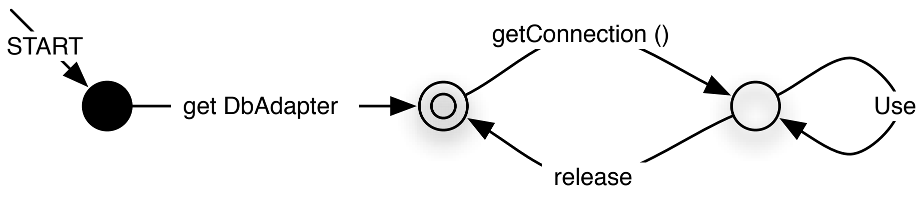


Figure 2.2: RMI protocol in the absence of network disconnects

This protocol, as written, is too simple for real use. Creation of `DbConnection` objects is very slow, so the database server tries to keep a pool of allocated but unused connections around. This pool introduces another problem. If the client does not release its `DbConnection` for any reason—client crashes, network errors, a hung client—the `DbConnection` object it is using will never be returned to the pool. This will cause a resource leak.

To combat this, the database server checks each of the current connections during the `getConnection` call to determine if the original client is still alive. If the server decides that an existing client is dead, then it will hand out an already allocated connection to the new client. The amended protocol that the codebase claims to implement is diagrammed in Figure 2.3.

If the network connection has actually been broken, this behavior is more or less correct. However, client death was determined using the client's idle time. If for any reason—database row locks, database backup, network latency, heavy load, long computations on the client—a client takes more than five minutes to process a transation,

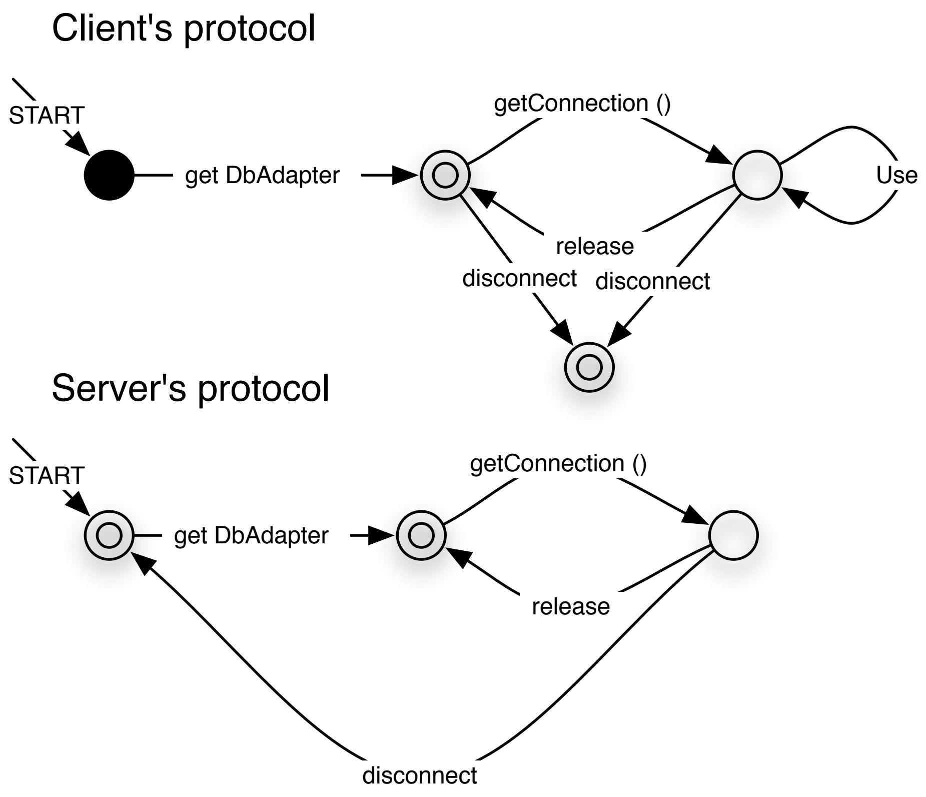


Figure 2.3: RMI protocol supposedly implemented

it runs the risk that its `DbConnection` will be given to some other client. The protocol as implemented is described in Figure 2.4.

If a client should time out and then have its connection given away, it is possible that two clients' transactions could be interleaved. Futhermore, the JDBC standard does not mandate that `java.sql.Connection` objects be thread safe. This can have disastrous consequences, both for thread safety and for database correctness.

This fault only shows up in the presence of concurrent behavior, and while it can show up with just two clients, it is very unlikely that a client's legitimate transactions

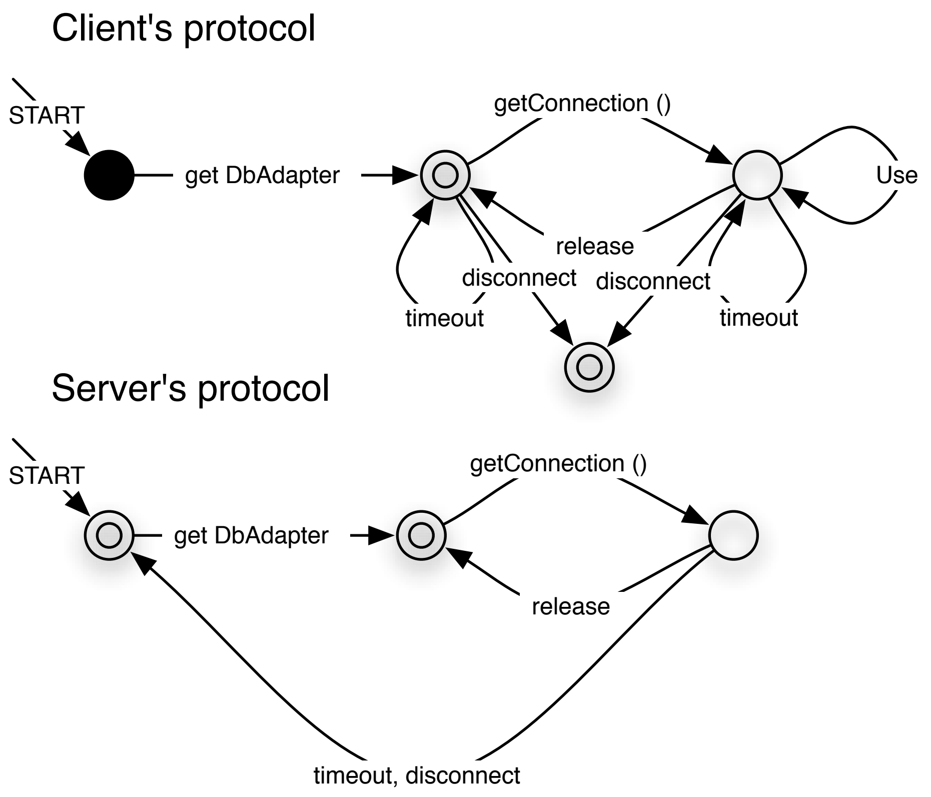


Figure 2.4: RMI protocol actually implemented

will take longer than five minutes except under heavy load. Reliably catching faults like this using a traditional testing methodology would be difficult.

To detect the problem, I created a stub client that fetched a `DbAdapter`, called `getConnection`, used it, and then released the connection. I instantiated several such clients to access the server concurrently. Rather than wait five minutes for timeouts to occur, I replaced the timeout code with a call to JPF's `randomBool` function, which forces JPF to explore the cases when that Boolean value is true as well as when it is false. Thus, I can cover all cases where any client could be disconnected at any time.

Finally, I inserted assertions to check whether any two clients could ever get the same `DbConnection`. JPF proved that this error was possible in only a few minutes, which was a surprising result considering the amount of code that was being run. I expected a state explosion and did not observe it in this case.

To address this issue, I recommended that the database team either use a more certain method of ascertaining whether a client is dead, or restructure the protocol to avoid this sort of behavior.

2.3.2 Cache Verification

After analyzing this portion of the code, at the request of the development team I began analyzing an object cache. This cache was designed to eliminate redundant database accesses.

This code contained numerous simple concurrency errors and poor design decisions; a simplistic static checker like FindBugs [27] could have detected most or all of them. The question of whether any of them could cause data corruption remained open, however. Accordingly, I ran the system with JPF using several threads working on different objects. I discovered in the process of running this that all of the threads were attempting to work on the same database object (object #0, as it happened). Confused, I spent some time trying to determine how our environment was faulty, in the belief that the system could not possibly be misbehaving in this fashion—it had, after all, been running its test cases and in production for years.

Tracing the error, I found that when the object cache reads a process definition from the database, the object cache stores the process definition's unique object identifier in a local field. When a new process definition is created, the identifier has not yet been determined. After the process definition is saved to the database, and thus assigned a unique object identifier, the object cache should update its copy of that object identifier. Instead, the object cache leaves its copy set to the default value of 0.

This bug was found almost immediately during the stress run, but the tests had obviously not been written to catch such an error. I would not have found it had we not discovered several threads which started out with entirely separate process definitions all ending up with the same one, a violation of the cache contract.

2.4 Results

I provide here timing data based on the previous work; specifically the database adapter communications error in Section 2.3.1 and the cache coherency analysis in Section 2.3.2.

The communications protocol flaw is too deeply embedded to be readily fixed; accordingly, I cannot provide verification timing. Detection of the error took 108 seconds, with further data provided in Table 2.1 in the E1 column.

With the concurrent cache, I found the error in 56 seconds; regrettably verification of the fixed but otherwise unmodified cache did not complete in a reasonable amount of time. Further data for this error detecting run is in Table 2.1 in the E2 column. Using manual slicing and with the insertion of appropriate synchronization commands to remove a number of benign races, I verified the cache with two threads in 21 hours. Using very strict synchronization—that is, where every public method of the cache locked the cache class object, ensuring that all cache operations were synchronous—I could verify the cache with two threads in 16 hours. Further details on these are available in Table 2.1 in the V1 and V2 column, respectively.

The gap between the proof-of-error runs and the verification runs is dramatic, but inevitable. Only one erroneous run needs to be produced to find a proof of an error, and the small systems hypothesis in verification is that 'most' bugs can be found on 'most' execution paths; accordingly as soon as an error is found, the process stops. Verification by definition must explore the entire state space of the program.

	E1	E2	V1	V2
States visited	4,710	18,997	19,941,239	10,616,063
Maximum stack depth	4,509	718	1,234	1,232
Memory used (in MB)	128.83	22.62	511.16	285.28
Execution time	108.4 s	55.6 s	20h 54m 55s	15h 54m 04s

Table 2.1: Analysis results

2.5 Methodology

As we briefly discussed in Section 2.3, my methodology is as follows:

1. Find an invariant to check.
2. Try to run the program to check it.
3. If a fault occurs because the environment is insufficiently faithful to the original system, improve the fidelity of the environment and go back to step 2.
4. If the system takes too long to run, find what is taking all the time and replace it with a stub with more abstract behavior. Go back to step 2.
5. If the invariant is violated, check to make sure that it could occur in the original system, and if so, report a violation. If not, correct the problem by improving the fidelity of the system and go back to step 2.
6. If no invariants are violated, check to make sure that your driving system is adequately faithful. If it is, report success. Otherwise, improve the fidelity of the system accordingly and go back to step 2.

This method is not an algorithm in the sense that these steps are not mechanical processes; the correct thing to do requires experience with model checking and with the system. Also, whether the method will terminate is questionable. Almost all of the effort is spent doing environment generation, not checking as such.

Some of the classes that I had to replace with stubs were required either as a logical consequence of the limitations of JPF—e.g., RMI—or required because of linkage to a third party product—e.g., JDBC. Any analysis on this code base will almost certainly have to stub classes in those two categories out, because JPF does not and cannot handle network code directly in its current form, and because simulating a relational database will almost certainly take far too long.

It would be very helpful if this entire process could be automated, and indeed the SLAM efforts mentioned earlier do automate a similar procedure for single threaded C programs. They use predicate abstraction, which has is fully automatic, but it is not clear how to extend their technique to multithreaded programs.

A tool to automate this environment generation would be very useful. This is an active research question, which I discuss in Section 10.2. These techniques are very interesting but in their current form they would not have been valuable to me because they explore the program only to a bounded depth in the program's call tree. Due to the system's internal structure, I feel I would need to set the bound depth too high to make the technique useful.

A different but no less important question is that of 'how did I know where to look' and the related question of 'what sorts of questions are appropriate for using model checking'. In my case, I were directed to problems by the development team that specified in one case a problem they had been having a great deal of trouble finding, and in the other case a request to verify that a specific module was sound. In both cases, I extracted a state machine describing the input of the module in question, and then explored that state machine looking for violations of simple properties; properties like 'do any two clients ever share a `DbConnection`' and 'can the cache fall out of synchronization with the database'.

Because the systems in question used concurrency extensively, either implicitly as any RMI client does, or explicitly in the case of the cache, testing these properties is difficult and prone to subtle timing issues. Testing also has trouble answering liveness properties—properties like "if a client has a `DbConnection` it will eventually release it". Accordingly I believe that an appropriate use of a model checker is the systematic exploration of the interface to a module while examining important invariants.

2.6 Summary

I demonstrated that significant analyzses can be performed using model checking on preexisting software, even software that is not designed for model checking. I expected to run into the major problem bedeviling analysis of large programs using model checking—the state explosion problem—and largely did not. Environment generation, on the other hand, required the vast majority of the effort. Accordingly, I examined the problem of environment generation further, culminating in my interface grammar language described next.

Chapter 3

Modular Verification with Interface Grammars: An Overview

The previous chapter demonstrates that verification of real world programs is feasible, but has some unexpected consequences. Verification took less than a day but it took me over two weeks per major program section to get the system to the point where it could be verified.

There are two major reasons for this. No model checker can verify programs that perform network communication, filesystem access, graphical interfaces or generally interaction with a user—in each of these cases a simpler model of the offending component must be written, and the original program must be modified to use this new code. At the same time, the original program usually does a wide variety of things that are totally irrelevant to verification; these must be worked around. These two problems can be grouped together as the "environment generation problem", and is a subject of active

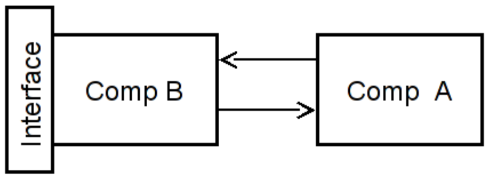


Figure 3.1: Broad overview

research. However, previous attempts at environment generation focused on doing the operation automatically and were too simple to be useful for me.

I have developed an alternate approach to the environment generation problem, and have used and extended this approach to verify nontrivial systems. Here I present an overview of how the approach works before delving into the details.

Consider a system composed of two *components*. A component here is a set of objects. I consider only single threaded systems. Assume that the interaction between two components is totally determined by the messages the components send to each other, as in figure 3.1. We can call this the *conversation*. Here calling $x.\mathit{foo}(a, b, c)$ when *this* is in one component and *x* is in another component sends the message *foo* with the arguments x, a, b, c. I model the return from this method call as a message in the other direction. If this model captures the way these two components interact, then we can replace one or the other with a stub that sends and receives the same messages, as in figure 3.2 on the next page.

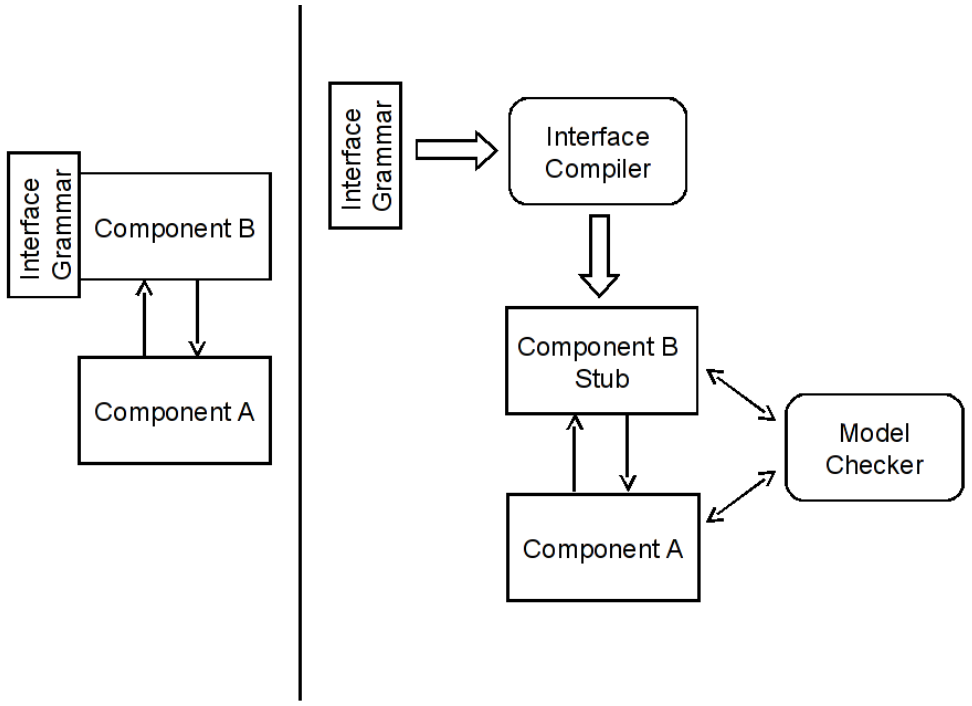


Figure 3.2: Operation of the interface compiler

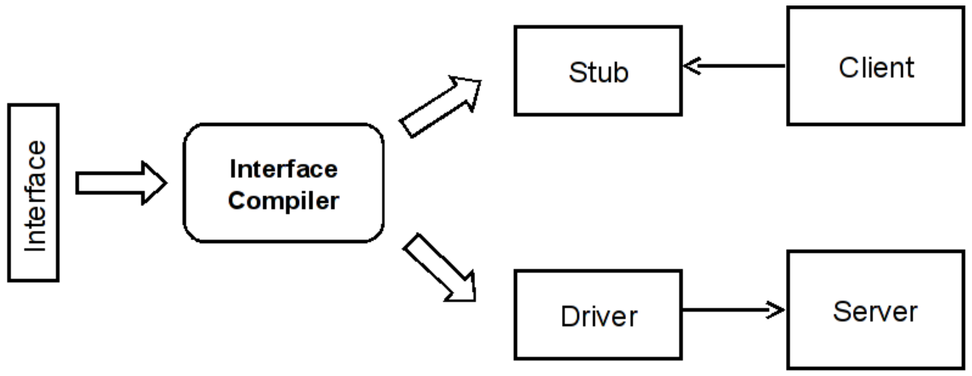


Figure 3.3: Generating stubs and drivers

I do not envision this as a fully automatic operation. Fully automatic operation comes with numerous caveats and scope restrictions that make it difficult to apply to real programs, which is my focus. But at the same time doing this by hand is no improvement. I envision something else; someone wanting to perform verification should write an *interface grammar*, a concise description of the possible conversations one component can have with another component. This grammar is written in the manner of a context-free grammar. My interface compiler will then create a stub that will verify that the opposite component responds to messages in the way depicted, and participate in that conversation effectively.

Up to now, this conversation has been entirely symmetric; either component could be stubbed. For the purpose of discussion, the component that we have instructed the compiler to stub out is called the *client* and the other component is the *rest of the program*. The generated code is termed the *stub*, or sometimes the *driver* if the interface grammar mostly specifies method calls from the driver to the rest of the program. These are depicted in figure 3.3 on the preceding page. All these are arbitrary terms to ease understanding; the formalism, the compiler, and the language itself do not enforce any of these distinctions.

Chapter 4

Defining Interface Grammars

I propose interface grammars as a language for specification of component interfaces. The core of an interface grammar is a set of production rules that specifies all acceptable method call sequences for the given component. Given an interface specification for a component, our interface compiler generates a stub for that component. This stub is a table-driven top-down parser [1] that parses the sequence of incoming method calls (i.e., the method invocations) based on the interface grammar defined by the interface specification.

For example, consider a component for transaction management with the following methods: `begin`, which begins a transaction; `commit` which commits a transaction; and `rollback` which rolls back a transaction. Now consider the following (simplified) interface grammar:

Grammar 1. Simple transaction grammar

$$\begin{array}{rcl} \textit{Start} & \rightarrow & \textit{Inactive} \\ \textit{Inactive} & \rightarrow & \texttt{begin}\ \textit{Active} \\ & | & \epsilon \\ \textit{Active} & \rightarrow & \texttt{commit}\ \textit{Inactive} \\ & | & \texttt{rollback}\ \textit{Inactive} \end{array}$$

This is a context free grammar with the nonterminal symbols *Start*, *Inactive*, and *Active*; the start symbol *Start*; and terminal symbols `begin`, `commit`, and `rollback`. Note that this grammar specifies a language that consists of sequences of symbols `begin`, `commit`, and `rollback`. In our framework, this language corresponds to the set of acceptable incoming call sequences for a component, i.e., the interface of the component. According to the above interface grammar, the first call to the transaction component must be a `begin` call which then should be followed by a `commit` or a `rollback` call.

Given the above grammar we can construct a parser which can serve as a stub for the transaction component. This stub/parser will simply use each incoming method call as a lookahead symbol and implement a table driven parsing algorithm. If at some point during the program execution the stub/parser cannot continue parsing, then we know that we have caught an interface violation.

However, the simple interface example I gave above does not require the power of grammars. The same interface can be specified using finite state machines. Instead, consider a transaction manager that allows nested transactions (also known as subtransactions). In nested transactions a subtransaction can begin within the scope of another transaction, hence allowing only a subset of the operations of the parent transaction to be rolled back in case of an error. The following interface grammar specifies the interface for the nested transaction manager:

Grammar 2. Nested transaction grammar

$$\begin{array}{rcl} \mathit{Start} & \rightarrow & \mathit{Base} \\ \mathit{Base} & \rightarrow & \texttt{begin}\ \mathit{Base}\ \mathit{Tail}\ \mathit{Base} \\ & | & \epsilon \\ \mathit{Tail} & \rightarrow & \texttt{commit} \\ & | & \texttt{rollback} \end{array}$$

Note that this interface specification allows nesting of matching `begin` and `commit` or `rollback` calls and, therefore, cannot be expressed using finite state machines.

Our interface specification language also supports specifying semantic predicates and semantic actions that can be used to write complex interface constraints. A semantic predicate is a piece of code that can influence the parse, whereas a semantic action is a piece of code that is executed during the parse. Semantic predicates and actions provide a way to escape out of the interface grammar framework and write Java code that becomes part of the component stub. The semantic predicates and actions are inserted to the right hand sides of the production rules, and they are executed at the appropriate time during the program execution (i.e., when the parser finds them at the top of the parse stack).

To demonstrate the use of semantic predicates and actions, I add to the nested transaction manager the `setRollbackOnly` method which forces all pending transactions to finish with `rollback` instead of `commit`. The method `setRollbackOnly` can only be invoked if there is an active transaction, and after it is invoked, the only way to finish the pending transactions is to invoke `rollback`. I will add a r global Boolean variable to keep track of the rollback-only state, and a l global variable to keep track of how many pending transactions are active; if $l \equiv 0$ then r is reset to false. If we denote the semantic action containing the code `x` as $\langle\!\langle \mathtt{x} \rangle\!\rangle$ and the semantic predicate evaluating the code `p` as $[\![\mathtt{p}]\!]$, then the amended grammar looks as follows:

Grammar 3. Nested transaction grammar with semantic elements

$$\begin{array}{rcl}
\textit{Start} & \rightarrow & \langle\!\langle r \leftarrow \mathbf{false}; l \leftarrow 0 \rangle\!\rangle\ \textit{Base} \\
\textit{Base} & \rightarrow & \texttt{begin}\ \langle\!\langle l \leftarrow l+1 \rangle\!\rangle\ \textit{Base Tail} \\
 & & \langle\!\langle l \leftarrow l-1; \mathbf{if}\ l \equiv 0\ \mathbf{then}\ r \leftarrow \mathbf{false} \rangle\!\rangle\ \textit{Base} \\
 & | & \texttt{setRollbackOnly}\ \langle\!\langle r \leftarrow \mathbf{true} \rangle\!\rangle\ \textit{Base} \\
 & | & \epsilon \\
\textit{Tail} & \rightarrow & [\![r \equiv \mathbf{false}]\!]\ \texttt{commit} \\
 & | & \texttt{rollback}
\end{array}$$

To summarize, the call sequences specified by grammar 1 on page 29 can also be specified using a finite state machine. However, the call sequences for recursive transactions specified by grammar 2 on page 31 and grammar 3 on the previous page cannot be specified using finite state machines.

4.1 Semantics

The above description of interface grammars is informal; here I present a formal definition. I define my grammars based on recognizing possible execution traces. I restrict my focus to method calls between a component and the rest of the program, and further on method calls from the program to the component, and the returns of these calls. I denote the initiation of a method call from a program into the component for the method a by $?a$, and I denote the termination of that method call—that is, the return—by $¿a$. For accurate modeling of a component, I must also track the method arguments and the return values. I assume that methods have one argument and one return value; multiple arguments are represented as a tuple. I write the initiation of a method call a with argument x by $?a[x]$ and the termination of that method call with return value y by $¿a[y]$. I frequently write the initiation of a method call that takes no arguments as $?a[]$ or $?a[\bot]$, and the termination of a method call that has a void return value as $¿a[]$ or $¿a[\bot]$.

These traces serve to formalize interface grammars. To define those grammars, I use the following notation. My notation is based on that of Nielson, et al. [42]. The fundamental sets I deal with here are set in **boldface**. I denote the Kleene closure of a set S by S^*. The empty sequence is denoted ϵ. I write the concatenation of two sequences s_0 and s_1 by $s_0 \| s_1$; in an abuse of notation I write the concatenation of a singleton x with a sequence s by $x \| s$. Since I never deal with sequences directly containing sequences, this is unambiguous. I use $\leftarrow$ for a decomposing assignment: thus after $\langle x, y \rangle \leftarrow \langle 1, 2 \rangle$ x is set to 1 and y is set to 2. I write the partial function f mapping some of the set X to the set Y as $f : X \overset{\mathrm{p}}{\to} Y$. For a function $f : X \to Y$ and a set $S \subseteq X$, I write the image of f under S as $f[S]$. For a function $f : X \to Y$ and arbitrary x and y, I write the function $f' : X \cup \{x\} \to Y \cup \{y\}$ where $f'(x) = y$ and $f'(a) = f(a)$ otherwise as

$f[x \mapsto y]$. Similarly $f[x \mapsto y, a \mapsto b] = f[x \mapsto y][a \mapsto b]$. For functions $f : X_0 \to Y_0$ and $g : X_1 \to Y_1$, I define the function $f \odot g : X_0 \cup X_1 \to Y_0 \cup Y_1$ to be the function $\forall x \in \mathrm{dom}(f) : f \odot g(x) = f(x)$ and $\forall x \in \mathrm{dom}(g) \setminus \mathrm{dom}(f) : f \odot g(x) = g(x)$. I write the function $f : \emptyset \to \emptyset$ as $\emptyset$. I will occasionally, in the cause of increased readability, use a decomposing conditional, writing **if** $(o = ?x(v))$ **then** s when I really mean **if** $\mathrm{first}(o) = ?x$ **then** $\langle ?x, v\rangle \leftarrow o; s$. I write the anonymous function that computes an expression e with free variable a as $\lambda a.e(a)$. I write a temporary variable assignment as **let** $x = y$ **in** z; this is equivalent to $(\lambda x.z(x))(y)$, but this way of writing it is less opaque.

I require several sets to define the semantics of my grammars. Accordingly, I presume the following ground sets: $\mathbf{B} = \{\textbf{true}, \textbf{false}\}$ is the set of Boolean values; $S \in \mathbf{NT}$ is a member of the set of nonterminals; $?a, ¿a, !a, ¡a \in \mathbf{\Sigma}$ are members of the alphabet of method calls and returns, where $?a$ denotes a call to a method a, $¿a$ denotes the return from a method a, $!a$ denotes an outgoing method call a, and $¡a$ denotes the return from an outgoing method call a; $\square \in \mathbf{\Sigma}$ denotes the end of parsing, that is the termination of the program; $\xi \in \mathbf{Loc}$ is a member of the set of locations where I may store values of variables; $x, y \in \mathbf{Var}$ are variables; and $v \in \mathbf{Dom}$ is an unconstrained domain set that represents the values the variables can take. Using these I now define the following sets:

$$
\begin{aligned}
S[x] &\in \mathbf{NT}_\circ \equiv \mathbf{NT} \times \mathbf{Var} \\
\rho &\in \mathbf{Env} \equiv \mathbf{Var} \xrightarrow{\text{p}} \mathbf{Loc} \\
\varsigma &\in \mathbf{Store} \equiv \mathbf{Loc} \xrightarrow{\text{p}} \mathbf{Dom} \\
\sigma &\in \mathbf{State} \equiv \mathbf{Var} \xrightarrow{\text{p}} \mathbf{Dom} \\
\langle\!\langle f \rangle\!\rangle &\in \mathbf{Action} \equiv \mathbf{State} \to \mathbf{State} \\
[\![p]\!] &\in \mathbf{Pred} \equiv \mathbf{State} \to \mathbf{B} \\
\Delta x &\in \mathbf{Decl} \subseteq \mathbf{Var} \\
s_1, s_2 \ldots &\in \mathbf{Sym} \equiv \mathbf{NT}_\circ \cup \mathbf{\Sigma} \cup \mathbf{Action} \cup \mathbf{Pred} \cup \mathbf{Decl} \cup \{\uparrow, \downarrow\} \\
A, B, C &\in \mathbf{Sym}^* \\
\mathbf{Prod} &\equiv \mathcal{P}(\mathbf{NT}_\circ \times \mathbf{Sym}^*)
\end{aligned}
$$

Here ρ, ς and σ are partial functions; not every location must be assigned a value, nor must every variable be bound to a location. In practice I will construct elements of $\mathbf{State}$ by composing elements of $\mathbf{Env}$ and $\mathbf{Store}$; so $\sigma = \varsigma \circ \rho$. Note that as written these semantics use dynamic scoping instead of lexical scoping; this is inevitable because at this low level there is no lexical information to be had, but I discuss the way I layer lexical scoping on top of it in chapter 6 on page 79. I use $\uparrow$ and $\downarrow$ to denote opening and closing a new scope, respectively. Also note that our nonterminals $S[x]$ here have arguments, reflecting a need for additional expressiveness when working with data structures. While this does not increase the power of our grammars—it could be done entirely through semantic actions—it does make it easier to express certain concepts. I use in-out parameter semantics for nonterminal arguments; that is, when $S[x]$ is being processed the corresponding argument y is bound to the value of x, and when processing of the nonterminal is complete x is bound to the value of y.

Unfortunately the above definition of $\mathbf{\Sigma}$ is not really sufficient; specifically the traces must record the method arguments and method return values. I define the set $\mathbf{\Sigma}_\circ \equiv \mathbf{\Sigma} \times \mathbf{Var}$ to be the set of symbols combined with variable names to denote the arguments or return values as appropriate, and $\mathbf{\Sigma}_\bullet \equiv \mathbf{\Sigma} \times \mathbf{Dom}$ to be the set of symbols combined with values that represent the record of a trace. To ease the presentation, I write $\langle ?a, x\rangle \in \mathbf{\Sigma}_\circ$ as $?a(x)$, and $\langle ?a, d\rangle \in \mathbf{\Sigma}_\bullet$ as $?a[d]$. Similarly I need to define

$$\mathbf{Sym}_\circ \equiv \mathbf{NT}_\circ \cup \mathbf{\Sigma}_\circ \cup \mathbf{Action} \cup \mathbf{Pred} \cup \mathbf{Decl} \cup \{\uparrow, \downarrow\}$$

and $\mathbf{Prod}_\circ \equiv \mathcal{P}(\mathbf{NT}_\circ \times \mathbf{Sym}_\circ^*)$.

From this I can define an interface grammar G as a tuple

$$G = \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S\rangle$$

with $\mathbf{Q} \subseteq \mathbf{State}$ being the grammar states, $\mathbf{SA} \subseteq \mathbf{Action}$ being the semantic actions used in the grammar, $\mathbf{SP} \subseteq \mathbf{Pred}$ being the semantic predicates used in the grammar, $\mathbf{P} \subseteq \mathbf{Prod}_\circ$ being the production rules, and $S \in \mathbf{NT}$ being the start symbol. I would like to define a method for derivation for this grammar, so that I can fully describe whether a sentence is in the language of the grammar. I define single-step derivation ($\Rightarrow$) and ultimate derivation ($\Rightarrow^*$) as binary relations in figure 4.1 on the next page. These definitions are nonconstructive; constructive versions follow in chapter 5 on page 48.

The signatures for $\Rightarrow$ and $\Rightarrow^*$ are as follows. On the left side, first a string of trace symbols $\mathbf{\Sigma}_\bullet^*$. Next, a $\mathbf{Env}$ and $\mathbf{Store}$, representing the current environment and store values. Finally, a string of grammar symbols that are not yet processed $\mathbf{Sym}_\circ^*$. The right side is either a string of trace symbols $\mathbf{\Sigma}_\bullet^*$, denoting that the derivation has used equation (4.1) on the next page and is in a state where the string can be accepted; or a string like the left side, meaning more work has to be completed before a derivation can be accepted. During the process of derivation, I must keep track of the environment and store both for the semantic predicates to function, and I must keep track of them separately rather than conflating them into a single $\mathbf{State}$ to permit lexical scoping of declarations.

$$\Rightarrow, \Rightarrow^* \in \left(\mathbf{\Sigma}_{\bullet}^* \times \mathbf{Env} \times \mathbf{Store} \times \mathbf{Sym}_{\circ}^*\right) \times \left(\mathbf{\Sigma}_{\bullet}^* \cup \left(\mathbf{\Sigma}_{\bullet}^* \times \mathbf{Env} \times \mathbf{Store} \times \mathbf{Sym}_{\circ}^*\right)\right)$$

$$A\,\{\rho,\varsigma\}\;\Rightarrow A \tag{4.1}$$

$$\frac{\mathrm{dom}(\mathrm{dom}(f)) \subseteq \mathrm{dom}(\rho) \quad \varsigma' = (\rho^{-1} \circ f(\varsigma \circ \rho)) \odot \varsigma}{A\,\{\rho,\varsigma\}\,\langle\!\langle f \rangle\!\rangle B \Rightarrow A\,\{\rho,\varsigma'\}\,B} \tag{4.2}$$

$$\frac{\mathrm{dom}(\mathrm{dom}(p)) \subseteq \mathrm{dom}(\rho) \quad p(\rho \circ \varsigma) = \textbf{true}}{A\,\{\rho,\varsigma\}\,[\![p]\!]\,B \Rightarrow A\,\{\rho,\varsigma\}\,B} \tag{4.3}$$

$$\frac{\xi \notin \mathrm{dom}(\varsigma)}{A\,\{\rho,\varsigma\}\,\Delta x\,B \Rightarrow A\,\{\rho[x \mapsto \xi], \varsigma[\xi \mapsto \bot]\}\,B} \tag{4.4}$$

$$\frac{\begin{array}{c}\langle S[x], s_1 \ldots s_n \rangle \in \mathbf{P} \quad \xi \notin \mathrm{dom}(\varsigma) \\ \{\rho[x \mapsto \xi], \varsigma[\xi \mapsto \varsigma \circ \rho(y)]\}\,s_1 \ldots s_n \Rightarrow^* C'\,\{\rho',\varsigma'\}\end{array}}{A\,\{\rho,\varsigma\}\,S[y]\,B \Rightarrow A\,C'\,\{\rho, \varsigma[\rho(y) \mapsto \varsigma' \circ \rho'(x)]\}\,B} \tag{4.5}$$

$$\frac{\{\rho,\varsigma\}\,B \Rightarrow^* B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\,\uparrow B \downarrow C \Rightarrow A\,B'\,\{\rho,\varsigma'\}\,C} \tag{4.6}$$

$$\frac{\xi_1, \xi_2 \notin \mathrm{dom}(\varsigma) \quad \{\rho[x \mapsto \xi_1, y \mapsto \xi_2], \varsigma[\xi_1 \mapsto v, \xi_2 \mapsto \bot]\}\,B \Rightarrow^* B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\,?a(x)\,B\,¿a(y)\,C \Rightarrow A\,?a[v]\,B'\,¿a[\varsigma' \circ \rho'(y)]\,\{\rho,\varsigma'\}\,C} \tag{4.7}$$

$$\frac{\xi \notin \mathrm{dom}(\varsigma) \quad \{\rho,\varsigma\}\,B \Rightarrow^* B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\,!a(x)\,B\,¡a(y)\,C \Rightarrow A\,!a[\varsigma \circ \rho(x)]\,B'\,¡a[v]\,\{\rho[y \mapsto \xi], \varsigma'[\xi \mapsto v]\}\,C} \tag{4.8}$$

$$\frac{\exists u_1, \ldots u_n : A\,\{\rho,\varsigma\}\,B\,C \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n \Rightarrow A\,B'\,\{\rho',\varsigma'\}\,C}{A\,\{\rho,\varsigma\}\,B\,C \Rightarrow^* A\,B'\,\{\rho',\varsigma'\}\,C} \tag{4.9}$$

$$\frac{\exists u_1, \ldots u_n : A\,\{\rho,\varsigma\}\,B \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n \Rightarrow A\,B'}{A\,\{\rho,\varsigma\}\,B \Rightarrow^* A\,B'} \tag{4.10}$$

Figure 4.1: Derivation rules

Equation (4.1) on the preceding page defines the derivation when the end of a string has been reached; I drop the scoping information ρ and the store information ς and accept.

Equation (4.2) on the previous page defines the derivation rule for a semantic action $\langle\!\langle f \rangle\!\rangle$. Since f maps $\mathbf{State}$ to $\mathbf{State}$, $\mathrm{dom}(f) = \mathbf{State} = \mathbf{Var} \rightarrow \mathbf{Dom}$ and $\mathrm{dom}(\mathrm{dom}(f))$ is just the variables it uses. Since it is syntactially possible for f to refer to variables that are not in $\mathrm{dom}(\rho)$, that is variables that are not in scope, I disallow derivation unless all variables f uses are in scope. $\varsigma \circ \rho \in \mathbf{State}$ is the current state, and $f(\varsigma \circ \rho)$ is the new state. However, I need an updated store, ς', rather than a state. I can reconstruct f's changes to the state by applying ρ^{-1}, but this will ignore the part of the state that f may not have been able to see and so I must adjoin it to the old store with $\odot$. I justify the existence of ρ^{-1} by noting that over the subset of $\mathbf{Var}$ where ρ is totally defined, it is one-to-one; it is only modified in equation (4.7) on the preceding page and equation (4.4) on the previous page, and there ξ, ξ_1 and ξ_2 are constrained to be values that have never been in $\mathrm{ran}(\rho)$.

It is concievable that $\varsigma \circ \rho$ may not be well defined; that is, $\mathrm{ran}(\rho) \not\subseteq \mathrm{dom}(\varsigma)$. I in fact guarantee that $\varsigma \circ \rho$ is well defined by construction; no matter what the derivation, $\mathrm{dom}(\varsigma)$ never shrinks, and each time I add a new element to $\mathrm{ran}(\rho)$—that is, each time I map a variable to a new location—I map that same location to a value in ς. So $\varsigma \circ \rho$ is always well defined.

Equation (4.3) on page 39 is similar to equation (4.2) on page 39, but as semantic predicates may not modify the state, it is somewhat simpler. Again $\mathrm{dom}(\mathrm{dom}(p))$ is the variables p uses, which I insist be a subset of the variables in scope. Similarly I insist that the predicate p be true in the current state.

Equation (4.4) on page 39 deals with variable declarations. I insist that ξ here be a fresh location, one that has never been assigned to any variable; since every location that has even been assigned to a variable is in $\mathrm{dom}(\varsigma)$ this is easily achieved. I must update ρ to reflect that x has been bound to the location ξ, and then bind the location ξ to $\perp$, reflecting that it has not yet been assigned a value.

Equation (4.5) on page 39 defines nonterminal substitution; for any production $S[x] \to s_1 \ldots s_n$ in $\mathbf{P}$, which is equivalently stated $\langle S[x], s_1 \ldots s_n \rangle \in \mathbf{P}$, I must bind x to the value of its argument (here computed to be $\varsigma \circ \rho(y)$), perform the derviation on the production, and finally rebind the value of the argument y to be the final value of x, since I have specified in-out parameter semantics. I use an auxiliary location ξ for this purpose. The final value of x is computed to be $\varsigma' \circ \rho'(x)$, and I need to set the location of y (which is just $\rho(y)$) to that value.

Equation (4.6) on page 39 defines block semantics; for a matched pair of $\uparrow$ and $\downarrow$, if we can derive B' with scope ρ' and store ς' from B with the original scope ρ and store ς, then we may continue with the original scope ρ (reflecting block scoping rules) and the new store ς' (reflecting any changes that may have been made to the store in B).

Equation (4.7) on page 39 defines method call semantics; given a matched pair $?a(x)$ (meaning the incoming method call a with argument x) and $¿a(y)$ (meaning the return from that method call, returning the value of y), I must first bind $?a$'s argument x to the value seen (here v) and bind the return variable y into a new scope. To do this I need two new locations ξ_1 and ξ_2. Now, if B with the original scope augmented with x and y derives B' with scope ρ' and store ς', I can derive the original method call, retaining the old scope but using the new store as in equation (4.6) on page 39 above. But, I must reconstruct the return value for $¿a$; this is simply the value of y in the state following the derivation of B', which is $\varsigma' \circ \rho'$. I record v and $\varsigma' \circ \rho'(y)$ for the trace in the result, using the shorthand defined above for $\Sigma_\bullet$. Note that this equation enforces call and return matching. This is due to a domain requirement—in most programming languages it is not syntactically possible to make a subroutine call and never return, or to return many times.

Equation (4.8) on page 39 defines method send semantics; it is similar in many ways to equation (4.7) on page 39, in that given a matched pair $!a(x)$ (meaning the outgoing method call a with argument x) and $¡a(y)$ (meaning the return from that method call, returning the value of y), we must first compute the value of the argument x (here $\varsigma \circ \rho(x)$), perform whatever subderivations are necessary, and then bind the return value (here v) to the variable y so that the return value may be visible. This requires an auxiliary location ξ. I have not here explicitly defined what the *target* of these method calls are; we may understand it to be the argument, or the first element of the argument tuple. Note that this equation, as with equation (4.7) on page 39, enforces that calls and returns match.

Finally, equation (4.9) on page 39 and equation (4.10) on page 39 define how to define multiple step derivation from the above rules. We need two rules to permit equation (4.1) on page 39 to be used.

Now, given all this I can finally define the language of our grammar; a string $A \in \mathbf{\Sigma}_\bullet^*$ is in $L(G)$ if and only if $\{\emptyset, \emptyset\}\ S \Rightarrow^* A$. If I want to accommodate global variables, I can say a string $A \in \mathbf{\Sigma}_\bullet^*$ is in $L(G[\rho, \varsigma])$ if and only if $\{\rho, \varsigma\}\ S \Rightarrow^* A$; here the global variables would be defined in the ρ and ς accordingly.

4.2 An example

An example is in order. Consider grammar 3 on page 32. I restate it formally here as the grammar G, where:

$$G = \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle$$

$$\mathbf{NT} = \{\textit{Start}, \textit{Base}, \textit{Tail}\}$$

$$\mathbf{\Sigma} = \{?\texttt{begin}, ?\texttt{setRollbackOnly}, ?\texttt{commit}, ?\texttt{rollback}, ¿\texttt{begin}, ¿\texttt{setRollbackOnly}, ¿\texttt{commit}, ¿\texttt{rollback}\}$$

$$\mathbf{Var} = \{r, l\}$$

$$\mathbf{Dom} = \mathbf{B} \cup \mathbb{Z}$$

$$\begin{aligned}
\mathbf{SA} = \; & \{\langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle, \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle, \\
& \langle\!\langle \lambda\sigma.\textbf{let}\, \sigma' = \sigma[l \mapsto \sigma(l) - 1]\, \textbf{in}\, \sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle, \\
& \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}] \rangle\!\rangle\}
\end{aligned}$$

$$\mathbf{SP} = \{[\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\}$$

$$\begin{aligned}
\mathbf{P} = \; & \{\langle \textit{Start}, \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle\; \textit{Base} \rangle, \\
& \langle \textit{Base}, ?\texttt{begin}()\; \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle\; ¿\texttt{begin}()\; \textit{Base Tail} \\
& \quad \langle\!\langle \lambda\sigma.\textbf{let}\, \sigma' = \sigma[l \mapsto \sigma(l) - 1]\, \textbf{in}\, \sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle, \\
& \quad \textit{Base} \rangle, \\
& \langle \textit{Base}, ?\texttt{setRollbackOnly}()\; \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}] \rangle\!\rangle \\
& \quad ¿\texttt{setRollbackOnly}()\; \textit{Base} \rangle, \\
& \langle \textit{Base}, \epsilon \rangle, \\
& \langle \textit{Tail},\; [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\; ?\texttt{commit}()\; ¿\texttt{commit}() \rangle, \\
& \langle \textit{Tail},\; ?\texttt{rollback}()\; ¿\texttt{rollback}() \rangle\}
\end{aligned}$$

As a notational convenience, when a method has no arguments rather than writing it as $?a(x)$ and then not using x, I write it as $?a()$ or $?a[]$. Similarly, when a method has a void return value, I write it as $¿a()$ or $¿a[]$. All the methods in this example take no arguments and have void returns. I use the syntax $\lambda x.y$ to denote the anonymous function taking one argument, x, and performing y.

In constructing this formal grammar, I have distinguished method calls and returns, which grammar 3 on page 32 conflated. I distinguish returns rather than conflating them with calls for two reasons: first, because the point at which a method returns has control flow implications in the system external to our component. Second, because I must track the return values for each method in the trace, and the only way to do that is to mark method returns in some fashion.

I assert that the trace

$$\begin{aligned} t_1 = {} & ?\texttt{begin}[]\,¿\texttt{begin}[]\,?\texttt{begin}[]\,¿\texttt{begin}[] \\ & ?\texttt{commit}[]\,¿\texttt{commit}[]\,?\texttt{rollback}[]\,¿\texttt{rollback}[] \end{aligned}$$

is in $L(G)$, or rather is in $L(G[\rho_0, \varsigma_0])$ where $\rho_0 = [r \mapsto \xi_0, l \mapsto \xi_1]$, and $\varsigma_0 = [\xi_0 \mapsto \bot, \xi_1 \mapsto \bot]$ since r and l are both global variables.

To prove this assertion, I begin a derivation from $\{\rho_0, \varsigma_0\}\, S$, as follows:

$$\begin{aligned}
&\{\rho_0, \varsigma_0\}\, S \\
\Rightarrow\ &\{\rho_0, \varsigma_0\}\, \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle\ \textit{Base} && \text{by 4.5} \\
\Rightarrow\ &\{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ \textit{Base} && \text{by 4.2} \\
\Rightarrow\ &\{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ ?\texttt{begin}()\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle\ ¿\texttt{begin}() && \text{by 4.5} \\
&\textit{Base Tail}\ \langle\!\langle \lambda\sigma.\textbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l) - 1]\,\textbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle \\
&\textit{Base}
\end{aligned}$$

To proceed, I need to apply equation (4.7) on page 39; to do that I need to perform a subderivation as follows:

$$\begin{aligned}
&\{\rho_0[x \mapsto \xi_2, y \mapsto \xi_3], [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0, \xi_{2\ldots3} \mapsto \bot]\} \\
&\langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle \\
\Rightarrow\ &\{\rho_0[x \mapsto \xi_2, y \mapsto \xi_3], [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 1, \xi_{2\ldots3} \mapsto \bot]\} && \text{by 4.2}
\end{aligned}$$

Now, I can apply equation (4.7) on page 39:

$$\{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ \texttt{?begin}()\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle\ \texttt{¿begin}()$$

$$\textit{Base Tail}\ \langle\!\langle \lambda\sigma.\textbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l)-1]\,\textbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle$$

$$\textit{Base}$$

$$\Rightarrow \texttt{?begin}[]\ \texttt{¿begin}[]\ \{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 1, \xi_{2\ldots3} \mapsto \bot]\}\ \textit{Base Tail} \qquad \text{by 4.7}$$

$$\langle\!\langle \lambda\sigma.\textbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l)-1]\,\textbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle\ \textit{Base}$$

Continuing in this same vein, I eventually derive that

$$\{\rho_0, \varsigma_0\}\, S \Rightarrow^{*}\ \texttt{?begin}[]\ \texttt{¿begin}[]\ \texttt{?begin}[]\ \texttt{¿begin}[]$$

$$\texttt{?commit}[]\ \texttt{¿commit}[]\ \texttt{?rollback}[]\ \texttt{¿rollback}[]$$

which is precisely the trace t_1. Therefore $t_1 \in L(G[\rho_0, \varsigma_0])$.

Chapter 5

Parsing Interface Grammars

The preceding semantics, while useful, is nonconstructive. I present here an algorithm that is an implementation of those semantics. To do so, I need to note a pair of issues that must be addressed in any implementation of the semantics, but that the semantics themselves do not discuss. Specifically:

- The semantics above are defined for a single threaded program; therefore there is a single thread of control that must be accounted for. Who, then, possesses the thread of control as execution begins? I call this the control problem.

- Which of the component and the rest of the program initially has the thread of control. I call this the main problem.

We can consider the control problem as answering the question ‘at any given point in a trace, where is the thread of control’. This has a straightforward answer: beginning at ?a and ¡a, the component under analysis has control, and beginning at ¿a and !a the rest of the program has control. This has important implications for how these algorithms must be designed. A bidirectional grammar, which I present later in 7, decomposes into two unidirectional grammars, with opposite control behaviors.

In order to match a stream of symbols against a grammar, it is natural to use a parser. I have chosen to use a parsing algorithm based on the LL (1) algorithm [1]. There are a number of ways to parse an LL (1) grammar, but for the purposes of this discussion I distinguish two; recursive descent and table driven. Both of these approaches have similar efficiency, differing only by constant factors. A recursive descent parser is generally considered easier to read for humans and therefore is preferable for hand coded parsers (which is not the case for this application). An advantage of using a recursive descent parser for interface grammars is the fact that the compiler can insert the semantic predicates and actions directly into the methods of the recursive descent parser, whereas for a table driven parser, I must find a way to represent semantic actions or semantic predicates as data. The most important difference for my purposes, however, is where the tokens come from.

I distinguish two styles of parsing: *parser-calls* where the parser controls when the next token is produced, that is, the parser has a way of demanding that its environment produce a token for it when it chooses; and *code-calls* where the code invoking the parser controls when the next token is produced. The control problem requires the code-calls convention, because the parser will only just regain control when the $?p$

or ¡p symbols have been seen. It is very difficult to write a single threaded recursive descent parser using the code-calls convention in this environment, because the most natural implementation of a recursive descent parser stores its internal state on the same control stack that the user code will be using. I could use threads to resolve this problem (in effect by creating a new control stack for the parser); however, this would require synchronization between the user code and the parser threads, and the additional concurrency could degrade the performance of the model checker I eventually intend to run this automatically generated code. If I was targetting a language with coroutines [40] such as Icon [23] or Modula-2 [61], then I could use those; but most languages, and Java in particular, lack those capabilities. Due to these concerns, these algorithms are developed from table-driven LL (1) parsers.

The main problem is the question 'before any symbols have been matched, who has the thread of control'. If we are asking this about any given trace *that has already been recorded* the answer is simple; if the first symbol is $!a$ then the component under analysis has control, and if the first symbol is $?a$ then the rest of the program has control. If we have an empty trace—that is, there was no communication between the component under analysis and the rest of the program—then we have insufficient data. Similarly if we want to know which portion has control in the middle of the derivation, we may not have enough information. It is not sufficient to examine the grammar; the grammar may have two productions such as $P \rightarrow ?a \ldots \mid !a \ldots$. This becomes critical when we consider the "main" procedure of the composed system. We must have additional information, so that we can know which element should have control first. In the presented algorithms, I presume that a global variable *control* is **component** if the component has control initially and **program** if the rest of the program has control initially. In the latter case, the MAIN procedure is assumed to be the initial procedure.

5.1 Main algorithms

Armed with the previous resolutions to the control problem and the main problem, I present my algorithm in two stages. Algorithm 2 on page 55 ignores variables, and algorithm 4 on page 61 extends that to give a full implementation of the semantics using the scoping algorithms presented in algorithm 3 on page 58. In both cases, the initial setup is performed by algorithm 1 on the following page. These algorithms are defined in the context of a nondeterministic execution model. Specifically **choose**(s) indicates a nondeterministic choice of one of the values of s, **succeed** indicates a successful execution, and **fail** indicates a failure of one branch of execution. I note the symbols matched in equation (4.7) on page 39 and equation (4.8) on page 39 with a call to the MATCH procedure; this makes flow of control clearer. I also note the invocation of semantic actions $\langle\langle f \rangle\rangle$ and semantic predicates $[\![p]\!]$ by INVOKE(f) and INVOKE(p) respectively, to make it more apparent when they are executed.

Algorithm 1 Initial execution

```
Globals $\mathit{table} : \mathbf{NT} \times \mathbf{\Sigma} \to \mathcal{P}(\mathbf{Prod} \times \mathbf{Pred})$
    $\mathit{stack} \in \mathbf{Sym}^*$, $\mathit{control} \in \{\mathbf{component}, \mathbf{program}\}$
    $G = \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle$
    $\mathit{binds} \in (\mathbf{Var} \to \mathbb{N})^*$
    $\mathit{store} \in \mathbb{N} \to \mathbf{Dom}$
    $i \in \mathbb{N}$, $\mathit{la} \in \mathbf{\Sigma} \cup \{\bot\}$

 1: procedure MAIN'
 2:     $\mathit{binds} \leftarrow \emptyset \| \epsilon$
 3:     $\mathit{store} \leftarrow \emptyset$
 4:     $i \leftarrow 0$
 5:     $\mathit{la} \leftarrow \bot$
 6:     $\mathit{stack} \leftarrow S \| \square$
 7:     COMPUTETABLE$(G, \mathit{table})$
 8:     if $\mathit{control} = \mathbf{component}$ then
 9:         AWAIT$(\square)$
10:     else
11:         MAIN
12:         WITNESS$(\square)$
13:     succeed
```

For all these algorithms, I make some simplifying assumptions. As in the semantics in chapter 4 on page 29, I assume all methods and nonterminals have a single argument and a single return value; it is straightforward to accommodate more but complicates the presentation. For the purpose of this presentation, I assume the rest of the program has been rewritten so that any function call into the component is of the form $x \leftarrow p(y)$, and is rewritten as *arg* $\leftarrow y$; WITNESS$(?p)$; $x \leftarrow$ *result*. My implementation overrides the p method in the component, which is semantically the same, does not require modification of the rest of the program, but complicates the presentation unnecessarily.

I discuss these parsing algorithms in the order of presentation. First is algorithm 1 on the preceding page, which provides initial setup. To set the parser up, first we initialize the scoping information, which is important for the variables in algorithm 4 on page 61. I discuss the details there when I discuss algorithm 3 on page 58. Next we must initialize the stack in the usual manner for table driven parsers—that is, the stack is set to be two elements deep, with the start symbol on top and the end-of-parsing symbol on the bottom. This is done in line 6.

Next we must set up the parsing table accordingly. This work can of course be precomputed, and is in my implementation, but for ease of presentation I present it without said optimization. This is line 7. The details of COMPUTETABLE are presented as algorithm 7 on page 68.

Now that the parser initialization is complete, we may begin parsing. If the control is given first to the component (expressed as $\textit{control} = \textbf{component}$) then we cannot know the first symbol of the input; all we know is that when end-of-parsing is seen, we should stop. Accordingly, we call AWAIT, which waits for a symbol without expressing any lookahead constraints. This corresponds to line 9. The details of AWAIT are presented in algorithm 2 on page 55 and algorithm 4 on page 61.

On the other hand, if the control is first given to the program, then we assume an entry point MAIN. We immediately call that; when it completes there will be no more terminals at all, and so we call WITNESS, which is like AWAIT but insists that the very next symbol parsed be its argument. In this case, WITNESS($\Box$) means "the program must end now". This scenario corresponds to lines 11 to 12. The details of WITNESS are presented in algorithm 2 on the following page and algorithm 4 on page 61.

Finally, we have completed processing. If we have reached line 13, then we have completed an execution of the entire system without erring, and we may signal success in our nondeterministic execution model. If some error did occur, execution of that branch of the nondeterministic model would be terminated. This concludes the initial execution of the program, but I have left underdefined several important procedures, which I present below.

Now we may begin to explain how to parse the grammars. We begin by ignoring any variables, to make the explanation easier to follow, then give the full algorithm. The resulting simplified algorithm is presented in algorithm 2 on the following page.

First, we define WITNESS. A call to WITNESS(t) means that "I know that the very next symbol in the token stream is t; parse until you see it". This differs from AWAIT, which is "I know that eventually you will see t in the token stream; parse until you see it" only in that WITNESS sets the lookahead symbol *la*. If the lookahead symbol is not clear then either we have witnessed a second call before the first lookahead symbol had been consumed (which is illegal) or the parser has erred; in either case, we signal an error and halt processing. Otherwise, the lookahead symbol is set to t and we call AWAIT.

Algorithm 2 Parsing without variables

```
Globals $la \in \mathbf{\Sigma} \cup \{\bot\}, table : \mathbf{NT} \times \mathbf{\Sigma} \to \mathcal{P}(\mathbf{Prod} \times \mathbf{Pred}), stack \in \mathbf{Sym}^*$

 1: procedure WITNESS($t$)
 2:     if $la \neq \bot$ then
 3:         error
 4:     $la \leftarrow t$
 5:     AWAIT($t$)

 6: procedure AWAIT($t$)
 7:     repeat
 8:         if $stack = \epsilon$ then
 9:             fail
10:         $o \| stack \leftarrow stack$
11:         if $o \in \mathbf{\Sigma} \wedge la \neq \bot$ then
12:             if $o \neq la$ then
13:                 fail
14:             $la \leftarrow \bot$
15:         if $o = \square$ then
16:         else if $o = ?p$ then
17:             MATCH($?p$)
18:             AWAIT($¿p$)
19:         else if $o = ¿p$ then
20:             MATCH($¿p$)
21:         else if $o = !p$ then
22:             MATCH($!p$)
23:             $p()$
24:             WITNESS($¡p$)
25:         else if $o = ¡p(v)$ then
26:             MATCH($¡p$)
27:         else if $o = [\![p]\!]$ then
28:             if $\neg$INVOKE($p$) then
29:                 fail
30:         else if $o = \langle\!\langle a \rangle\!\rangle$ then
31:             INVOKE($a$)
32:         else if $o \in \mathbf{NT}$ then
33:             if $la = \bot$ then
34:                 $possible \leftarrow$
                        $\bigcup table[\{o\} \times \mathbf{\Sigma}]$
35:             else
36:                 $possible \leftarrow table(\langle o, la \rangle)$
37:             $viable \leftarrow \{P : (P, [\![p]\!]) \in$
                    $possible \wedge$ INVOKE($p$)$\}$
38:             $S, s_0, \ldots, s_n \leftarrow$ choose($viable$)
39:             $stack \leftarrow s_0 \| \ldots \| s_n \| stack$
40:         else
41:             error
42:     until $o = t$
```

AWAIT is structured as a repeat-until loop; we will continue parsing until the top of the stack is equal to t. To do so, we must retrieve the top of the stack. If the stack is ϵ, then it is empty while we are expecting additional tokens; this indicates the parse has somehow gone wrong. Accordingly we abandon this branch of execution. This corresponds to lines 8 to 9.

If the stack is not equal to ϵ, then we may pop the top of it; this is line 10. If the old top of the stack, which we call o now, is a terminal and we have a lookahead symbol, then it must be the case that $o = la$; otherwise, our lookahead symbol would be wrong. If the lookahead symbol was set and o is a terminal, we should clear it now, as we have matched the lookahead. This all is lines 11 to 14.

Now we must examine what o is and process it accordingly. It may be $\square$, the end of input token; in this case we need do nothing special. Either $o = t$, in which case we will end the loop, or $o \neq t$ in which case the stack will be empty (as $\square$ is placed at the bottom of the stack in algorithm 1 on page 52 and is never placed on the stack anywhere else) and it will also fail. This is line 15.

Alternatively, o may be a ? terminal. In this case we match it, match that terminal, and recursively continue, waiting for its pair ¿p. This is the only place where AWAIT(¿p) happens. This is lines 16 to 18.

Instead o may be a ¿ terminal; in this case we merely match it and match it, in lines 19 to 20.

o may be a ! terminal. In this case we match it and match it, then call the method specified. When the method returns, we know that the very next symbol is ¡p (because we just saw the return), and so we call WITNESS with it. This is lines 21 to 24.

o may be a ¡ terminal. We do no special handling here, merely match and match it, in lines 25 to 26.

We have now exhausted the terminals. o may be a semantic predicate, in which case we should execute it and, if it returns false, signal an error. This takes place in lines 27 to 29. Alternatively o may be a semantic action, in which case we should just execute it and move on; this is in lines 30 to 31.

o may be a nonterminal. In this case we must select a production from the parsing table. If we have a lookahead, we select the possible productions for the state o starting with the lookahead symbol. If we do not have a lookahead, then we must gather all possible productions for the state o for any lookahead symbol. This computation happens between lines 33 and 36. Now that we have the possible productions, we must weed out any non-viable productions; that is, we must eliminate any productions that are paired with a predicate that is false. This strictly speaking is an optimization and is not necessary for correctness, but it frequently trims productions that are guaranteed to fail, and so I include it. This occurs on line 37. Finally we must choose one production nondeterministically of all the viable productions, and prepend it to the stack. This occurs between lines 38 and 39.

Finally if o is of any other type, then something has gone fundamentally wrong with the parsing and we should halt execution and signal an error.

To discuss the additional complications involved in introducing variables to the parsing, I need to first present the symbol table I will be using to keep track of them and their scopes. I do so in algorithm 3 on the following page. I first define a global variable *binds*, which is a sequence of mappings from variable names to their location

Algorithm 3 Scoping

```
Globals $binds \in (\mathbf{Var} \to \mathbb{N})^*, store \in \mathbb{N} \to \mathbf{Dom}, i \in \mathbb{N}$

 1: procedure OPENSCOPE
 2:     $binds \leftarrow \emptyset \| binds$
 3: procedure CLOSESCOPE
 4:     $ignored \| binds \leftarrow binds$

 5: procedure BIND($v, x$)
 6:     if $binds = \epsilon$ then
 7:         error
 8:     else
 9:         $\rho \| s \leftarrow binds$
10:         $binds \leftarrow \rho[v \mapsto i] \| s$
11:         $store \leftarrow store[i \mapsto x]$
12:         $i \leftarrow i + 1$

13: procedure SET($v, x$)
14:     $store \leftarrow$ SET'$(v, x, binds)$
15: function SET'($v, x, s$)
16:     if $s = \epsilon$ then
17:         error
18:     else
19:         $\rho \| s' \leftarrow s$
20:         if $v \in \operatorname{dom} \rho$ then
21:             return $store[\rho(v) \mapsto x]$
22:         else
23:             return SET'$(v, x, s')$
24: function VALUE($v$)
25:     return VALUE'$(v, binds)$
26: function VALUE'($v$, $s$)
27:     if $s = \epsilon$ then
28:         error
29:     else
30:         $\rho \| s' \leftarrow s$
31:         if $v \in \operatorname{dom} \rho$ then
32:             return $store \circ \rho(v)$
33:         else
34:             return VALUE'$(v, s')$

35: function GETSTATE
36:     return GETSTATE'$(binds, \epsilon)$
37: function GETSTATE'($s, \sigma$)
38:     if $s = \epsilon$ then
39:         return $\sigma$
40:     else
41:         $\rho \| s' \leftarrow s$
42:         for all $v \in \operatorname{dom} \rho$ do
43:             if $v \notin \operatorname{dom} \sigma$ then
44:                 $\sigma \leftarrow \sigma[v \mapsto (store \circ \rho)(v)]$
45:         return GETSTATE'$(s', \sigma)$

46: procedure UPDATESTATE($\sigma$)
47:     for all $v \in \operatorname{dom} \sigma$ do
48:         SET$(v, \sigma(v))$
```

indexes; here, I am using the natural numbers for the set $\mathbf{Loc}$. When we introduce a new scope, as with the OPENSCOPE procedure, we prepend an empty map to the sequence. Similarly when we discard a scope, as in CLOSESCOPE, we pop the top of the sequence off and throw it away. Next, I define a map *store* that corresponds to $\mathbf{Store}$ in the formalism previously. Finally, I define a counter i that I use to ensure that I will give unique location indexes. These are all initialized in algorithm 1 on page 52 to their default values: *binds* $= \emptyset \| \epsilon$ is the sequence consisting of a single element, the empty map; *store* $= \emptyset$ is initialized to the empty map; and $i = 0$ is the first index element.

BIND is the first interesting procedure. My intention is that BIND associate the variable v with the value x in the mapping at the top of *binds*. If *binds* is empty—perhaps because no initial scope has been introduced—then that triggers an error; otherwise we take the top of *binds* and force it to associate v with x. Note that this, as required by the semantics, can discard older binds.

SET uses an auxiliary procedure SET'. The goal is that the topmost mapping that associates v with anything at all should instead associate it with x; accordingly we recurse over *binds*. If we reach the end of the mapping sequence, then we could not find any mapping that associated v with anything at all, and halt execution; this corresponds to lines 16 to 17. Otherwise we examine the top of the sequence ρ. If ρ maps v to something, then we change it to map v to x instead and return in line 21. Otherwise we leave ρ alone and recurse down the sequence in line 23.

We must be able to retrieve the current value of a variable v. This is in VALUE, which also uses an auxiliary procedure VALUE' and is structurally very similar to SET. The goal is that we should retrieve the value of v in the topmost mapping that associates v with anything at all. As in SET, we recurse over *binds*. If we reach the end, there is no value for v at all, and we halt execution; otherwise we examine the topmost mapping ρ.

If ρ maps v to something, then we return that something; otherwise we recurse down the sequence.

Finally, we need analogues for the $\varsigma \circ \rho$ used in equation (4.2) and equation (4.3) on page 39 and the $(\rho^{-1} \circ \sigma) \odot \varsigma$ used in equation (4.2). The former constructs a state mapping from the scope and state mappings, and the latter updates the state mapping from the changes made in σ. My analogue are GETSTATE and UPDATESTATE respectively.

GETSTATE calls GETSTATE' with the null state ϵ, which recurses down *binds*. For each entry ρ in *binds*, every variable v defined in ρ that is not already in σ is added to σ with the value $\textit{store}(\rho(v))$. The end result is that for every variable v that has been bound, $(\text{GETSTATE})(v) \equiv \text{VALUE}(v)$.

UPDATESTATE is simpler; for every entry in σ, we need to update its location in *binds*, which we use SET for. If a value is unchanged, then no harm is done. If σ contains values that are not bound, then SET will fail with an error. Since σ was initialized in GETSTATE with all variables that had been bound and not shadowed by an existing binding, assuming *binds* has not changed between the call to GETSTATE and UPDATESTATE it should be impossible for UPDATESTATE to change the scoping. This upholds the quality that in equation (4.2) on page 39, *binds* is unchanged while *store* is updated.

Algorithm 4 Full parsing

```
Globals  $arg \in \mathbf{Dom}$, $result \in \mathbf{Dom}$, $la \in \mathbf{\Sigma} \cup \{\bot\}$
    $table : \mathbf{NT} \times \mathbf{\Sigma} \rightarrow \mathcal{P}(\mathbf{Prod} \times \mathbf{Pred})$, $stack \in \mathbf{Sym}^*$

 1: procedure WITNESS($t$)
 2:     if $la \neq \bot$ then
 3:         error
 4:     $la \leftarrow t$
 5:     AWAIT($t$)

 6: procedure AWAIT($t$)
 7:     repeat
 8:         if $stack = \epsilon$ then
 9:             fail
10:         $o \| stack \leftarrow stack$
11:         if $o \in \mathbf{\Sigma} \wedge la \neq \bot$ then
12:             if $o \neq la$ then
13:                 fail
14:             $la \leftarrow \bot$
15:         if $o = \square$ then
16:         else if $o = ?p(v)$ then
17:             OPENSCOPE()
18:             MATCH($?p, arg$)
19:             BIND($v, arg$)
20:             AWAIT($¿p$)
21:             CLOSESCOPE()
22:         else if $o = ¿p(v)$ then
23:             $result \leftarrow$ VALUE($v$)
24:             MATCH($¿p, result$)
25:         else if $o = !p(v)$ then
26:             OPENSCOPE
27:             $r \leftarrow p($VALUE($v$)$)$
28:             MATCH($!p, r$)
29:             BIND(“$r”, $r$)
30:             WITNESS($¡p$)
31:         else if $o = ¡p(v)$ then
32:             $r \leftarrow$ VALUE(“$r”)
33:             CLOSESCOPE
34:             MATCH($¡p, r$)
35:             BIND($v, r$)
36:         else if $o = \Delta x$ then
37:             BIND($x, \bot$)
38:         else if $o = \uparrow$ then
39:             OPENSCOPE
40:         else if $o = \downarrow$ then
41:             CLOSESCOPE
42:         else if $o = [\![p]\!]$ then
43:             $\sigma \leftarrow$ GETSTATE
44:             if $\neg$INVOKEPREDICATE($p, \sigma$) then
45:                 fail
46:         else if $o = \langle\!\langle a \rangle\!\rangle$ then
47:             $\sigma \leftarrow$ GETSTATE
48:             $\sigma' \leftarrow$ INVOKE($a, \sigma$)
49:             UPDATESTATE($\sigma'$)
50:         else if $o = S[v]$ then    ▷ where $S \in \mathbf{NT}$
51:             if $la = \bot$ then
52:                 $possible \leftarrow \bigcup table[\{o\} \times \mathbf{\Sigma}]$
53:             else
54:                 $possible \leftarrow table(\langle o, la \rangle)$
55:             $viable \leftarrow \{P : (P, [\![p]\!]) \in possible$
                    $\wedge$ INVOKEPREDICATE($p, \sigma$)$\}$
56:             $S[x], s_0, \ldots, s_n \leftarrow$ choose($viable$)
57:             BIND(“$a”, VALUE($v$))
58:             OPENSCOPE
59:             BIND($x$, VALUE(“$a”))
60:             OPENSCOPE
61:             $stack \leftarrow s_0 \| \ldots \| s_n \| \downarrow \|$
                    $\langle\!\langle$SET(“$a”, VALUE($x$))$\rangle\!\rangle \| \downarrow \|$
                    $\langle\!\langle$SET($v$, VALUE(“$a”))$\rangle\!\rangle \|$
                    $stack$
62:         else
63:             error
64:     until $o = t$
```

Finally we may discuss the full parsing algorithm. Much of this is similar to algorithm 2 on page 55, and I discuss only the places where they differ here. First, however, we must change INVOKE to accommodate the constructions in equation (4.2) on page 39 and equation (4.3) on page 39. From now on, INVOKE(a, σ) invokes the semantic action a with the state σ returning the new state σ', and INVOKEPREDICATE(p, σ) invokes the semantic predicate p with the state σ and returns true or false depending on the value of the predicate.

The first point of difference is if o is a ? terminal. Here v is p's argument. v is merely the name; the actual value is stored in the global variable *arg*. Accordingly we match the pair ?p with the observed value, bind v to the observed value, and await its pair ¿p. The OPENSCOPE and CLOSESCOPE are there to protect the scoping information, as required by the semantics I specified.

If o is a ¿ terminal, then we must retrieve the value of the variable v. We store this in the global variable *result*, and match the pair ¿p with the observed return value.

If o is a ! terminal, then we must again retrieve the value of the variable v. The presence of the scoping information is less obvious. After calling the function p in line 27, we have its return value r. We will eventually see a token of the form ¡$p(v)$; in that case we should call BIND(v, r). But we need to store the return value somewhere where it will not be accidentally overwritten by a recursive call. We store it in the special variable name "\$r", and we open a scope first to ensure that we will not accidentally crush some previous binding of “\$r”. Finally we call WITNESS(¡p) to note the method return. The scope we opened will be closed in handling the ¡p; we cannot close it here because otherwise we will destroy the return value.

The other half of this is if o is a ¡ terminal. In this case we retrieve the return value r from the special variable “\$r”, match the pair ¡p, r, close the scope the ! opened and only then bind the variable v to r. Binding v any earlier would result in its binding being destroyed by the CLOSESCOPE.

There are some special symbols that come up with scoping that are presented in the formalism but were irrelevant to algorithm 2 on page 55. These include Δx, meaning “bind x to $\perp$” used for introducing new variables into a scope, $\uparrow$ to open a new scope and $\downarrow$ to close an existing scope. Handling of all these is simple.

Semantic predicates and semantic actions are unchanged here, but the values of the variables currently in scope need to be retrieved from *binds*. This is handled using the GETSTORE and UPDATESTORE methods described previously.

Finally, the case that o is a nonterminal must be handled. This is the same as before, with the exception that we must have in-out semantics for S’s argument v. To do so, we must know what name S assigns to its argument, which we get on line 56. We bind the value of the argument v to a temporary special variable, \$a to protect ourselves from accidental variable collisions. We now open a scope to bind the value of the variable v (stored as the value of \$a) to this variable x. We must open a scope here to avoid overwriting a preexisting binding for x. We then open a second scope to ensure that the execution of $s_0, \ldots, s_n$ will not overwrite the new binding for x. We then prepend the right hand side of the production to the stack, but we also append a close scope (to close our innermost open scope), a semantic action to retrieve the current value of x and set \$a to it, a second close scope (to close our outermost open scope), and finally set v to \$a, which is just the value of x at the end of the production.

5.2 Constructing the parse table

These algorithms use an auxiliary data structure *table*. This data structure is a modified LL (1) table. To construct this data structure, we use a modified LL (1) parse table construction. I must modify the standard LL (1) parse table construction algorithm due to two reasons:

- My language has semantic predicates that can influence the parse, by disallowing certain productions;
- The parser may or may not be able to determine the lookahead symbol at run time;
- I want to support nondeterministic choice in interface specifications which will be resolved by the target model checker's search heuristics at run time.

Accordingly, given a grammar $G = \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle$, I will construct a parsing table *table* : $\mathbf{NT} \times \mathbf{\Sigma} \rightarrow \mathcal{P}(\mathbf{P} \times \mathbf{SP})$.

Some features of this table that are notable: first, in normal LL(1) parsing a nonterminal—the top of the parse stack—and a terminal—the lookahead symbol—define at most one legal production. If more than one production is legal, then this indicates that the grammar in question is not LL(1). I wish to support nondeterministic choice and semantic predicates; the simplest way to do so is to relax this restriction; that is, to permit multiple productions to be available for any given nonterminal/terminal pair.

Second, because I wish to support semantic predicates, it can be the case that a production is not available at run time due to the semantic predicate being false. I could support this by simply using the **fail** construction defined earlier, but this would necessitate investigating multiple doomed partial parses. As an optimization, whenever this is sensible my compiler propagates semantic predicates forward so that the parser can choose to avoid clearly doomed paths. To support this, I pair the production available with the semantic predicate controlling whether it is *viable*.

Finally I may not be able to determine the symbol of lookahead. In this case, the parser takes the nonterminal at the top of the stack and computes the union of any available production pairing that nonterminal with any terminal.

Algorithm 5 Computing *first* sets

```
 1: procedure FIRST($G$, out first)
 2:     $\langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle \leftarrow G$
 3:     $first \leftarrow \emptyset[\epsilon \mapsto \{\langle [\![\mathbf{true}]\!], \epsilon \rangle\}]$
 4:     for all $n \in \mathbf{NT}$ do
 5:         $first \leftarrow first[n \mapsto \emptyset]$
 6:     repeat
 7:         for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ do
 8:             if $P \neq X \rightarrow \epsilon$ then
 9:                 $s \leftarrow first(Y_1 Y_2 \ldots Y_n)$
10:                 $p \leftarrow [\![\mathbf{true}]\!]$
11:                 for all $Y_i$ do
12:                     $d \leftarrow$ false
13:                     if $Y_i \in \mathbf{SP}$ then
14:                         $p \leftarrow p \wedge Y_i$
15:                     else if $Y_i \in \mathbf{SA}$ then
16:                         do nothing
17:                     else if $Y_i \in \mathbf{\Sigma}_\circ$ then
18:                         $target \leftarrow first(Y_i)$
19:                         if $\exists x : \langle x, \epsilon \rangle \in target$ then
20:                             $putative \leftarrow \{\langle q, t \rangle : \langle q, t \rangle \in target \wedge t \neq \epsilon\}$
21:                             if $s \neq putative \wedge s \neq target$ then
22:                                 $first \leftarrow first[Y_1 Y_2 \ldots Y_n \mapsto putative]$
23:                         else
24:                             if $s \neq target$ then
25:                                 $first \leftarrow first[Y_1 Y_2 \ldots Y_n \mapsto target]$
26:                             $d \leftarrow$ true
27:                     else
28:                         if $Y_i \notin s$ then
29:                             $first \leftarrow first[Y_1 Y_2 \ldots Y_n \mapsto first(Y_1 Y_2 \ldots Y_n) \cup \{\langle p, Y_i \rangle\}]$
30:                         $d \leftarrow$ true
31:                 if $\neg d$ then
32:                     $first \leftarrow first[Y_1 Y_2 \ldots Y_n \mapsto first(Y_1 Y_2 \ldots Y_n) \cup \{\langle p, \epsilon \rangle\}]$
33:         for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ do
34:             $first \leftarrow first[P \mapsto first(P) \cup first(Y_1 Y_2 \ldots Y_n)]$
35:     until no element in first has changed
```

Now that I have motivated the design of this particular parse table, I need two auxiliary functions *first* and *follow* a la Aho et al. [1], which I compute using the algorithms shown in algorithm 5 on the preceding page and algorithm 6.

Algorithm 6 Computing *follow* sets

```
1:  procedure FOLLOW($G$, $first$, out $follow$)
2:      $\langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle \leftarrow G$
3:      $follow \leftarrow \emptyset[\epsilon \mapsto \{\langle [\![\mathbf{true}]\!], \square \rangle\}]$
4:      for all $n \in \mathbf{NT}$ do
5:          $follow \leftarrow follow[n \mapsto \emptyset]$
6:      repeat
7:          for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ do
8:              for $i \leftarrow 1 \ldots n$ do
9:                  if $Y_i \in \mathbf{NT}$ then
10:                     if $i = n$ then
11:                         $s \leftarrow \epsilon$
12:                         $f \leftarrow \{\langle [\![\mathbf{true}]\!], \epsilon \rangle\}$
13:                     else
14:                         $s \leftarrow Y_{i+1} \ldots Y_n$
15:                         $f \leftarrow first(s)$
16:                         if $\exists p : \langle p, \epsilon \rangle \in f$ then
17:                             $f \leftarrow f \cup follow(X)$
18:                     $follow \leftarrow follow[Y_i \mapsto follow(Y_i) \cup f]$
19:     until no element in $follow$ has changed
```

Finally, given these algorithms we may compute the parsing table as given in algorithm 7 on the following page.

Algorithm 7 Computing *table*

```
1:  procedure ComputeTable($G$, out table)
2:      $\langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle \leftarrow G$
3:      First($G$, first)
4:      Follow($G$, first, follow)
5:      table $\leftarrow \emptyset$
6:      for all $n \in \mathbf{NT}$ do
7:          for all $t \in \mathbf{\Sigma}$ do
8:              table $\leftarrow$ table$[\langle n, t \rangle \mapsto \emptyset]$
9:      for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ do
10:         for all $\langle p, t \rangle \in$ first$(X)$ do
11:             if $t = \epsilon$ then
12:                 for all $\langle p', t' \rangle \in$ follow$(X)$ do
13:                     table $\leftarrow$ table$[\langle X, t' \rangle \mapsto$ table$(\langle X, t' \rangle) \cup \{\langle P, p \wedge p' \rangle\}]$
14:             else
15:                 table $\leftarrow$ table$[\langle X, t \rangle \mapsto$ table$(\langle X, t \rangle) \cup \{\langle P, p \rangle\}]$
```

Symbols	Set
Start	$\{\langle [\![\textbf{true}]\!], \epsilon \rangle, \langle [\![\textbf{true}]\!], ?\textit{setRollbackOnly} \rangle, \langle [\![\textbf{true}]\!], ?\textit{begin} \rangle\}$
Base	$\{\langle [\![\textbf{true}]\!], \epsilon \rangle, \langle [\![\textbf{true}]\!], ?\textit{setRollbackOnly} \rangle, \langle [\![\textbf{true}]\!], ?\textit{begin} \rangle\}$
Tail	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!], ?\textit{commit} \rangle, \langle [\![\textbf{true}]\!], ?\textit{rollback} \rangle\}$

Table 5.1: Relevant portions of the *first* set for grammar G

Symbols	Set
ϵ	$\{\langle [\![\textbf{true}]\!], \Box \rangle\}$
Start	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!], ?\textit{commit} \rangle, \langle [\![\textbf{true}]\!], ?\textit{rollback} \rangle, \langle [\![\textbf{true}]\!], \epsilon \rangle,$ $\langle [\![\textbf{true}]\!], ?\textit{setRollbackOnly} \rangle, \langle [\![\textbf{true}]\!], ?\textit{begin} \rangle\}$
Base	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!], ?\textit{commit} \rangle, \langle [\![\textbf{true}]\!], ?\textit{rollback} \rangle, \langle [\![\textbf{true}]\!], \epsilon \rangle,$ $\langle [\![\textbf{true}]\!], ?\textit{setRollbackOnly} \rangle, \langle [\![\textbf{true}]\!], ?\textit{begin} \rangle\}$
Tail	$\emptyset$

Table 5.2: Relevant portions of the *follow* set for grammar G

5.3 An example

An example is in order at this time. Consider grammar 3 on page 32, the example from section 4.2 on page 43, slightly modified to avoid using global variables. Here, the resulting grammar is as follows:

$$\begin{aligned}
\mathit{Start} &\rightarrow \Delta r\ \Delta l\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0]\rangle\!\rangle\ \mathit{Base}[r,l] \\
\mathit{Base}[r,l] &\rightarrow ?\texttt{begin()}\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1]\rangle\!\rangle\ ¿\texttt{begin()}\ \mathit{Base}[r,l]\ \mathit{Tail}[r] \\
&\quad \langle\!\langle \lambda\sigma.\textbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l)-1]\,\textbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)]\rangle\!\rangle \\
&\quad \mathit{Base}[r,l] \\
&\mid\ ?\texttt{setRollbackOnly()}\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}]\rangle\!\rangle \\
&\quad ¿\texttt{setRollbackOnly()}\ \mathit{Base}[r,l] \\
&\mid\ \epsilon \\
\mathit{Tail}[r] &\rightarrow [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\ ?\texttt{commit()}\ ¿\texttt{commit()} \\
&\mid\ ?\texttt{rollback()}\ ¿\texttt{rollback()}
\end{aligned}$$

The grammar is encoded into the formalism as follows:

$$\mathbf{P} = \{\langle \mathit{Start}, \qquad \Delta r\ \Delta l\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0]\rangle\!\rangle\ \mathit{Base}[r,l]\rangle,$$

$\mathbf{NT}$	$\mathbf{\Sigma}$	Set
Start	*?begin*	$\{\langle [\![\mathbf{true}]\!], \mathit{Start} \rightarrow \Delta r, l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{false}, l \mapsto 0] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Start	*?setRollbackOnly*	$\{\langle [\![\mathbf{true}]\!], \mathit{Start} \rightarrow \Delta r, l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{false}, l \mapsto 0] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Start	*?commit*	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \mathbf{false}]\!], \mathit{Start} \rightarrow \Delta r, l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{false}, l \mapsto 0] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Start	*?rollback*	$\{\langle [\![\mathbf{true}]\!], \mathit{Start} \rightarrow \Delta r, l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{false}, l \mapsto 0] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Start	$\Box$	$\{\langle [\![\mathbf{true}]\!], \mathit{Start} \rightarrow \Delta r, l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{false}, l \mapsto 0] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Base	*?begin*	$\{\langle [\![\mathbf{true}]\!], \mathit{Base}[r, l] \rightarrow \mathit{?begin}() \ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle \ \mathit{¿begin}() \ \mathit{Base}[r, l] \ \mathit{Tail}[r] \ \langle\!\langle \lambda\sigma.\mathbf{let}\, \sigma' = \sigma[l \mapsto \sigma(l) - 1] \ \mathbf{in}\, \sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle \ \mathit{Base}[r, l] \rangle\}$
Base	*?setRollbackOnly*	$\{\langle [\![\mathbf{true}]\!], \mathit{Base}[r, l] \rightarrow \mathit{?setRollbackOnly}() \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \mathbf{true}] \rangle\!\rangle \ \mathit{¿setRollbackOnly}() \ \mathit{Base}[r, l] \rangle\}$
Base	*?commit*	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \mathbf{false}]\!], \mathit{Base}[r, l] \rightarrow \epsilon \rangle\}$
Base	*?rollback*	$\{\langle [\![\mathbf{true}]\!], \mathit{Base}[r, l] \rightarrow \epsilon\}$
Base	$\Box$	$\{\langle [\![\mathbf{true}]\!], \mathit{Base}[r, l] \rightarrow \epsilon \rangle\}$
Tail	*?commit*	$\{\langle [\![\lambda\sigma.\sigma(r) \equiv \mathbf{false}]\!], \mathit{Tail}[r] \rightarrow [\![\lambda\sigma.\sigma(r) \equiv \mathbf{false}]\!] \ \mathit{?commit}() \ \mathit{¿commit}() \rangle\}$
Tail	*?rollback*	$\{\langle [\![\mathbf{true}]\!], \mathit{Tail}[r] \rightarrow \mathit{?rollback}() \ \mathit{¿rollback}() \rangle\}$
All other $\mathbf{NT} \times \mathbf{\Sigma}$ combinations map to $\emptyset$		

Table 5.3: Parse table for grammar G

$$\langle \mathit{Base}[r,l], ?\texttt{begin()}\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle\ ¿\texttt{begin()}\ \mathit{Base}[r,l]\ \mathit{Tail}[r]$$

$$\langle\!\langle \lambda\sigma.\textbf{let}\ \sigma' = \sigma[l \mapsto \sigma(l)-1]\ \textbf{in}\ \sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \rangle\!\rangle$$

$$\mathit{Base}[r,l]\rangle,$$

$$\langle \mathit{Base}[r,l], ?\texttt{setRollbackOnly()}\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}] \rangle\!\rangle$$

$$¿\texttt{setRollbackOnly()}\ \mathit{Base}[r,l]\rangle,$$

$$\langle \mathit{Base}[r,l], \epsilon\rangle,$$

$$\langle \mathit{Tail}[r],\quad [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\ ?\texttt{commit()}\ ¿\texttt{commit()}\rangle,$$

$$\langle \mathit{Tail}[r],\quad ?\texttt{rollback()}\ ¿\texttt{rollback()}\rangle\}$$

For clarity, here, I am not replacing variable names with unique numbers. This should, of course, be done in actual execution, but pedagogically it is easier to follow if the variables have names, and in any case it is not that important for this grammar, as r and l were originally global variables. The *first* set for that grammar G is presented in table 5.1 on page 68, and the *follow* set is presented in table 5.2 on page 68. The resulting parse table is presented in table 5.3 on the previous page.

If we are to use that parse table to parse a string—say, t_1 from section 4.2 again, we would proceed as follows. First, the input string, again.

$$t_1 = \, ?\textit{begin}[]\, ¿\textit{begin}[]\, ?\textit{begin}[]\, ¿\textit{begin}[]$$

$$?\textit{commit}[]\, ¿\textit{commit}[]\, ?\textit{rollback}[]\, ¿\textit{rollback}[]$$

We begin in MAIN', initializing *binds*, *store*, i, and *stack*. This is a component called by the main program, so *control* $=$ **program** and we call MAIN. At some point, *begin* is called with no arguments and expecting no results. This is rewritten as *arg* $\rightarrow \emptyset;$ WITNESS$(?begin);$ *ignored* $\rightarrow$ *result* and so we call WITNESS$(?begin)$. *la* has been initialized previously to $\perp$, so it is okay to set *la* $= ?begin$. *stack* $=$ *Start*$\|\Box$, so it is not empty and we pop the first value (in this case, *Start*) off and assign it to o. $o \notin \mathbf{\Sigma}$, so we proceed to the main conditional block. We reach the nonterminal case. Since *la* $\neq \perp$, we may use it to guide the parse. Accordingly,

$$\textit{possible} = \{\langle [\![\textbf{true}]\!], \textit{Start} \rightarrow \Delta r\ \Delta l\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0]\rangle\!\rangle$$

$$\textit{Base}[r, l]\rangle\}$$

Since all productions in *possible* are viable (because the only production present has $[\![\textbf{true}]\!]$ as its condition),

$$\textit{viable} = \{\textit{Start} \rightarrow \Delta r\ \Delta l\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0]\rangle\!\rangle$$

$$\textit{Base}[r, l]\}$$

Since there is only one production in *viable*, it is chosen. A scope is opened, meaning *binds* $= \emptyset \| \epsilon$. In this case, *Start* has no arguments and takes no arguments, so we open a second scope (now *binds* $= \emptyset \| \emptyset \| \epsilon$) and we set

$$\begin{aligned} \mathit{stack} = & \Delta r \ \Delta l \ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle && (5.1) \\ & \mathit{Base}[r, l] \ \downarrow \ \langle\!\langle \textsc{Set}(\text{“\$a”}, \textsc{Value}(\mathit{ignored})) \rangle\!\rangle \ \downarrow && (5.2) \\ & \langle\!\langle \textsc{Set}(\mathit{ignored}, \textsc{Value}(\text{“\$a”})) \rangle\!\rangle \ \square && (5.3) \end{aligned}$$

Since $o \neq t$ (because *Start* $\neq ?begin$), we repeat the loop again. *stack* is still not empty, so we pop the first value off and assign it to o. In this case, that value is Δr. This means we call $\textsc{Bind}(r, \perp)$, meaning *binds* $= \{r \mapsto \perp\} \| \emptyset \| \epsilon$. The next symbol is Δl, so we do it again, meaning *binds* $= \{r \mapsto \perp, l \mapsto \perp\} \| \emptyset \| \epsilon$. The next symbol is a semantic action. We call GETSTATE to initialize $\sigma = \{r \mapsto \perp, l \mapsto \perp\}$. We call INVOKE to execute the semantic action; this means $\sigma' = \{r \mapsto \textbf{false}, l \mapsto 0\}$. Finally we update *binds* with UPDATESTATE, meaning *binds* $= \{r \mapsto \textbf{false}, l \mapsto 0\} \| \emptyset \| \epsilon$.

The next symbol is $\mathit{Base}[r, l]$, which is the first nonterminal we have seen with arguments. *la* $=$ *?begin* still, so only one viable production exists, namely

$$\begin{aligned} \mathit{Base}[r, l] \rightarrow & ?\mathit{begin}() \ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle \\ & ¿\mathit{begin}() \ \mathit{Base}[r, l] \ \mathit{Tail}[r] \end{aligned}$$

$$\begin{aligned}
&\langle\!\langle \lambda\sigma.\mathbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l) - 1]\\
&\qquad \mathbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)]\rangle\!\rangle\\
&\mathit{Base}[r, l]
\end{aligned}$$

Accordingly we open another scope. Since we are coding two arguments as an ordered pair, $r = \langle \mathbf{false}, 0\rangle$. After the bind, *binds* $= \{r \mapsto \mathbf{false}, l \mapsto 0\} \| \{r \mapsto \mathbf{false}, l \mapsto 0\} \| \emptyset \| \epsilon$. Finally we open one more scope and set *stack* to

$$\begin{aligned}
\mathit{stack} = &?\mathit{begin}()\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1]\rangle\!\rangle\\
&¿\mathit{begin}()\ \mathit{Base}[r, l]\ \mathit{Tail}[r]\\
&\langle\!\langle \lambda\sigma.\mathbf{let}\,\sigma' = \sigma[l \mapsto \sigma(l) - 1]\\
&\qquad \mathbf{in}\,\sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)]\rangle\!\rangle\\
&\downarrow\ \langle\!\langle \textsc{Set}(\text{“\$a”}, \langle \textsc{Value}(r), \textsc{Value}(l)\rangle)\rangle\!\rangle\\
&\downarrow\ \langle\!\langle \textsc{Set}(r, \mathrm{first}(\textsc{Value}(\text{“\$a”})));\\
&\qquad \textsc{Set}(l, \mathrm{second}(\textsc{Value}(\text{“\$a”})))\rangle\!\rangle\\
&\downarrow\ \langle\!\langle \textsc{Set}(\text{“\$a”}, \textsc{Value}(\mathit{ignored}))\rangle\!\rangle\\
&\downarrow\ \langle\!\langle \textsc{Set}(\mathit{ignored}, \textsc{Value}(\text{“\$a”}))\rangle\!\rangle\ \square
\end{aligned}$$

Finally, we now match the $?begin$. In so doing, we match and consume our lookahead, we open another scope, call $\textsc{Match}(?begin, [])$, bind the argument (to nothing, since it is ignored), and recursively call $\textsc{Await}(¿receive)$. This recursive call invokes $\langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l) + 1] \rangle\!\rangle$ before matching $¿begin$ and matching $\textsc{Match}(¿receive, [])$. We then exit the recursive call, and since $o = t$ now, finally exit from the original $\textsc{Witness}$, returning control to the main program.

The remainder of the parsing proceeds normally. When the main program finally ends, we call $\textsc{Witness}(\square)$, consuming the end-of-input token and completing the parse successfully.

5.4 Correctness

It is important to guarantee that algorithm 1 on page 52 together with algorithm 4 on page 61 upholds the semantics described in figure 4.1 on page 39. I provide a proof sketch of this here.

The first task is to verify that the tables in section 5.2 on page 64 are constructed correctly; that is, if $\langle n, t\rangle \mapsto S$, it should be the case that any production that can be taken from the nonterminal n given the lookahead symbol t should be somewhere in S. My algorithms are straightforward modifications to the normal LL (1) algorithms to track predicates and to retain any ambiguity. The more interesting question is do these algorithms uphold the semantics in figure 4.1 on page 39. I take the equations in figure 4.1 on page 39 in order, showing that the algorithm presented upholds these semantics.

Equation (4.1) specifies that if we reach the end of the input stream, we may accept. If *control* $=$ **component**, this corresponds to seeing the $\Box$ on the end of the stack as set in line 6 in algorithm 1 on page 52, and matching it using the AWAIT call on line 9 of that same algorithm. Algorithm 4 on page 61 performs no MATCH calls when matching $\Box$, so our final trace is identical to that of equation (4.1).

Equation (4.2) specifies that if f does not refer to variables that are not in scope, a is invoked with the current running state and the running store ς is updated with the changes a makes. This is handled in lines 46 to 49 of algorithm 4 on page 61. We use the GETSTATE and UPDATESTATE algorithms from algorithm 3 on page 58 here; GETSTATE retrieves the state, we invoke a with it, and then update the store with the new state. Assuming GETSTATE and UPDATESTATE are correct, then this section is also correct.

Equation (4.3) specifies that if the predicate p succeeds when applied to the current scope, then processing may continue. Since there are no other derivation rules that refer to predicates, this effectively means that if p fails then processing should stop. This is handled in lines 42 to 45 in algorithm 4 on page 61. The current state is retrieved with GETSTATE and then p is invoked with it. If the predicate fails, we halt processing.

Equation (4.4) handles introducing new variable declarations. This is handled in lines 36 and 37. I bind the variable x to $\bot$. The text of BIND in algorithm 3 on page 58 ensures that a new location not previously used is assigned, so upholding the semantic requirement that $\xi \notin \mathrm{dom}(\varsigma)$.

Equation (4.5) handles nonterminals. The specific production is chosen between lines 50 and 56; these remove some possibilities from contention, but those possibilities could never have been feasible. A new scope is opened on line 58, ensuring that the old store ρ is just a CLOSESCOPE() away. x is bound to a fresh location in line 59, fulfilling the requirement that $\xi \notin \mathrm{dom}(\varsigma)$. A further scope is opened, which ensures that ξ can still be retrieved later. The contents of the nonterminal are pushed to the stack along with two $\downarrow$s that will ensure that the CLOSESCOPES() occur after the nonterminal is finished executing; additionally v is set to the value of the parameter x after execution, upholding the requirement that $\rho(y) \mapsto \varsigma' \circ \rho'(x)$ in equation (4.5). So this derivation, too, is handled.

Equation (4.6) handles block scoping; as with the nonterminal described above, we handle this with OPENSCOPE and CLOSESCOPE. This protects the old ρ without having any implications for the store.

Equation (4.7) handles ? events. This is handled in a different way in algorithm 4, although equivalent. Specifically because we cannot know, upon seeing a $?p(v)$ what the name of its partner is. Accordingly we match the sequence $?p[\mathit{arg}]$, bind v to that argument, and recursively parse until we see its partner. Once the partner is seen, the global variable *result* is bound to the return value and the corresponding sequence ¿$p[\textit{result}]$ is matched. The scope opening and closing is to protect the old ρ as required by the semantics.

Equation (4.8) handles ! events. Again, since we cannot know the event's pair when we parse it we handle this slightly differently. We open a scope to protect the "$r" variable, match the observed sequence !$p[\textsc{Value}(v)]$, and call the function and then recursively parse, looking for the matching ¡p. That retrieves the "$r", closes the scope that the ! opened, matches the sequence ¡$p[r]$, and then binds v to r.

Equation (4.9) and Equation (4.10) are syntactical requirements for derivation and have no execution considerations.

Thus, my algorithm upholds the semantics I specified.

Chapter 6

Implementing Interface Grammars in Java

Chapter 5 on page 48 described several algorithms for parsing interface grammars, but they do not directly translate to Java. There are four main complications:

- Java is not inherently nondeterministic.
- I must encode semantic actions and semantic predicates as data in a Java program in such a way that they can be executed as code later.
- Java libraries routinely use exceptions, and so our parser must be robust in the face of them.
- I want to be able to use interface grammars without invasive changes to existing Java code.

I deal with each of these in turn.

6.1 Nondeterminism in Java

Java the language does not contain any immediate support for nondeterminism. I could possibly accommodate it using threading, but this would necessitate making our parser thread safe and complicate our eventual goal of model checking the program. I could accommodate nondeterminism through some sort of code transformation—for example, in the way Screamer [47] extends Common Lisp to support nondeterminism. This is not in any way easy to do even for Common Lisp, and would probably necessitate some sort of transformation of the targeted Java code to Continuation Passing Style [15]. In any case it is hard to see a way to implement nondeterminism directly in Java that would be noninvasive, an important goal for this research.

Instead, I have chosen to use the nondeterministic framework that must already be built into the model checker I eventually intend to use. This does not require invasive code changes and leverages a stable body of code. Almost all Java model checkers provide some way to interface with the model checker's nondeterminism; since I used JPF in this work I describe that model checker's interface, but my compiler could be easily ported to any other model checker that provides a similar interface.

For our purposes we need to model three things:

- **choose**(S) is a function that nondeterministically returns some $s \in S$.
- **succeed** signals that the current branch of execution has completed successfully.

- **fail** signals that the current branch of execution has failed in some fashion, and that a different branch should be attempted.

To achieve these, I use two functions from JPF's interface, which is encoded in the `gov.nasa.jpf.jvm.Verify` interface. To choose an element from a set S, I call `Verify.random` to nondeterministically select an integer from 0 to $|S| - 1$; I then retrieve the item $s \in S$ indexed by that integer. For **fail**, I use `Verify.ignoreIf`. Unfortunately JPF does not provide a method useful for **succeed**, or rather it conflates **fail** and **succeed**. As a result, what can happen is that all branches of execution fail but JPF reports no error, misleading the user. Accordingly for a **succeed** call we print "Success" before calling `Verify.ignoreIf`; if no "Success" appears in the output, then no branch of execution completed successfully.

6.2 Encoding code as data

The next hurdle I must overcome is the process of encoding Java code as a data object that can be stored in data structures and the like. In another language I might use closures or function pointers; in Java I use anonymous inner classes. I use the JGA [24] library here as a basis for these inner classes, which JGA refers to as "generators".

Since I will shortly be discussing Java code templates with appropriately substituted values, I will refer to the substitution of some computed value x within a template as

$\$(x)$ or $\$x$ when unambiguous. Sometimes it will be clearer to not use a template, in which case I use the OUTPUT function.

Now, to encode some Java code, I write simply

```
new Generator () {
    public Object gen () {
        $code
    }
}
```

Predicates are encoded in a similar fashion.

However, this introduces a new problem; now, every Java escape is in a lexically distinct context from every other Java escape. While writing interfaces, it is very useful to retain some information across Java escapes, and additionally arguments in method calls can define new variables that the stub must make decisions on. If I was using a recursive descent parser, I could exploit the Java compiler's scoping, but I have dismissed that possibility above; accordingly I must track it myself with my own symbol table.

My symbol table algorithms are encoded in algorithm 3 on page 58, and the algorithms in that chapter already track these variables. What I must do here is show how to exploit that information to make my generators work properly. If in this symbol table I used the variable names as keys, I would get dynamic scoping; referring to the variable e would retrieve the most recently bound e, and not necessarily the e that is lexically appropriate. I desire lexical scoping. Accordingly, I assign to each variable declaration in the program a unique number, and use that as a key. This gives me lexical scoping provided it is not possible to get what are essentially function pointers to grammar fragments; in that case I would be faced with the so-called “funarg” problem and would have to adopt a different approach. I avoid this problem through the simple expedient of disallowing pointers to grammar fragments.

I must also keep track of the declarations that are visible both before and after every Java escape. Given this, I can now alter the body of the Java closure code to be as follows:

```
for all $decl \in visibleSymbolsBefore$ do
    OUTPUT(“$\$(decl.type)$ $\$(decl.name) \leftarrow$ VALUE($\$(decl.id)$)”)
OUTPUT(“$\$code$”)
for all $decl \in visibleSymbolsAfter$ do
    OUTPUT(“SET($\$(decl.id), \$(decl.name)$)”)
```

This takes care of the GETSTATE, INVOKE, UPDATESTATE sequence in lines 46 to 49 of algorithm 4 on page 61. I must also include opening scopes, closing scopes, and binding in my semantics interface grammar; fortunately I have already done so.

6.3 Handling exceptions

Java code frequently throws exceptions when the code performs an illegal operation. Throwing exceptions in an uncontrolled manner can cause the parse information to be

Algorithm 8 Error recovery

```
Globals $stack \in \mathbf{Sym}^*$
  procedure TOSSUNTIL($t$)
      repeat
          if $stack = \epsilon$ then
              fail
          $o \| stack \leftarrow stack$
      until $o = t$
```

destroyed; for example it can cause my grammar parser to not consume the ¿ tokens properly. In the case of an exception in a semantic action or semantic predicate, this is an error in the grammar itself and I can specify any operation that is not needlessly destructive. I have chosen to, upon an error, remove elements from the stack until the expected matching token is seen, and then rethrow the exception. This is similar to the error handling a normal LL(1) parser will perform upon illegal input, and is implemented in algorithm 8.

But not all exceptions are errors: a faithful representation of the interface may require that exceptions be thrown, and then the grammar implementation must throw them in a manner consistent with the parsing algorithm. In this case, I store the exception in a member variable and rethrow it as soon as it is safe, thus preserving the parse information.

An example is worthwhile. Consider the following pseudo-grammar:

Grammar 4. Throwing an exception (pseudo-grammar)

$$\mathit{Start} \quad \rightarrow \quad \text{?begin()} \; \langle\!\langle \ldots \rangle\!\rangle \; \textbf{throw new} \; \text{Exception()} \; \text{¿begin}$$

How should we write a real grammar to encode the **throw** operation? It is nonsensical to specify computation between the **throw** and the ¿. We can encode it as follows:

Grammar 5. Throwing an exception

$$\mathit{Start} \quad \rightarrow \quad \text{?begin()} \; \langle\!\langle \ldots \rangle\!\rangle \; \langle\!\langle \lambda\sigma.\mathit{exception} \leftarrow \textbf{new} \; \text{Exception()} \rangle\!\rangle \; \text{¿begin}$$

When the ¿begin is consumed, we check the contents of *exception* and throw it if *exception* was non null.

6.4 Stubbing methods

The remaining hurdle is that the implementation in chapter 5 on page 48 specifies that the client code must be changed. This is undesirable as usually the components we would like to stub out using interface grammars have much more client code than stub code. Fortunately, this is unnecessary. If our interface grammar implements a Java interface, then for each method specified by the interface our stub may specify an implementation of that method that calls WITNESS() as desired. For each such method, I output the following implementation:

```
public $returnType(stub) $name(stub)
        ($argument(stub)) {
    argument = $argument(stub);
    result = null; exception = null;
    WITNESS(?$name(stub));
    try {
        WITNESS(¿$name(stub));
    } catch (Exception e) {
        TOSSUNTIL(¿$name(stub));
        exception = e;
    }
    if (exception != null)
        throw exception;
    return ($returnType(stub)) result;
}
```

Again, I assume one argument and one return value. The `exception` variable here, and the try/catch block, are for exception handling as discussed in section 6.3 on page 83. The `result` value has a similar use: it is assigned to in algorithm 4 on page 61, and then we simply cast the result to the correct return type (since the Java compiler will refuse to compile otherwise) and return the value.

Chapter 7

Specification of Semantic Predicates and Actions Using JML

With the interface grammars as defined previously in chapter 4 on page 29, many things become convenient to specify, and with the Java escapes in the semantic actions and semantic predicates everything that the environment does can be specified. But excessive use of the semantic actions and semantic predicates can defeat the entire goal of interface grammars, which is to be more flexible and more convenient than writing raw Java code. I present here a layer on top of the grammars previously presented that makes the specification of certain bidirectional grammars much easier. To motivate this, I consider the problem of object generation and matching.

A weakness of the interface specification language defined in some of my previous work [28] is that it does not provide direct support for describing the data associated with the method calls and returns of a component, i.e., the arguments and return values for the component methods. However, the interface specification language presented in that work allows specification of semantic predicates and actions. This enables the users to insert arbitrary Java code to interface specifications. These semantic predicates and actions can be treated as nonterminals with epsilon-productions and the Java code in them are executed when the corresponding nonterminal appears at the top of the parser stack. The user can do object validation and generation using such semantic predicates and semantic actions. However, this approach is unsatisfactory for the same reason that hand writing a component stub in Java directly is unsatisfactory; it is frequently brittle and difficult to understand. Accordingly, I extended my interface grammars to support generating and validating data, and did so in a way that preserves the advantages of grammars.

7.1 Shape types

The shape types of Fradet and le Métayer [19] are an attractive formalism based on graph grammars that can be used to express recursive data structures. I have been inspired by their formalism, but to accommodate the differences between their goal and mine, my implementation becomes substantially different. Nonetheless, it is worthwhile explaining Fradet and le Métayer's shape types and then explaining how my approach differs syntactically before explaining my implementation.

Shape types are an extension to a traditional type system. Their goal is to extend an underlying type system so that it can specify the shape of a data structure; for example, a doubly linked list. This extension is done through extending a normal context free grammar, which we will proceed to explain.

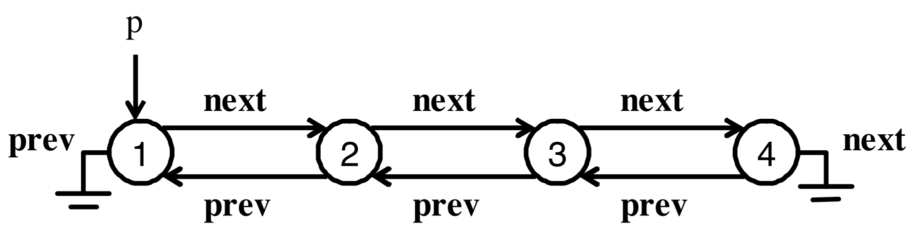
Consider the language of strings $(\mathbf{name}\ x\,y)^*$, where $\mathbf{name}$ is some string and x and y are integers. If we regard x and y as vertices, then we can obtain a labeled directed graph from any such string by regarding the string $\mathbf{name}\ x\,y$ as defining an edge from the vertex labeled x to the vertex labeled y, itself labeled $\mathbf{name}$. If we further regard the vertices in this graph as representing objects and the edges as representing fields, we can obtain an object graph. Note that this mapping is not one-to-one: if the strings are reordered the same graph is obtained.

We can represent external pointers into this object graph by adding strings of the form $\mathbf{p}\ x$; here, the pointer named $\mathbf{p}$ points to the object x.

I now want a grammar that can output these graph encodings. While we can regard $\mathbf{name}$ as a terminal, the vertices are not so simple. We extend the context-free grammar to permit parameters; so the production $N\ \ x\,y\ \rightarrow\ \mathbf{next}\ \ x\,y$ describes the string $\mathbf{next}\ \ x\,y$, whatever its parameters x and y are. If a variable is referred to in the right hand side of a production but not listed in the parameters, then it represents a new object that has not yet been observed. Fradet and le Métayer use $\mathbf{next}\ \ x\,x$ to represent terminal links; we prefer to use $\mathbf{next}\ \ x\ \mathrm{null}$ for the same purpose.

$$\begin{array}{rcl} \textit{Doubly} & \rightarrow & \mathbf{p}\ x, \mathbf{prev}\ x\ \text{null}, L\ x \\ L\ x & \rightarrow & \mathbf{next}\ x\, y, \mathbf{prev}\ y\, x, L\ y \\ L\ x & | & \mathbf{next}\ x\ \text{null} \end{array}$$

(a) Shape type



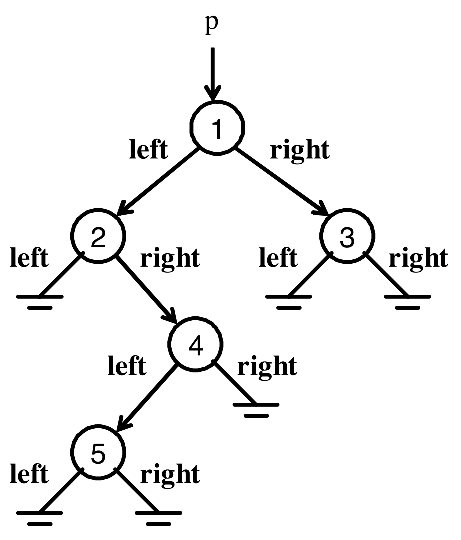
(b) An example of that type

Figure 7.1: The shape type for a doubly linked list, with an example

Shape types provide a powerful formalism for specification of object graphs. In figure 7.1 I show the shape type for a doubly linked list and an example of that type. In figure 7.2 on the next page I show the shape type for a binary tree and an example of that type. It is also possible to specify data structures such as left-child, right-sibling trees, red-black trees, and skip lists using shape types [19].

$$\begin{array}{rcl} \textit{Bintree} & \rightarrow & \mathbf{p}\ x, B\ x \\ B\ x & \rightarrow & \mathbf{left}\ x\,y, \mathbf{right}\ x\,z, B\ y, B\ z \\ B\ x & \mid & \mathbf{left}\ x\ \text{null}, \mathbf{right}\ x\ \text{null} \end{array}$$

(a) Shape type



(b) An example of that type

Figure 7.2: The shape type for a binary tree, with an example

7.2 Object generation with interface grammars

In this section I discuss how I integrated shape types into my interface grammar specification language. First, I start with a brief discussion on alternative ways of generating arbitrary object graphs in a running Java program. Next, I give an overview of my extended interface grammar language and discuss how this extended language supports shape types. I conclude this section by presenting an example interface grammar for a left-child, right-sibling (LCRS) tree cursor.

7.2.1 Creating object graphs

There are three major techniques for object graph creation: with JVM support, serialization, and method construction. The first technique uses support from the JVM to create objects arbitrarily and in any form desired. Visser et al. [57] use this technique, extending the Java PathFinder model checker appropriately. While this is very powerful, it is necessarily coupled to a specific JVM and can be easy to inadvertently create object structures that cannot be recreated by a normal Java program. I have rejected this approach because we do not want to be overly coupled to a specific JVM.

The second technique uses the Java serialization technologies used by Remote Method Invocation (RMI) [59]. This is almost as powerful as the first technique and has the advantage of being more portable. Since the serialization format is standardized, it is relatively easy to create normal serialization streams by fiat. There are two major issues with this approach. First, it requires that all the objects that one might want to generate be serializable, which requires changing the source code in many cases. Second, it is possible for an object to arbitrarily redefine its serialization format or to add arbitrarily large amounts of extra data to the object stream. This is common in the Java system libraries. Accordingly I have rejected this approach as well.

The third approach, and the one I settled upon, is to generate object graphs through the object's normal methods. The main advantages this approach has is that it works with any object, it is as portable as the original program, and it is impossible to get an object graph that the program could not itself generate. The main disadvantage is that this is not amenable to fully automated analysis; if the objects are all well behaved and follow the Java Beans specification it is just possible to automatically generate a tree, but automatically determining the appropriate degree of sharing in the object graph is immensely difficult. Since I do semiautomated analysis, I use this technique with the shape types of the previous section and ask the user to tell the compiler what sort of shape they desire.

7.2.2 Support for shape types

I can obtain all the power required to embed the shape types of section 7.1 on page 88 into my interface grammars with one feature: rules have parameters. The semantics of this situation has already been defined in chapter 4 on page 29, but I examine the reasoning behind it here. Because I need to be able to pass objects to the rules as well as retrieve them, I have chosen to use call-by-value-return semantics for our parameters—rather like the "in out" parameters of the Ada language. Because we

Shape type

$$\begin{array}{rcl}
\mathit{Tree} & \rightarrow & \mathbf{p}\ x, \mathbf{parent}\ x\ \text{null}, \mathbf{left}\ x\ \text{null}, \mathbf{right}\ x\ \text{null}, T\ x \\
T\ x & \rightarrow & \mathbf{child}\ x\ \text{null} \\
T\ x & \rightarrow & \mathbf{child}\ x\, y, \mathbf{parent}\ y\, x, \mathbf{left}\ y\ \text{null}, R\ \langle y, x\rangle, T\ y \\
R\ \langle x, p\rangle & \rightarrow & \mathbf{right}\ x\ \text{null} \\
R\ \langle x, p\rangle & \rightarrow & \mathbf{right}\ x\, y, \mathbf{left}\ y\, x, \mathbf{parent}\ y\, p, R\ \langle y, p\rangle, T\ y
\end{array}$$

Tree walker

$$\begin{array}{rcl}
\mathit{tree}[n] & \rightarrow & \langle\!\langle \texttt{if(n==null)n=newNode();} \rangle\!\rangle\ \mathit{tree}'[n] \\
\mathit{tree}'[n] & \rightarrow & ?\texttt{moveDown}()\ \text{¿}\texttt{moveDown}() \\
 & & \Delta\mathit{child}\ \langle\!\langle \texttt{child=n.getFirstChild();} \rangle\!\rangle \\
 & & \mathit{tree}[\mathit{child}]\ \langle\!\langle \texttt{n.setFirstChild(child);} \rangle\!\rangle \\
 & & \langle\!\langle \texttt{child.setParent(n);} \rangle\!\rangle\ \mathit{maybeUp}[\mathit{child}] \\
 & | & ?\texttt{moveRight}()\ \text{¿}\texttt{moveRight}() \\
 & & \Delta\mathit{sib}\ \langle\!\langle \texttt{sib=n.getRightSibling();} \rangle\!\rangle \\
 & & \mathit{tree}[\mathit{sib}]\ \langle\!\langle \texttt{n.setRightSibling(sib);} \rangle\!\rangle \\
 & & \langle\!\langle \texttt{sib.setLeftSibling(n);} \rangle\!\rangle\ \mathit{maybeLeft}[\mathit{sib}] \\
 & | & ?getTree()\ \text{¿}getTree(n)\ \mathit{tree}[n] \\
 & | & \epsilon \\
\mathit{maybeUp}[n] & \rightarrow & ?\texttt{moveUp}()\ \text{¿}\texttt{moveUp}() \\
 & & \Delta\mathit{parent}\ \langle\!\langle \texttt{parent=n.getParent();} \rangle\!\rangle \\
 & & \mathit{tree}[\mathit{parent}] \\
 & | & \epsilon \\
\mathit{maybeLeft}[n] & \rightarrow & ?\texttt{moveLeft}()\ \text{¿}\texttt{moveLeft}() \\
 & & \Delta\mathit{sib}\ \langle\!\langle \texttt{sib=n.getLeftSibling();} \rangle\!\rangle \\
 & & \mathit{tree}[\mathit{sib}] \\
 & | & \epsilon
\end{array}$$

Figure 7.3: Tree cursor with data

have chosen uniform call-by-value-return semantics, only the name of a variable is a permissible argument. This is seen in equation (4.5) on page 39.

The algorithms require a certain amount of care in these conditions—when we see a nonterminal on the stack, we must store the parameter in a special location to prevent it from being overwritten. This is all handled between lines 50 and 61 of algorithm 4 on page 61.

7.2.3 A tree cursor

As a demonstration of this new ability to encode data more directly using my interface grammar specification language, I present a tree cursor for a left-child, right-sibling tree. It contains movement methods `moveDown`, `moveRight`, `moveUp` and `moveLeft` and a `getTree` method, which will return a tree populated as far as the cursor has explored. The interface grammar is shown in figure 7.3 on the preceding page. We omit generation of unreached areas and error checking when moving right and up to make the example clearer—naturally an example suitable for verification would need to include them. Note that the interface grammar shown in figure 7.3 on the previous page makes sure that the the tree that is returned by the `getTree` method is always consistent with the previous calls to the `moveUp`, `moveDown`, `moveLeft` and `moveRight` methods that have been observed.

Shape type

$$\begin{array}{rcl} \textit{Doubly} & \rightarrow & \mathbf{p}\ x, \mathbf{prev}\ x\ \text{null}, L\ x \\ L\ x & \rightarrow & \mathbf{next}\ x\, y, \mathbf{prev}\ y\, x, L\ y \\ L\ x & \mid & \mathbf{next}\ x\ \text{null} \end{array}$$

Generation

$$\begin{array}{rcl} \textit{genDoubly}[x] & \rightarrow & \langle\!\langle \texttt{x = new Node ();} \rangle\!\rangle\ \langle\!\langle \texttt{x.setPrev (null);} \rangle\!\rangle\ \textit{genL}[x] \\ \textit{genL}[x] & \rightarrow & \Delta y\ \langle\!\langle \texttt{y = new Node ();} \rangle\!\rangle\ \langle\!\langle \texttt{x.setNext (y);} \rangle\!\rangle \\ & & \langle\!\langle \texttt{y.setPrev (x);} \rangle\!\rangle\ \textit{genL}[y] \\ & \mid & \langle\!\langle \texttt{x.setNext (null);} \rangle\!\rangle \end{array}$$

Matching

$$\begin{array}{rcl} \textit{matchDoubly}[x] & \rightarrow & \Delta \textit{ns}\ \langle\!\langle \texttt{ns = new HashSet ();} \rangle\!\rangle \\ & & [\![\texttt{x instanceof Node}]\!]\ [\![\texttt{!ns.contains (x)}]\!] \\ & & \langle\!\langle \texttt{ns.insert (x);} \rangle\!\rangle\ [\![\texttt{x.getPrev () == null}]\!] \\ & & \textit{matchL}[\langle x, \textit{ns} \rangle] \\ \textit{matchL}[\langle x, \textit{ns} \rangle] & \rightarrow & [\![\texttt{x.getNext () == null}]\!]\ \epsilon \\ & \mid & [\![\texttt{x.getNext () != null}]\!]\ \Delta y\ \langle\!\langle \texttt{y = x.getNext ();} \rangle\!\rangle \\ & & [\![\texttt{y instanceof Node}]\!]\ [\![\texttt{!ns.contains (y)}]\!] \\ & & \langle\!\langle \texttt{ns.insert (y);} \rangle\!\rangle \\ & & [\![\texttt{x.getNext () == y}]\!]\ [\![\texttt{y.getPrev () == x}]\!] \\ & & \textit{matchL}[\langle y, \textit{ns} \rangle] \end{array}$$

Figure 7.4: Interface grammars for doubly linked list generation and matching

Shape type

$$\begin{array}{rcl} \textit{Bintree} & \rightarrow & \mathbf{p}\ x, B\ x \\ B\ x & \rightarrow & \mathbf{left}\ x\ y, \mathbf{right}\ x\ z, B\ y, B\ z \\ B\ x & \mid & \mathbf{left}\ x\ \mathrm{null}, \mathbf{right}\ x\ \mathrm{null} \end{array}$$

Generation

$$\begin{array}{rcl} \textit{genBintree}[x] & \rightarrow & \langle\!\langle \texttt{x = new Node ();} \rangle\!\rangle\ \textit{genB}[x] \\ \textit{genB}[x] & \rightarrow & \Delta y\ \langle\!\langle \texttt{y = new Node ();} \rangle\!\rangle\ \Delta z\ \langle\!\langle \texttt{z = new Node ();} \rangle\!\rangle \\ & & \langle\!\langle \texttt{x.setLeft (y);} \rangle\!\rangle\ \langle\!\langle \texttt{x.setRight (z);} \rangle\!\rangle\ \textit{genB}[y]\ \textit{genB}[z] \\ & \mid & \langle\!\langle \texttt{x.setLeft (null);} \rangle\!\rangle\ \langle\!\langle \texttt{x.setRight (null);} \rangle\!\rangle \end{array}$$

Matching

$$\begin{array}{rcl} \textit{matchBintree}[x] & \rightarrow & \Delta \textit{ns}\ \langle\!\langle \texttt{ns = new HashSet ();} \rangle\!\rangle \\ & & [\![\texttt{x instanceof Node}]\!]\ [\![\texttt{!ns.contains (x)}]\!] \\ & & \langle\!\langle \texttt{ns.insert (x);} \rangle\!\rangle\ [\![\texttt{x.getPrev () == null}]\!] \\ & & \textit{matchB}[\langle x, \textit{ns} \rangle] \\ \textit{matchB}[\langle x, \textit{ns} \rangle] & \rightarrow & [\![\texttt{x.getLeft () == null}]\!]\ [\![\texttt{x.getRight () == null}]\!] \\ & \mid & [\![\texttt{x.getLeft () != null}]\!]\ \Delta y\ \langle\!\langle \texttt{y = x.getLeft ();} \rangle\!\rangle \\ & & [\![\texttt{y instanceof Node}]\!]\ [\![\texttt{!ns.contains (y)}]\!] \\ & & \langle\!\langle \texttt{ns.insert (y);} \rangle\!\rangle\ \Delta z\ \langle\!\langle \texttt{z = x.getRight ();} \rangle\!\rangle \\ & & [\![\texttt{z instanceof Node}]\!]\ [\![\texttt{!ns.contains (z)}]\!] \\ & & \langle\!\langle \texttt{ns.insert (z);} \rangle\!\rangle \\ & & [\![\texttt{x.getLeft () == y}]\!] \\ & & [\![\texttt{x.getRight () == z}]\!]\ \textit{matchB}[\langle y, \textit{ns} \rangle] \\ & & \textit{matchB}[\langle z, \textit{ns} \rangle] \end{array}$$

Figure 7.5: Interface grammars for binary tree generation and matching

7.3 Object generation vs. object validation

Using my interface grammar specification language, it is possible to specify both generation and validation of data structures, and to do so in a manner that is reminiscent of the shape types of section 7.1 on page 88. Object validation is used to check that the arguments passed to a component by its clients satisfy the constraints specified by the component interface. Object generation, on the other hand, is used to create the objects that are returned by the component methods based on the constraints specified in the component interface.

Figure 7.4 on page 96 and figure 7.5 on the preceding page shows object generation and validation for doubly linked list and binary tree examples. The figure contains three specifications for each of the two examples. At the top of the figure we repeat the shape type specifications for doubly linked list and binary tree examples from section 7.1 on page 88 for convenience. The middle of the figure contains the interface grammar rules for generation of these data structures. Note the close similarity between the shape type productions and the productions in the interface grammar specification. The bottom of the figure shows the interface grammar rules for validation of these data structures.

Object generation and validation tasks are broadly symmetric, and their specification as interface grammar rules reflects this symmetry as seen in figure 7.4 on page 96 and figure 7.5 on the previous page. While in object generation semantic actions are used to set the fields of objects to appropriate values dictated by the shape type specification, in object validation, these constraints are checked using semantic predicates specified as guards. Note that the set of nonterminals and productions used for object generation and validation are the same.

The most significant difference between the object generation and validation tasks is the treatment of aliasing among different nodes in an object graph. The semantics of shape type formalism makes some implicit assumptions about aliasing between the nodes. Intuitively, shape type formalism assumes that there is no aliasing among the nodes of the object graph unless it is explicitly stated. During object generation it is easy to maintain this assumption. During generation, every `new` statement creates a new object what is not shared with any other object in the system. If the specified data structure requires aliasing, this can be achieved by passing nodes as arguments as is done in shape type formalism.

Detecting aliasing among objects is necessary during object validation. Note that, since shape type formalism assumes that no aliasing should occur unless it is explicitly specified, during object validation we need to make sure that there is no unspecified aliasing. Instead of trying to enforce a fixed policy on aliasing, we leave the specification of the aliasing policy during object validation to the user. The typical way to check aliasing would be by using a hash-set as demonstrated by the two object validation examples shown in figure 7.4 on page 96 and figure 7.5 on page 97. Note that, the interface grammar rules for object validation propagate the set of nodes that have been observed and make sure that there is no unspecified aliasing among them.

7.4 Using JML to specify object validation and generation

The generation and validation sections of figure 7.4 on page 96 and figure 7.5 on page 97 are similar, but this similarity is obscured by the mechanics of validation and generation. For example, the construct ⟪x = new Node ();⟫ corresponds to the constructs

⟦x instanceof Node⟧ ⟦!ns.contains (x)⟧ ⟪ns.insert (x);⟫

in that both constructs ensure that as execution proceeds, x is a fresh `Node` instance. Similarly, the correspondence between ⟪x.setLeft (null);⟫ and ⟦x.getLeft () == null⟧ should be apparent.

It would be nice to only have to write this once. It may not always be possible, but it would be useful if in the common case we could only have to write the validation procedure and infer the generation procedure from that, or vica versa. In fact a great deal of effort has gone into doing just this; Korat [13] requires a specially formatted validation procedure, from which it can infer a generation procedure, and Visser et. al [58] did similar work using JPF. One disadvantage of this approach is that these inferences tend to be brittle; Visser et. al mention that an incorrectly written validation procedure can cause an infinite loop during generation.

I have opted for a different approach. In my approach, the user specifies invariants of the system in the Java Modeling Language (JML) [12, 32, 34] and the interface grammar compiler will construct a correct validation procedure and a correct generation procedure from these invariants. This problem is enormously difficult and undecidable in its general form. Even the JML specification [35] does not guarantee that all JML expressions can be translated to Java; section 11.4.24.4 notes that it is possible to write quantified expressions that are not be translated. Accordingly at present my compiler can only translate a small portion of the JML specification. This portion is adequate for the examples presented here, however.

7.4.1 The Java Modeling Language

JML is a specification language for Java programs, using Hoare style preconditions, postconditions and invariants. It is usually embedded in Java code using annotation comments. The language supports many different sorts of modeling; for my purposes here I am principally interested in the expression language.

This expression language is a superset of the side-effect free portions of the Java expression syntax. It is extended with various Boolean operators like $==>$ (denoting logical implication), existential and universal quantifiers, and a host of extensions resembling functions (for example \fresh, which returns true if the variables given are bound to new objects that did not previously exist in the system, or \old, which refers to the value of an expression in some previous state). The user can even extend JML's primitives with user defined functions obeying certain restrictions.

As mentioned above, arbitrary JML expressions can be nontrivial to translate to Java, and additionally generating an object graph capable of satisfying an arbitrary expression would almost certainly require a theorem prover. Different JML tools support different subsets of the language. Accordingly at this time I consider only a minimal subset capable of expressing the properties I am interested in. If the user requires more elaborate operations, semantic predicates and actions written in Java remain available. I present this subset in figure 7.6 on the following page. As a stylistic convention, I use *id* to denote a variable name, b to denote a Boolean value, i to denote an integer, s to denote a string, and *Type* to denote a type name.

$$\begin{array}{rcll}
R & \rightarrow & R \mathbin{\&\&} R & (7.1)\\
& | & R \| R & (7.2)\\
& | & R \texttt{==>} R & (7.3)\\
& | & E == E & (7.4)\\
& | & E.\mathsf{equals}(E) & (7.5)\\
& | & I[\mathit{id}] & (7.6)\\
& | & b & (7.7)\\
& | & \mathit{id} & (7.8)\\
& & \quad \mathit{id}\text{ a Boolean variable} & \\
& | & E.\mathsf{isEmpty}() & (7.9)\\
& | & E.\mathsf{contains}(E) & (7.10)\\
& | & \backslash\mathsf{fresh}(E^*) & (7.11)\\
& | & \backslash\mathsf{old}(R) & (7.12)\\
& | & (\backslash\mathsf{forall}\ \mathit{Type}\ \mathit{id}; & (7.13)\\
& & \qquad S[\mathit{id}];\ R) & \\
& | & (\backslash\mathsf{forall}\ \mathsf{int}\ \mathit{id}; & (7.14)\\
& & \qquad I[\mathit{id}];\ R) & \\
E & \rightarrow & i & (7.15)\\
& | & s & (7.16)\\
& | & b & (7.17)\\
& | & \mathit{id} & (7.18)\\
& | & \mathsf{null} & (7.19)\\
& | & E[E] & (7.20)\\
& | & E.\mathit{id} & (7.21)\\
& & \quad \text{for a field } \mathit{id} & \\
& | & E.\mathsf{get}(E) & (7.22)\\
& | & E.\mathsf{get}\mathit{Field}() & (7.23)\\
& & \quad \text{for a field } \mathit{field} & \\
& | & \backslash\mathsf{old}(E) & (7.24)\\
S[\mathit{id}] & \rightarrow & E.\mathsf{contains}(\mathit{id}) & (7.25)\\
& | & E.\mathsf{containsKey}(\mathit{id}) & (7.26)\\
I[\mathit{id}] & \rightarrow & \mathit{id} > i & (7.27)\\
& | & \mathit{id} >= i & (7.28)\\
& | & \mathit{id} > E & (7.29)\\
& | & \mathit{id} >= E & (7.30)\\
& | & \mathit{id} < i & (7.31)\\
& | & \mathit{id} <= i & (7.32)\\
& | & \mathit{id} < E & (7.33)\\
& | & \mathit{id} <= E & (7.34)\\
& | & I[\mathit{id}] \mathbin{\&\&} I[\mathit{id}] & (7.35)\\
& | & I[\mathit{id}] \| I[\mathit{id}] & (7.36)
\end{array}$$

Figure 7.6: The subset of JML I accept

Table 7.1: Attributes used

Attribute	Meaning
$R.mg$	Macroexpansion of a relation for generation
$R.mv$	Macroexpansion of a relation for verification
$R.or$	\old variable substitution within a relation
$R.om$	\old variable-to-expression map within a relation
$R.gs$	Java statements to ensure that the relation R is fulfilled
$R.gr$	a Java expression to verify that the relation R is fulfilled
$E.r$	a Java expression to get E's value (E as "r-value")
$E.l$	a function taking one expression and returning a Java statement to set E's value to that argument (E as "l-value")
$S.i$	a Java expression required to generate an iterator for all values matched by S
$I.l$	the lowest possible value for I
$I.h$	the highest possible value for I
$I.r$	a Boolean expression encoding the constraint I

Algorithm 9 JML expression expansion

```
1: function CONVERT($P = X \to Y_1 Y_2 \ldots Y_n, d$)
2:     $ys' = [1 \le i \le n : \textsc{MacroExpand}(Y_i, d)]$
3:     $ys'' = [1 \le i \le n : \textsc{OldExpand}(ys')]$
4:     $oldmap = \bigcup_{i=1}^{n} \textsc{OldMap}(ys')$
5:     $oldinit = \Big\Vert_{x \in \mathrm{dom}(oldmap)} \langle\langle \text{“}x \leftarrow \text{”} \Vert oldmap(x) \Vert \text{“; ”} \rangle\rangle$
6:     return $X \to oldinit \Vert \Big\Vert_{i=1}^{n} \textsc{Generate}(ys''_i, d)$
```

7.4.2 Translating JML expressions to Java

There are three stages of translation for this to be rendered to interface grammars. The algorithm for translating JML expressions is shown in algorithm 9, using auxiliary functions defined in figure 7.7 on the following page and figure 7.8 on page 106. These functions use attribute grammars defined in figure 7.9 on page 107, figure 7.10 on page 110, figure 7.11 on page 113, figure 7.12 on page 114, figure 7.13 on page 115,

$$\begin{aligned}
\textsc{MacroExpand}(\mathsf{ensure}\ R, \textbf{normal}) &= R.\mathit{mg} && (7.37)\\
\textsc{MacroExpand}(\mathsf{ensure}\ R, \textbf{inverse}) &= R.\mathit{mv} && (7.38)\\
\textsc{MacroExpand}(\mathsf{require}\ R, \textbf{normal}) &= R.\mathit{mv} && (7.39)\\
\textsc{MacroExpand}(\mathsf{require}\ R, \textbf{inverse}) &= R.\mathit{mg} && (7.40)\\
\textsc{MacroExpand}(\mathsf{let}\ R, d) &= R.\mathit{mg} && (7.41)\\
\textsc{MacroExpand}(\mathsf{assert}\ R, d) &= R.\mathit{mv} && (7.42)\\
\textsc{MacroExpand}(\mathsf{when}\ R, d) &= R.\mathit{mv} && (7.43)
\end{aligned}$$

or for any other grammar construct P

$$\textsc{MacroExpand}(P, d) = P \qquad (7.44)$$

$$\begin{aligned}
\textsc{OldExpand}(\mathsf{ensure}\ R) &= R.\mathit{or} && (7.45)\\
\textsc{OldExpand}(\mathsf{require}\ R) &= R.\mathit{or} && (7.46)\\
\textsc{OldExpand}(\mathsf{let}\ R) &= R.\mathit{or} && (7.47)\\
\textsc{OldExpand}(\mathsf{assert}\ R) &= R.\mathit{or} && (7.48)\\
\textsc{OldExpand}(\mathsf{when}\ R) &= R.\mathit{or} && (7.49)
\end{aligned}$$

or for any other grammar construct P

$$\textsc{OldExpand}(P) = P \qquad (7.50)$$

Figure 7.7: Auxiliary functions for expansion

and figure 7.14 on page 117. I discuss each of these grammars below. These grammars use attributes defined in table 7.1 on the preceding page.

First, I extend the grammar syntax of section 4.1 on page 33 with five new statements. The simplest are when and assert; both convert to a predicate halting execution unless the provided JML predicate is true. Next simplest is let, which guarantees that the provided JML predicate is true. The final two, ensure and require, are for specifying

$$\begin{aligned}
\textsc{OldMap}(\textsf{ensure}\, R) &= R.om && (7.51)\\
\textsc{OldMap}(\textsf{require}\, R) &= R.om && (7.52)\\
\textsc{OldMap}(\textsf{let}\, R) &= R.om && (7.53)\\
\textsc{OldMap}(\textsf{assert}\, R) &= R.om && (7.54)\\
\textsc{OldMap}(\textsf{when}\, R) &= R.om && (7.55)
\end{aligned}$$

or for any other grammar construct P

$$\textsc{OldMap}(P) = P \qquad (7.56)$$

$$\begin{aligned}
\textsc{Generate}(\textsf{ensure}\, R, \mathbf{normal}) &= \langle\!\langle R.gs \rangle\!\rangle && (7.57)\\
\textsc{Generate}(\textsf{ensure}\, R, \mathbf{inverse}) &= [\![R.gr]\!] && (7.58)\\
\textsc{Generate}(\textsf{require}\, R, \mathbf{normal}) &= [\![R.gr]\!] && (7.59)\\
\textsc{Generate}(\textsf{require}\, R, \mathbf{inverse}) &= \langle\!\langle R.gs \rangle\!\rangle && (7.60)\\
\textsc{Generate}(\textsf{let}\, R, d) &= \langle\!\langle R.gs \rangle\!\rangle && (7.61)\\
\textsc{Generate}(\textsf{assert}\, R, d) &= [\![R.gr]\!] && (7.62)\\
\textsc{Generate}(\textsf{when}\, R, d) &= [\![R.gr]\!] && (7.63)
\end{aligned}$$

or for any other grammar construct P

$$\textsc{Generate}(P, d) = P \qquad (7.64)$$

Figure 7.8: Auxiliary functions for expansion, part 2

bidirectional grammars. In the so-called "normal" mode of execution, ensure acts as let and require acts as assert. In the so-called "inverse" mode of execution, the compiler is asked to express the counterpart to the specified grammar. That is, if the user writes a grammar for a component in a system, "normal" execution generates a stub for that component. "Inverse" execution generates a stub for the rest of the system the component communicates with.

$$
\begin{aligned}
R \to\ & E_0\mathsf{==}\backslash\mathsf{new}(\mathit{Type}; & R.\mathit{mg} =\ & E_0\mathsf{==new}\,\mathit{Type}(E_1,\ldots,E_n) \\
& \quad E_1,\ldots,E_n; R_1,\ldots,R_m) & & \quad \&\&\ \backslash\mathsf{fresh}(E_0) \\
& & R.\mathit{mv} =\ & E_0\ \mathsf{instanceof}\ \mathit{Type}\ \&\& \\
& & & \quad \backslash\mathsf{fresh}(E_0)\ \&\&\ R_1\ \&\&\ \ldots\ \&\&\ R_m
\end{aligned}
\tag{7.65}
$$

$$
\begin{aligned}
R \to\ & \backslash\mathsf{add}(E_1, E_2) & R.\mathit{mg} =\ & E_1.\mathsf{add}(E_2) \\
& & R.\mathit{mv} =\ & E_1.\mathsf{containsAll}(\backslash\mathsf{old}(\mathsf{new\ ArrayList}(E_1))) \\
& & & \quad \&\&\ E_1.\mathsf{contains}(E_2)
\end{aligned}
\tag{7.66}
$$

$$
\begin{aligned}
R \to\ & \backslash\mathsf{push}(E_1, E_2) & R.\mathit{mg} =\ & E_1.\mathsf{addLast}(E_2) \\
& & R.\mathit{mv} =\ & E_1.\mathit{size}() == \backslash\mathsf{old}(E_1.\mathit{size}()) + 1\ \&\& \\
& & & \quad \backslash\mathsf{oldcoll}(E_1, 0, \backslash\mathsf{old}(E_1.\mathit{size}()), 0) \\
& & & \quad \&\&\ E_1.\mathsf{get}(E_1.\mathit{size}() - 1) == E_2
\end{aligned}
\tag{7.67}
$$

$$
\begin{aligned}
R \to\ & E_0 == \backslash\mathsf{pop}(E_1) & R.\mathit{mg} =\ & E_0 == E_1.\mathsf{removeLast}() \\
& & R.\mathit{mv} =\ & E_1.\mathit{size}() == \backslash\mathsf{old}(E_1.\mathit{size}()) - 1\ \&\& \\
& & & \quad \backslash\mathsf{oldcoll}(E_1, 0, \backslash\mathsf{old}(E_1.\mathit{size}()) - 1, 0) \\
& & & \quad \&\&\ E_0 == \backslash\mathsf{old}(E_1.get(E_1.\mathit{size}() - 1))
\end{aligned}
\tag{7.68}
$$

Figure 7.9: Macro expansion prior to \old conversion. $R.\mathit{mg}$ represents the correct expansion if we are in a generation context, whereas $R.\mathit{mv}$ represents the correct expansion if we are in a verification context.

The function CONVERT translates a grammar using JML expressions into a grammar using only Java expressions. The first argument is the production to translate, and the second argument is the direction—that is, "normal" or "inverse".

The first step of conversion is expanding certain macros. I use these because it is difficult to concisely express concepts like "this array is unchanged, except that this object is appended to it" in raw JML. Since I hope to construct from the JML expression both validation and generation code, it is important that these be easily recognized. I define four macros, whose expansion is presented in figure 7.9 on the preceding page.

The first macro, $\backslash\mathsf{new}$, solves a technical problem in translating object creation. There are four important facts needed to both verify object creation and ensure that it happens; first, that the object is brand new and has never been seen before. Second, that the object is of a certain type. Third, the constructor arguments. Fourth, any resulting assignments to the object properties. The first two are not difficult to establish, but there is no way to automatically infer the constructor arguments from any object properties, nor is there any way to automatically infer the object properties from the constructor arguments. I could follow the JavaBeans specification [49] and use a no-argument constructor to create the object and then use setter methods; however, this is not useful when dealing with value objects like strings, which cannot be modified after construction. Accordingly I use a macro that expresses both the constructor arguments ($E_1, \ldots, E_n$ in equation (7.65) on the previous page) and any corresponding relations upheld by the constructor ($R_1, \ldots, R_m$ in equation (7.65)).

The second through fourth macros represent modification of collection objects. It is relatively easy to express change for mapping; we can just write $X.\mathsf{get}(Y) == Z$, and in one direction interpret this literally and in the other direction change it to $X.\mathsf{put}(Y, Z)$. But adding a collection to an array implies that everything that used to exist in the collection still does, as well as this new object. Similarly, pushing and popping from a stack imply that the order of the stack is not changed, merely the size (by one) and perhaps the last element. These are not trivial to translate, and so I provide macros. In each case, one direction is much simpler than the other, as I can rely upon the Java standard library to work properly, while it is inappropriate to assume that code under examination does the right thing. These expansions make use of a specialized construction I define below.

The second step in translation is dealing with the \old operator. That command refers to some previous binding of a variable, frequently used in Design By Contract (DBC) [39] style pre- and postconditions. For DBC, which is focused on objects and methods, the idea of where in the execution of the program the \old expression should be computed is defined to be the start of the annotated method. Since I use grammars, I must define precisely where this previous binding takes place, and how the \old substitution takes place.

While I could define a complex system of rules for determining when the contents of the \old commands are computed, due to the grammar abstraction I have a very natural position already; the start of the grammar production itself. I could declare that \old expressions are computed at procedure entry, but it is not always clear what the enclosing procedure is for any given JML expression, whereas the start of the grammar production is obvious. Since grammar productions are wholly under the control of the

$$E \to \backslash\mathsf{old}(E_1) \qquad E.or = E.i \tag{7.69}$$
$$E.om = E_1.om[E.i \mapsto E_1]$$

$$R \to \backslash\mathsf{old}(R_1) \qquad R.or = R.i \tag{7.70}$$
$$R.om = R_1.om[R.i \mapsto R_1]$$

$$R \to \backslash\mathsf{oldarr}(E_1, s_1, e_1, s_2) \qquad \begin{aligned} R.or = {} & \text{“}(\backslash\mathsf{forall\ int}\text{”}\|R.j\|\text{“};\text{”}\|R.j \\ & \|\text{“} >= \text{”}\|s_1\|\text{“}\ \&\&\ \text{”}\|R.j \\ & \|\text{“} < \text{”}\|e_1\|\text{“};\text{”}\|E_1\|\text{“[”}\| \\ & R.j\|\text{“]} == \text{”}\|R.i \\ & \|\text{“[”}\|R.j\|\text{“} - \text{”}\|s_1\|\text{“])”} \\ R.om = {} & (R_1.om \cup E_1.om) \\ & [R.i \mapsto \text{“}\mathsf{Arrays.copyOfRange}(\text{”}\| \\ & \quad E_1\|\text{“},\text{”}\|s_2\|\text{“},\text{”}\| \\ & \quad e_1 - s_1 + s_2\|\text{“})\text{”}] \end{aligned} \tag{7.71}$$

$$R \to \backslash\mathsf{oldcoll}(E_1, s_1, e_1, s_2) \qquad \begin{aligned} R.or = {} & \text{“}(\backslash\mathsf{forall\ int}\text{”}\|R.j\|\text{“};\text{”}\|R.j \\ & \|\text{“} >= \text{”}\|s_1\|\text{“}\ \&\&\ \text{”}\|R.j \\ & \|\text{“} < \text{”}\|e_1\|\text{“};\text{”}\|E_1\|\text{“}\mathsf{.get}(\text{”}\| \\ & R.j\|\text{“}) == \text{”}\|R.i\|\text{“}\mathsf{.get}(\text{”}\| \\ & R.j\|\text{“} - \text{”}\|s_1\|\text{“} + \text{”}\|s_2\|\text{“}))\text{”} \\ R.om = {} & (R_1.om \cup E_1.om) \\ & [R.i \mapsto \text{“}\mathsf{new\ ArrayList}(\text{”}\|E_1\|\text{“})\text{”}] \end{aligned} \tag{7.72}$$

otherwise for any other rule $NT \to S$ with nonterminals $NT_1, \ldots, NT_n$

$$NT \to S \qquad NT.om = \bigcup_{i=1}^{n} NT_i.om \tag{7.73}$$

Figure 7.10: Computation of \old expressions. Here *NT*$.i$ and *NT*$.j$ are unique identifier names, *NT*$.or$ is the expression after \old substitution, and *NT*$.om$ is synthesized attribute containing a mapping from identifiers to expressions. Because JML prohibits enclosing quantified variables in \old expressions, we need not specifically handle \forall expressions.

user, this should permit controlling where these expressions will be executed, in the event that this is important. Accordingly, during translation of the \old expressions I accumulate a mapping of variable names to expressions, and bind those variables to those names at the start of the production. I omit dealing with types here for simplicity; since Java is strongly typed, to generate correct Java code in line 5 of algorithm 9 on page 104 requires being able to determine which type $\mathit{oldmap}(x)$ evaluates to. Doing this is complex and tedious but well understood, and so I omit it to make the presentation more comprehensible.

Figure 7.10 on the previous page describes the computation of the \old expressions. It uses an attributed grammar, that rewrites the tree accordingly, and stores the expressions-to-be-calculated in $\mathit{NT}.\mathit{or}$ as a map. When an expression is replaced with a variable, a mapping from the variable to the expression is added to $\mathit{NT}.\mathit{or}$. If there are no \old-type expressions in a JML expression, we unify the $\mathit{NT}.\mathit{or}$ maps of all sub-expressions. If variable capture was possible, for example in anonymous functions or the like, then this technique would not work; in translation the value for the captured variable would somehow need to be precomputed. This is possible in JML thanks to quantified variables in \forall expressions and its cousins. I must support \forall; there is no sensible way to handle arrays, lists and the like without it. Fortunately, the JML specification itself anticipates this problem and prohibits referring to quantified variables within \old expressions. So for the most part, I can simply copy the expression and use it directly, as in equation (7.69) on page 110 and equation (7.70) on page 110.

I need one more \old-like command, however. Without quantified variables in \old expressions, it is difficult to establish conditions such as "the contents of this array have not changed". If the length of the array is bounded one could enumerate all possible indexes, but this is tedious and brittle. Accordingly I permit two commands \oldarr and \oldcoll, which establish that the "old" version of E_1 from s_1 to e_1 and the "new" version of s_2 of the same length contain identical objects. Ideally I would only have one command, but as Java distinguishes between primitive arrays and collection classes I must have both. These are translated as in equation (7.71) on page 110 and equation (7.72). In both cases, R is replaced by a \forall expression that verifies that each component of the array or collection is unchanged, and $R.r$ is updated with a binding for $R.i$ that will copy the array structure (a so-called "shallow copy"). $R.j$ is used as the name for the quantifier variable.

The final step in conversion is, with macros expanded and \old substitution complete, translating the JML expressions to Java code. This corresponds to line 6 of algorithm 9 on page 104. These translations are presented in figure 7.11 on the following page, figure 7.12 on page 114, figure 7.13 on page 115 and figure 7.14 on page 117.

$$
\begin{array}{lll}
E \rightarrow i & E.r = \text{“}i\text{”} & (7.74) \\
E \rightarrow s & E.r = \text{“}s\text{”} & (7.75) \\
E \rightarrow b & E.r = \text{“}b\text{”} & (7.76) \\
E \rightarrow \mathit{id} & E.r = \text{“}\mathit{id}\text{”} & (7.77) \\
 & E.l = \lambda x.E.r \| \text{“} = \text{”} \| x & \\
E \rightarrow \mathsf{null} & E.r = \text{“}\mathsf{null}\text{”} & (7.78) \\
E \rightarrow E_1[E_2] & E.r = E_1.r \| \text{“[”} \| E_2.r \| \text{“]”} & (7.79) \\
 & E.l = \lambda x.E.r \| \text{“} = \text{”} \| x & \\
E \rightarrow E_1.\mathit{id} & E.r = E_1.r \| \text{“.”} \| \mathit{id} & (7.80) \\
 & E.l = \lambda x.E.r \| \text{“} = \text{”} \| x & \\
E \rightarrow E_1.\mathsf{get}(E_2) & E.r = E_1.r \| \text{“.}\mathsf{get}(\text{”} \| E_2.r \| \text{“)”} & (7.81) \\
 & E.l = \lambda x.E_1.r \| \text{“.}\mathsf{put}(\text{”} \| E_2.r \| \text{“,”} \| x \| \text{“)”} & \\
E \rightarrow E_1.\mathsf{get}\mathit{Field}() & E.r = E_1.r \| \text{“.}\mathsf{get}\mathit{Field}()\text{”} & (7.82) \\
 & E.l = \lambda x.E_1.r \| \text{“.}\mathsf{set}\mathit{Field}(\text{”} \| x \| \text{“)”} &
\end{array}
$$

Figure 7.11: Generation rules for JML expressions. Here $E.r$ is the expression required to get E's value (E as an “r-value”), and $E.l$ is a function of one argument that sets E's value to that argument (E as an “l-value”). When $E.l$ is not specified, E is not an l-value.

Figure 7.11 describes the translation for expressions. Because expressions can be used for their value and also as an assignment location, I give the text required to get the value of the location and a function returning the text required to assign to the location separately, as $E.r$ and $E.l$ respectively. $E.l$ must be a function of one variable, because the variable is sometimes a function argument and other times not (contrast equation (7.77) with equation (7.82)).

$$R \to R_1 \mathbin{\&\&} R_2 \qquad R.gs = R_1.gs \| R_2.gs \tag{7.83}$$
$$R.gr = R_1.gr \| \text{“\&\&”} \| R_2.gr$$
$$R \to R_1 \,\|\, R_2 \qquad R.gs = \text{“if}(\text{Verify.random}(2) == 0)\{\text{”} \| R_1.gs \| \text{“}\}\text{else}\{\text{”} \| R_2.gs \| \text{“}\}\text{”} \tag{7.84}$$
$$R.gr = R_1.gr \| \text{“}||\text{”} \| R_2.gr$$
$$R \to R_1 \Longrightarrow R_2 \qquad R.gs = \text{“if(”} \| R_1.gr \| \text{“})\{\text{”} \| R_2.gs \| \text{“}\}\text{”} \tag{7.85}$$
$$R.gr = \text{“!”} \| R_1.gr \| \text{“}||\text{”} \| R_2.gr$$
$$R \to E_1 == E_2 \qquad R.gs = E_1.l(E_2.r) \| \text{“;”} \tag{7.86}$$
$$R.gr = E_1.r \| \text{“ == ”} \| E_2.r$$
$$R \to E_1.\text{equals}(E_2) \qquad R.gs = E_1.l(E_2.r) \| \text{“;”} \tag{7.87}$$
$$R.gr = E_1.r \| \text{“.equals(”} \| E_2.r \| \text{“)”}$$
$$R \to I[id] \qquad R.gs = \text{“Verify.random(”} \| I.l \| \text{“, ”} \| I.h \| \text{“); ”} \| \text{“Verify.ignoreIf(!”} \| I.r \| \text{“); ”} \tag{7.88}$$
$$R.gr = I.r$$
$$R \to \textbf{true} \qquad R.gs = \text{“;”} \tag{7.89}$$
$$R.gr = \text{“}\textbf{true}\text{”}$$
$$R \to \textbf{false} \qquad R.gs = \text{“}\textbf{fail};\text{”} \tag{7.90}$$
$$R.gr = \text{“}\textbf{false}\text{”}$$
$$R \to id \qquad R.gs = \text{“if (!”} \| id \| \text{“) }\textbf{fail};\text{”} \tag{7.91}$$
$$R.gr = id$$
$$R \to E.\text{isEmpty}() \qquad R.gs = E.r \| \text{“.clear(); ”} \tag{7.92}$$
$$R.gr = E.r \| \text{“.isEmpty()”}$$

Figure 7.12: Generation rules for JML relations, part 1. Here $R.gs$ are the statements required to ensure R is true and $R.gr$ is the expression stating whether R is upheld.

Figure 7.12 and figure 7.13 on the next page use these rules to translate relations. In some cases I need an expression that yields true or false depending on whether the relation succeeds or fails now, and in other cases I need a series of statements that ensures that if execution continues, the relation is true. These two requirements are separated

$$\begin{aligned}
R \to E_1.\mathsf{contains}(E_2) \qquad & R.gs = E_1.r \| \text{“.add(”} \| E_2.r \| \text{“);”} \\
& R.gr = E_1.r \| \text{“.contains(”} \| E_2.r \| \text{“)”}
\end{aligned} \tag{7.93}$$

$$\begin{aligned}
R \to \backslash\mathsf{fresh}(E_1, \ldots, E_n) \qquad & R.gs = \big\|_{i=1}^{n} \text{“if(seenObjects.contains(”} \| \\
& \qquad E_i.r \| \text{“))}\textbf{fail}\text{;”} \| \\
& \qquad \text{“seenObjects.add(”} \| E_i.r \| \text{“);”} \\
& R.gr = \text{“}\textbf{true}\text{”} \| \big\|_{i=1}^{n} \text{“\&\&”} \| E_i.r \| \text{“!= null \&\&”} \| \\
& \qquad\qquad \text{“!seenObjects.contains”} \| \\
& \qquad\qquad \text{“(”} \| E_i.r \| \text{“)”}
\end{aligned} \tag{7.94}$$

$$\begin{aligned}
R \to (\backslash\mathsf{forall}\ \mathit{Type}\ \mathit{id};\ S[\mathit{id}];\ R_1) \qquad & R.gs = \text{“for(”} \| \mathit{Type} \| \text{“ ”} \| \mathit{id} \| \text{“ : ”} \| S.i \| \text{“)”} \| \\
& \qquad \text{“\{”} \| R_1.gs \| \text{“\}”} \\
& R.gr = \text{“!}\mathit{Filter.filter}\text{(”} \| S.i \| \text{“, new } \mathit{UnaryFunctor}\text{()\{} \\
& \qquad\qquad \text{public } \mathit{Object} \text{ fn(”} \| \mathit{Type} \| \text{“ ”} \| \mathit{id} \| \text{“)\{} \\
& \qquad\qquad\quad \text{return”} \| R_1.gr \| \text{“;\}).}\mathit{empty}\text{()”}
\end{aligned} \tag{7.95}$$

$$\begin{aligned}
R \to (\backslash\mathsf{forall\ int}\ \mathit{id};\ I[\mathit{id}];\ R_1) \qquad & R.gs = \text{“for(”} \| \text{“int”} \| \mathit{id} \| \text{“ = ”} \| I.l \| \text{“;”} \| \\
& \qquad\qquad \mathit{id} \| \text{“ <= ”} \| I.h \| \text{“;”} \| \mathit{id} \| \text{“++)”} \| \\
& \qquad \text{“\{if(!”} \| I.r \| \text{“) continue;”} \| R_1.gs \| \text{“\}”} \\
& R.gr = \text{“!}\mathit{Filter.filter}(\mathit{arrayFrom}\text{(”} \| I.l \| \text{“,”} \| I.h \| \text{“),} \\
& \qquad\qquad \text{new } \mathit{UnaryFunctor}\text{()\{} \\
& \qquad\qquad\quad \text{public } \mathit{Object} \text{ fn(”} \| \mathit{Integer} \| \text{“ ”} \| \mathit{id} \| \text{“)\{} \\
& \qquad\qquad\quad \text{return”} \| I.r \| \text{“ \&\& ”} \| R_1.gr \| \text{“;\}).}\mathit{empty}\text{()”}
\end{aligned} \tag{7.96}$$

Figure 7.13: Generation rules for JML relations, part 2. Here $R.gs$ are the statements required to ensure R is true and $R.gr$ is the expression stating whether R is upheld.

into $R.gr$ and $R.gs$ respectively. An example of the need for this is in equation (7.85), where in the definition of $R.gs$ R_2 is made to be true if R_1 holds—to do this, I must have an expression for R_1, not a series of statements.

Some translations warrant specific note. Equation (7.87) on the preceding page calls the function $E_1.l$ with the r-value of E_2 to obtain the Java expression required to set E_1

to E_2. Equation (7.88) on page 114 creates an integer upholding an expression using the model checker's random number generation function. For efficiency's sake, rather than generate one of every possible integer that could be represented by a variable, I establish upper and lower bounds (here $I.h$ and $I.l$ respectively) and generate an integer between those. This may not be sufficient; for example, we may be presented with finding an x that satisfies $x > 5 \wedge x > 0 \wedge x < 10$. The lower bound here for x is clearly 1, but no value for x that is less than 6 will satisfy the expression. Thus, we ignore any values that do not satisfy the relation.

Equation (7.92) on page 114 ensures that the expression will be empty. Not all collections permit the clear method to be called, and there is no good way to determine whether any given collection is read-only. In the event that it is not, an exception will be thrown, so the error will be detected.

Equation (7.94) on the previous page uses an auxiliary global set seenObjects that stores all objects that have been "seen" and adds any that are "seen". This is not totally satisfactory as long existing objects that are never noticed by the grammar parser will be classified as fresh, but absent more intrusive measures or a theorem prover this is the best possible. It is sufficient to detect cycles in data structures.

Equation (7.95) on the preceding page and equation (7.96) translate quantification expressions over sets and integer ranges, respectfully. $S[id]$ and $I[id]$ restrict the types of

$$S[id] \rightarrow E.\mathsf{contains}(id) \qquad S.i = E.r \tag{7.97}$$

$$S[id] \rightarrow E.\mathsf{containsKey}(id) \qquad S.i = E.r \| \text{“}.\mathsf{keySet}()\text{”} \tag{7.98}$$

$$I[id] \rightarrow id > i \qquad I.l = i + 1 \qquad I.h = \top \qquad I.r = id \| \text{“} > i\text{”} \tag{7.99}$$

$$I[id] \rightarrow id >= i \qquad I.l = i \qquad I.h = \top \qquad I.r = id \| \text{“} >= i\text{”} \tag{7.100}$$

$$I[id] \rightarrow id > E \qquad I.l = \bot + 1 \qquad I.h = \top \qquad I.r = id \| \text{“} > \text{”} \| E.r \tag{7.101}$$

$$I[id] \rightarrow id >= E \qquad I.l = \bot \qquad I.h = \top \qquad I.r = id \| \text{“} >= \text{”} \| E.r \tag{7.102}$$

$$I[id] \rightarrow id < i \qquad I.l = \bot \qquad I.h = i - 1 \qquad I.r = id \| \text{“} < i\text{”} \tag{7.103}$$

$$I[id] \rightarrow id <= i \qquad I.l = \bot \qquad I.h = i \qquad I.r = id \| \text{“} <= i\text{”} \tag{7.104}$$

$$I[id] \rightarrow id < E \qquad I.l = \bot \qquad I.h = \top - 1 \qquad I.r = id \| \text{“} < \text{”} \| E.r \tag{7.105}$$

$$I[id] \rightarrow id <= E \qquad I.l = \bot \qquad I.h = \top \qquad I.r = id \| \text{“} <= \text{”} \| E.r \tag{7.106}$$

$$I[id] \rightarrow I_1[id] \&\& I_2[id] \qquad I.l = \max(I_1.l, I_2.l) \qquad I.h = \min(I_1.h, I_2.h) \qquad I.r = I_1.r \| \text{“}\&\&\text{”} \| I_2.r \tag{7.107}$$

$$I[id] \rightarrow I_1[id] || I_2[id] \qquad I.l = \max(I_1.l, I_2.l) \qquad I.h = \min(I_1.h, I_2.h) \qquad I.r = I_1.r \| \text{“}||\text{”} \| I_2.r \tag{7.108}$$

Figure 7.14: Generation rules for JML expressions. Here $S.i$ is the expression required to generate an iterator for all values matched by S, $I.l$ is the lowest possible value for *id*, $I.h$ is the highest possible value for *id*, and $I.r$ is a Boolean expression encoding the constraint.

the expressions *id* is quantified over to, in one case, be easily translatable set expressions, and in the other case be easily translatable integer expressions. The JML specification itself notes that not all tools will be able to handle arbitrary expressions here, so these limitations are fairly typical. Both expressions translate straightforwardly to loops for generation. Unfortunately, Java is not powerful enough to permit translating quantified expressions to Java expressions directly unless I make use of anonymous inner classes and auxiliary functions. Accordingly $R.r$ uses two libraries from the JGA [24] library to perform this operation. *Filter.filter*(S, f) returns a collection containing only the elements $s \in S$ where $f(s)$ holds. *UnaryFunctor* is the superclass for the anonymous inner classes required to make closures possible; it is related to the *Generator* discussed in section 6.2 on page 81. *arrayFrom*(l, h) is a custom library that creates an iterator that steps through every integer i where $l \leq i < h$. We could generate a temporary array for this, but this could require a very large array and generating the integers on-the-fly is more efficient.

Finally, figure 7.14 on the preceding page defines translation rules for these quantification expressions. For sets, I currently permit only quantification over a set, and quantification over the domain of a map. For integers, I permit a variety of relational expressions. Defining these is tedious but not difficult. In the case that *id* is compared with an arbitrary expression, I cannot, of course, know the bounds of that expression absent a theorem prover. I do not attempt it, setting the low and high values to be the extreme values for the type (in the case of integers, Integer.MIN_VALUE and

Integer.MAX_VALUE respectively). Conjunction of integer expressions maintains these values.

7.4.3 An example

After all this, an example is in order. I have re-expressed the examples in figure 7.4 on page 96 and in figure 7.5 on page 97 using this new framework. The resulting grammars can generate both the matching and generation code originally written while being approximately as simple as the generation and much simpler than the matching. These re-expressed examples are presented in figure 7.15 on the next page along with the corresponding shape types.

7.5 Verification Using JML

Correct clients				**Incorrect clients**					
deparent		*voider*		*reparent*		*increaser*			
Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Accounts	Entries
0:11	26	0:17	27	0:10	27	0:14	27	1	2
0:14	26	0:23	37	0:16	36	0:13	27	1	4
0:21	34	0:38	39	0:20	36	0:14	27	1	6
0:49	36	2:55	41	0:17	36	0;14	27	1	8
3:38	36	15:37	50	0:18	36	0:14	27	1	10

Table 7.2: Run time and memory usage vs. number of entries

Doubly linked list

Shape type

$$\begin{aligned} \textit{Doubly} &\rightarrow \mathbf{p}\ x, \mathbf{prev}\ x\ \text{null}, L\ x \\ L\ x &\rightarrow \mathbf{next}\ x\, y, \mathbf{prev}\ y\, x, L\ y \\ L\ x &\mid \mathbf{next}\ x\ \text{null} \end{aligned}$$

Grammar

$$\begin{aligned} \textit{Doubly}[x] &\rightarrow \textsf{ensure}\, x == \backslash\textsf{new}(\textit{Node})\ \&\&\ x.\textit{getPrev}() == \textsf{null};\ L[x] \\ L[x] &\rightarrow \Delta y\ \textsf{ensure}\, y == \backslash\textsf{new}(\textit{Node})\ \&\&\ x.\textit{getNext}() == y \\ &\qquad\qquad \&\&\ y.\textit{getPrev}() == x;\ L[y] \\ &\mid \textsf{ensure}\, x.\textit{getNext}() == \textsf{null}; \end{aligned}$$

Binary tree

Shape type

$$\begin{aligned} \textit{Bintree} &\rightarrow \mathbf{p}\ x, B\ x \\ B\ x &\rightarrow \mathbf{left}\ x\, y, \mathbf{right}\ x\, z, B\ y, B\ z \\ B\ x &\mid \mathbf{left}\ x\ \text{null}, \mathbf{right}\ x\ \text{null} \end{aligned}$$

Grammar

$$\begin{aligned} \textit{Bintree}[x] &\rightarrow \textsf{ensure}\, x == \backslash\textsf{new}(Node);\ B[x] \\ B[x] &\rightarrow \Delta y, z\ \textsf{ensure}\, y == \backslash\textsf{new}(\textit{Node})\ \&\&\ z == \backslash\textsf{new}(\textit{Node}) \\ &\qquad\qquad \&\&\ x.\textit{getLeft}() == y\ \&\&\ x.\textit{getRight}() == z; \\ &\quad B[y]\ B[z] \\ &\mid \textsf{ensure}\, x.\textit{getLeft}() == \textsf{null}\ \&\&\ x.\textit{getRight}() == \textsf{null}; \end{aligned}$$

Figure 7.15: JML expressions of earlier examples.

Correct clients				**Incorrect clients**					
deparent		*voider*		*reparent*		*increaser*			
Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Time (m:s)	Memory (MB)	Accounts	Entries
0:14	26	0:23	37	0:16	36	0:13	27	1	4
1:09	35	2:35	41	0:56	38	0:13	27	2	4
19:09	37	34:18	43	14:03	39	0:19	27	3	4

Table 7.3: Run time and memory usage vs. number of accounts

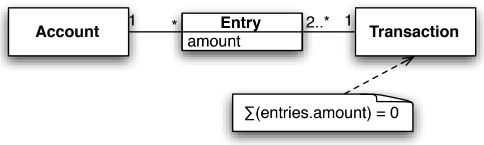


Figure 7.16: UML diagram of the Account pattern

As a demonstration of this work, I have written several small clients for an Enterprise Java Beans [50] (EJB) persistence layer. Other work on EJB layers is discussed in chapter 8 on page 126, including a description of the EJB framework itself. Here I use it as a convenient interface for handling some types of queries and performing relational integrity checks upon the resulting database.

I chose to base my clients around the Account pattern from Fowler [18]. Strictly speaking this is a pattern for an object schema; accordingly I have implemented it for these tests with the SQL mapping in the EJB framework. The Account pattern is useful for me here because it represents structured data and also has a hierarchical element.

A brief description of the Account pattern and how I interpreted it is in order. A UML diagram illustrating all this can be seen in figure 7.16. An *account* contains entries and can be a parent to other accounts; the account instances make up a forest. An *entry* is associated with exactly one account and exactly one monetary transaction, and has a field representing an amount of money. A *monetary transaction* is associated with at least two entries, and the sum of all entries in every monetary transaction must be zero at the end of a database transaction—this is often stated as “money is neither created nor destroyed”. Since unfortunately the term ‘transaction’ here refers to two distinct concepts both of which are important to us, I must be explicit: in the absence qualification, ‘transaction’ always refers to a monetary transaction.

This structure possesses a number of natural invariants. I have already mentioned the key transaction invariant. Accounts and their children must possess the tree property; that is no account can ever has two parents. The sum of all entries in all accounts in the system should also be zero; if it is not, I may have forgotten to store an account, an entry, or a transaction. Because I permit more than only two entries per transaction, my transactions are called multi-legged; it is usually considered undesirable or an outright error for one transaction to have more than one “leg” in any one account.

All these data invariants are in addition to the order in which the methods should be called. No query parameters should be adjusted following execution of the query. The queries themselves ought to be executed during a database transaction in order to obtain a consistent view of the database between each query. The *getResultList* or *getSingleResult* methods should be the last operation performed on the query object—these methods request either all results from a single database query or only one result.

I used my interface grammar compiler to create a stub for the EJB Persistence API that encodes all these invariants. Because the database can change in unpredictable and arbitrary ways between database transactions, my stub entirely regenerates the database every time a transaction is begun. If a transaction is rolled back, it could well be in an incomplete state and so applying database invariants is folly; yet if a transaction is committed it must be verified.

My stub contains two tunable parameters, corresponding to an upper bound on the number of accounts in the system and an upper bound on the number of entries in the system. The number of transactions in the system is always nondeterministically chosen to be between 1 and $\lfloor \mid \mathit{entries} \mid /2 \rfloor$, inclusive.

To exercise this stub, I wrote four EJB Persistence API clients, and ran the clients with varying parameters in the Java PathFinder model checker. Two stubs are correct in their use of the database and I expect that JPF will report this. Two are incorrect; one triggers a fault almost immediately, and the other is only invalid some of the time. My clients are as follows:

1. *deparent* takes an account and removes it from its parent. Changing the parent of an account can cause cycles if done naïvely, but removing the parent is always safe. I expect that as the number of accounts and entries increases verifying this

operation will take an exponentially increasing amount of time to complete, as normal for model checking.

2. *voider* selects a transaction and 'voids' it, by creating a new transaction negating the original transaction. This introduces new objects into the system, and so one would expect it to be slower than *deparent*. Since most of the work involves creating duplicate entry transactions, I would expect it to be more affected by increases in entries than by increases in accounts.

3. *reparent* takes two entries in the system and trades their transactions. If the entries encode the same monetary value this can be safe, but in the general case this operation will break the transaction invariant. An additional complication is that if there are less than four entries in the system, *reparent* cannot fail; there is only one transaction available. It is hard to say *a priori* how much time *reparent* will take to fail; the model checker's scheduler will determine this. However the proportion of the state space where *reparent* is valid to the size of the total state space decreases precipitously as the number of entries in the system increases. Accordingly I expect that it the running time will eventually come to some equilibrium if I increase the number of entries, but will consume an exponentially increasing amount of time if I hold the number of entries constant and increase the number of accounts.

4. *increaser* increases the monetary value of entries in the system. This will always trip the transaction invariant, with even two entries in the system, and so I expect it will take approximately the same amount of time to complete regardless of the size of the state space.

My results are presented in table 7.2 on page 119 and table 7.3 on page 121. My clients behave largely according to the expectations given above. *deparent* and *voider*'s run times increase with large jumps as the number of entries rises from 6 to 8 and from 8 to 10. Similarly their run times increase violently as the number of entries rise. *voider* is slower than *deparent*, although both are so sensitive to the number of accounts that I was unable to examine whether the exponential function for *voider* is exponentially higher for entries than it is for accounts. *reparent*'s run time does stabilize as entries increase, but for small numbers of entries and for varying numbers of accounts it does display an exponential increase in run time. And finally *increaser* is insensitive to either parameter; the jump from 13 seconds with 4 entries and 2 accounts to 19 seconds with 4 entries and 3 accounts is probably experimental error, as the amount of memory consumed did not increase.

Chapter 8

Applying Interface Grammars to EJB Clients

I have applied my technique and tool to the task of verifying clients of the Enterprise Java Beans 3.0 Persistence API.

8.1 Enterprise Java Beans 3.0 Persistence API

Enterprise Java Beans 3.0, or EJB 3.0, is the third major revision of the Enterprise Java Beans specification. The full specification is concerned with large scale software architecture with a web focus; I am interested here in the Java Persistence API, an affiliated but distinct API for object-relational mapping. That is, the Java Persistence API is a standardized interface to a framework for mapping a Java object graph to and from a relational database. The Persistence API in EJB 3.0 has been inspired by a number of third party object-relational mapping tools, including Hibernate and JDO, and in turn the new specification has been implemented independent of the EJB 3.0 framework; examples of this include Hibernate again and Glassfish.

The entry point to this API is the `EntityManager` interface, an instance of which is obtained from a `EntityManagerFactory`. The core of the interface is simple enough, with methods like *persist*, *remove*, *find* and *contains*. Each `EntityManager` has an associated transaction object, and a common idiom includes code like `em.getTransaction().begin();`.

Objects in the Persistence API have a four phase life-cycle:

- unmanaged, or transient objects are not stored in the database—for example, newly created objects;
- persistent objects are stored in the database;
- detached objects are persistent objects that have become separated from their EntityManager—this becomes useful in certain situations concerning long lived client objects where a long term database transaction is undesirable;
- removed objects are scheduled to be removed from the database.

The mapping from an object to a relational table is supported by Java annotations on the classes, fields and methods of data objects. For example, all classes intended to participate in the Persistence API must have the `Entity` annotation on the class, marking it as an entity bean. The primary key can be marked with `Id` and can be attached to methods or fields, and as well methods can be marked to be executed before or after database events like insertion or updates.

The Persistence API also contains a query language similar to SQL. I do not consider a simulation of the query language in this paper, largely because simulating it properly would require a full string parser for the SQL-like syntax. As well, my interface specification does not model concurrent update operations or the XML extension defined by the Persistence API.

8.2 Persistence API clients

The normal life cycle of a Persistence API client is to retrieve the global `EntityManagerFactory`, use it to obtain a thread-local `EntityManager`, begin a transaction, modify the database, and then commit or rollback the transaction. Misbehaving clients, or even properly behaving clients in some circumstances can trigger exceptions during this process. Some of these exceptions are pedestrian—for example, calling `flush` outside of a transactional context—but others are more alarming.

As an example of the latter, the `getReference` method returns a proxy for a database object. This proxy can serve as a stand in for the real object in many cases, and is used when making a separate database query to retrieve the object is undesirable—for example, chasing links in a tree. An eager loading implementation might end up loading

component *transaction* **implements** *EntityTransaction*:
$\langle\langle$*entity_manager* m; $\rangle\rangle$

$$
\begin{array}{rcl}
\textit{start} & \rightarrow & \textit{inactive};\\
\\
\textit{inactive} & \rightarrow & ?\textit{begin}();\ !\textit{begin}(m);\ \text{¡}\textit{begin}();\ \text{¿}\textit{begin}();\ \textit{active};\\
 & | & ?\textit{isActive}();\ \text{¿}\textit{isActive}(\langle\langle\textbf{false}\rangle\rangle);\\
 & | & ?\textit{getRollbackOnly}();\ \text{¿}\textit{getRollbackOnly}(\langle\langle\textbf{false}\rangle\rangle);\\
 & | & \epsilon\\
\\
\textit{active} & \rightarrow & ?\textit{commit}();\ !\textit{commit}(m);\ \text{¡}\textit{commit}();\ \text{¿}\textit{commit}();\ \textit{inactive};\\
 & | & ?\textit{commit}();\ !\textit{rollback}(m);\ \text{¡}\textit{rollback}();\\
 & & \langle\langle\textbf{throw new}\ \textit{RollbackException}\rangle\rangle;\ \text{¿}\textit{commit}();\ \textit{inactive};\\
 & | & ?\textit{setRollbackOnly}();\ \text{¿}\textit{setRollbackOnly}();\ \textit{rollback\_only};\\
 & | & ?\textit{isActive}();\ \text{¿}\textit{isActive}(\langle\langle\textbf{true}\rangle\rangle);\ \textit{active};\\
 & | & ?\textit{getRollbackOnly}();\ \text{¿}\textit{getRollbackOnly}(\langle\langle\textbf{false}\rangle\rangle);\ \textit{active};\\
 & | & ?\textit{rollback}();\ !\textit{rollback}(m);\ \text{¡}\textit{rollback}();\ \text{¿}\textit{rollback}();\ \textit{inactive};\\
\\
\textit{rollback\_only} & \rightarrow & ?\textit{setRollbackOnly}();\ \text{¿}\textit{setRollbackOnly}();\ \textit{rollback\_only};\\
 & | & ?\textit{isActive}();\ \text{¿}\textit{isActive}(\langle\langle\textbf{true}\rangle\rangle);\ \textit{rollback\_only};\\
 & | & ?\textit{getRollbackOnly}();\ \text{¿}\textit{getRollbackOnly}(\langle\langle\textbf{true}\rangle\rangle);\ \textit{rollback\_only};\\
 & | & ?\textit{rollback}();\ !\textit{rollback}(m);\ \text{¡}\textit{rollback}();\ \text{¿}\textit{rollback}();\ \textit{inactive};
\end{array}
$$

Figure 8.1: A portion of the EJB interface grammar that specifies the nonrecursive transactional interface constraints.

the entire tree into memory one node at a time by requesting parents and children, when only one node is needed.

The part that makes this alarming is that the presence of the referenced object is not checked at method call time; instead, it is checked the first time data from the putative

$\textbf{component}\ \textit{recursive\_transaction}\ \textbf{implements}\ \textit{EntityTransaction}$:
$\langle\!\langle \textit{entity\_manager}\ m; \textit{int}\ l; \textit{bool}\ r; \rangle\!\rangle$

$$
\begin{array}{rcl}
\textit{start} & \rightarrow & \textit{base}; \\
\\
\textit{base} & \rightarrow & ?\textit{begin}(); \langle\!\langle l \leftarrow l + 1 \rangle\!\rangle; !\textit{begin}(m); \text{¡}\textit{begin}(); \text{¿}\textit{begin}(); \\
 & & \textit{base}; \textit{tail}; \textit{base}; \\
 & | & ?\textit{setRollbackOnly}(); \langle\!\langle r \leftarrow \textbf{true} \rangle\!\rangle; \\
 & & \text{¿}\textit{setRollbackOnly}(); \textit{base}; \\
 & | & ?\textit{isActive}(); \text{¿}\textit{isActive}(\langle\!\langle l > 0 \rangle\!\rangle); \textit{base}; \\
 & | & ?\textit{getRollbackOnly}(); \text{¿}\textit{getRollbackOnly}(\langle\!\langle r \rangle\!\rangle); \textit{base}; \\
 & | & \\
\\
\textit{tail} & \rightarrow & [\![\neg r]\!]; ?\textit{commit}(); !\textit{commit}(m); \text{¡}\textit{commit}(); \\
 & & \langle\!\langle l \leftarrow l - 1; r \leftarrow r \wedge \neg(l \equiv 0) \rangle\!\rangle; \text{¿}\textit{commit}(); \\
 & | & [\![\neg r]\!]; ?\textit{commit}(); !\textit{rollback}(m); \text{¡}\textit{rollback}(); \\
 & & \langle\!\langle l \leftarrow l - 1; r \leftarrow r \wedge \neg(l \equiv 0) \rangle\!\rangle; \\
 & & \langle\!\langle \textbf{throw new}\ \textit{RollbackException} \rangle\!\rangle; \text{¿}\textit{commit}(); \\
 & | & ?\textit{setRollbackOnly}(); \langle\!\langle r \leftarrow \textbf{true} \rangle\!\rangle; \\
 & & \text{¿}\textit{setRollbackOnly}(); \textit{tail}; \\
 & | & ?\textit{isActive}(); \text{¿}\textit{isActive}(\langle\!\langle \textbf{true} \rangle\!\rangle); \textit{tail}; \\
 & | & ?\textit{getRollbackOnly}(); \text{¿}\textit{getRollbackOnly}(\langle\!\langle r \rangle\!\rangle); \textit{tail}; \\
 & | & ?\textit{rollback}(); !\textit{rollback}(m); \text{¡}\textit{rollback}(); \\
 & & \langle\!\langle l \leftarrow l - 1; r \leftarrow r \wedge \neg(l \equiv 0) \rangle\!\rangle; \text{¿}\textit{rollback}();
\end{array}
$$

Figure 8.2: A portion of the EJB interface grammar that specifies the recursive transactional interface constraints.

object is referenced. This could be in an entirely different piece of code, a piece of code unrelated to the database.

Another example of a properly behaving client nonetheless triggering an exception is in committing a successful transaction; because the Persistence API supports optimistic locking it is possible that a commit can be aborted because the database row corresponding to the object in question has changed since it was first read, with no possibility of safe detection by the user code.

These consequences, and the difficulty of verifying properties of a program that depends intimately on an enormous third party database for its operation, motivate some sort of modular analysis that captures all these strange error conditions but yet is not too heavyweight to be used; thus I applied my interface grammar tools to the Persistence API. I can also use my framework to analyze extensions to the API; one such extension might be recursive transactions, which are not supported in EJB 3.0 but are very common in the databases themselves.

To verify clients, I have written interface grammars for each relevant interface: `EntityManagerFactory`, `EntityManager` and `EntityTransaction`. Portions of these grammars are shown in figure 8.1 on page 129 and figure 8.2 on the previous page. Our grammars in total are some 474 lines long, defining all three fundamental classes and their behaviors; by comparison the abstract class in Hibernate that defines just the EntityManager interface is some 657 lines long, and the total code required to implement the Persistence API using Hibernate as a backend is some 64,000 lines of Java code.

Table 8.1: “Correct” execution times

	Execution time (in ms)								
	One repetition			Three repetitions			Five repetitions		
Test	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.
Cont	3133	6662	10143	11898	31567	51930	26914	73628	126342
Clear	2115	3959	5876	6424	16688	27276	13338	37395	63491
Pers	2191	4247	6304	6974	18584	29622	15062	41746	70251
Open	1928	3473	4835	5659	14345	22306	12102	32407	52011
Bidir	2420	5288	8829	7504	22810	43396	16536	52894	99923
Merge	2358	4517	6992	7100	19122	32617	14839	43074	75385
CBack	2122	4018	6164	6357	16843	28087	13132	37677	64340
Exc	2100	4213	6399	6341	16903	29138	13144	39010	68074
Get	1944	3817	5962	5364	15242	27762	11053	34712	63903
Nonex	1816	3148	4237	4946	11637	18502	9996	26663	43501
Inher	2556	5374	8931	7867	23297	42481	17296	54548	102449
Trans	2520	6121	11784	7882	27999	58125	17122	64811	138157

8.3 Experiments

I have applied these grammars to several test cases from the Hibernate implementation. In some sense these are excellent measures of the fidelity of my interface; since they were written to expose errors in Hibernate they should similarly expose errors in my simulation of the Persistence API. As well, the test cases include some invalid clients that trigger exceptions; I can use these to verify clients against the interface, marking clients with erroneous behavior.

Table 8.2: "Correct" execution state counts

	States visited								
	One repetition			Three repetitions			Five repetitions		
Test	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.
Cont	66	198	338	377	1157	1985	932	2872	4932
Clear	32	96	168	173	545	965	422	1342	2382
Pers	41	123	213	227	707	1235	557	1747	3057
Open	28	78	128	149	437	725	362	1072	1782
Bidir	46	162	318	257	941	1865	632	2332	4632
Merge	36	114	208	197	653	1205	482	1612	2982
CBack	27	81	143	143	455	815	347	1117	2007
Exc	27	90	173	143	509	995	347	1252	2457
Get	15	57	123	71	311	695	167	757	1707
Nonex	12	30	48	53	149	245	122	352	582
Inher	44	168	348	245	977	2045	602	2422	5082
Trans	45	222	523	251	1301	3095	617	3232	7707

Table 8.3: "Incorrect" execution times

	Execution time (in ms)								
	One repetition			Three repetitions			Five repetitions		
Test	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.
Cont	1913	1910	1908	1905	1912	1909	1927	1897	1933
Get	1809	1807	1799	1799	1801	1807	1800	1800	1817
Nonex	1785	1785	1772	1778	1779	1776	1774	1774	1791
Trans	2064	2060	2085	2076	2068	2066	2062	2070	2058

Table 8.4: "Incorrect" execution state counts

	States visited								
	One repetition			Three repetitions			Five repetitions		
Test	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.	1 Obj.	3 Obj.	5 Obj.
Cont	25	25	25	25	25	25	25	25	25
Get	8	8	8	8	8	8	8	8	8
Nonex	10	10	10	10	10	10	10	10	10
Trans	23	23	23	23	23	23	23	23	23

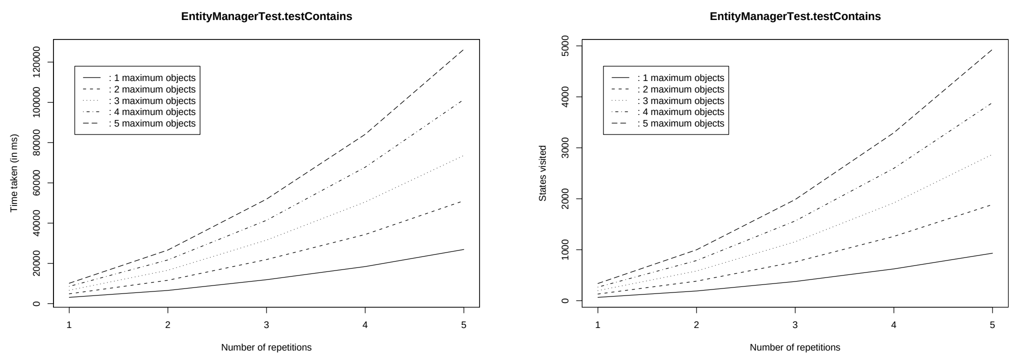


Figure 8.3: Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **Cont**

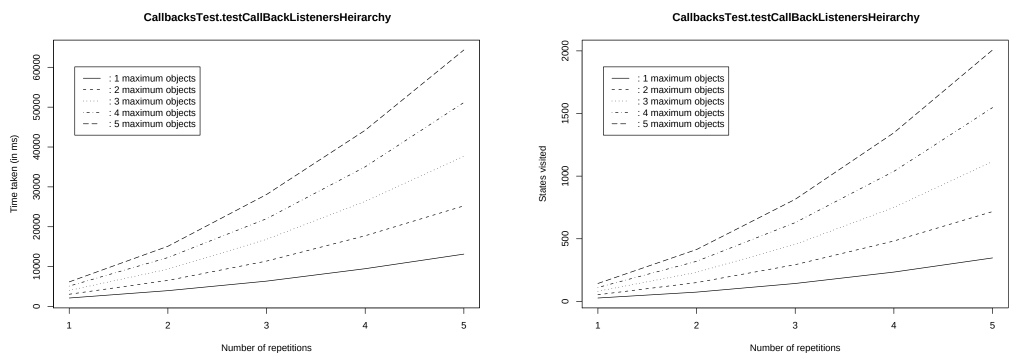


Figure 8.4: Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **CBack**

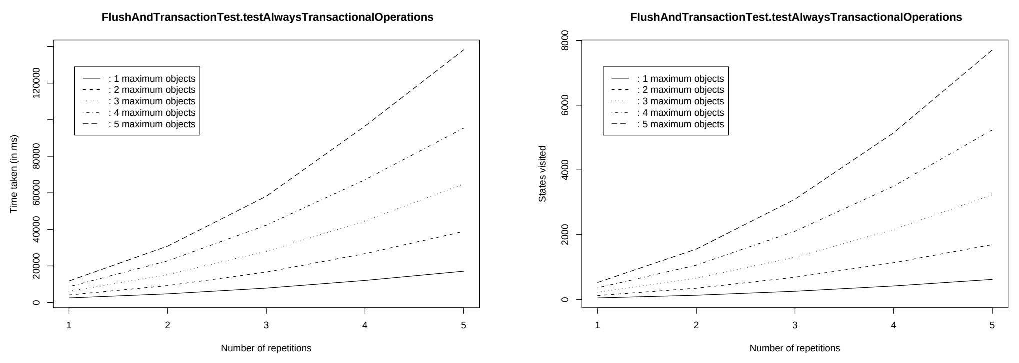
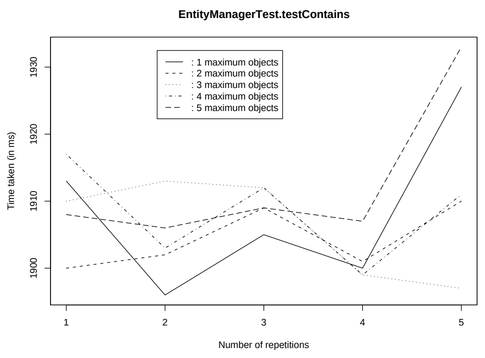


Figure 8.5: Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **Trans**



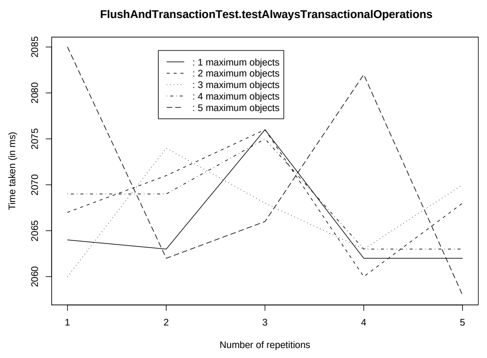


Figure 8.6: Run time for "incorrect" execution for **Cont** and **Trans**

To increase legibility, I give the full test name here once, and refer to an abbreviation in future. I have analyzed the following test cases:

- *EntityManagerTest.testContains* (**Cont**) tests minimal normal functionality, like persisting an object and retrieving it under its primary key. It also ensures that trying to check the status of a non-manageable object will fail with an exception.
- *EntityManagerTest.testClear* (**Clear**) ensures that objects managed by the *EntityManager* transition to the detached state after a `clear`.
- *EntityManagerTest.testPersistNoneGenerator* (**Pers**) ensures that a simple object is equal to itself after it has been persisted and reloaded.
- *EntityManagerTest.testIsOpen* (**Open**) verifies that an *EntityManager* is open upon creation and stays that way until it is closed.

- *AssociationTest.testBidirOneToOne* (**Bidir**) verifies that persisting one half of a bidirectional association will persist the other half as well.
- *AssociationTest.testMergeAndBidirOneToOne* (**Merge**) verifies that the bidirectional association works even with detached objects.
- *CallbacksTest.testCallBackListenersHeirarchy* (**CBack**) verifies that all methods tagged with *@PrePersist* are called when the object in question is persisted.
- *CallbacksTest.testException* (**Exc**) verifies that methods in other classes that have a declared *@EntityListeners* relationship with the persisted object are also called. The name comes from the method that is to be called, which throws an exception.
- *GetReferenceTest.testWrongIdType* (**Get**) verifies that asking for objects using the wrong primary key type is an illegal operation.
- *ExceptionTest.testEntityNotFoundException* (**Nonex**) verifies that nonexistent objects fetched with *getReference* should raise exceptions when they are referred to.
- *InheritanceTest.testFind* (**Inher**) verifies that if A is a subclass of B, persisting an instance of A and asking for all Bs should retrieve the first object.
- *FlushAndTransactionTest.testAlwaysTransactionalOperations* (**Trans**) checks that flushes and locks are only valid from within transactions.

As befits a good test harness, some of these tests verify that correct use of the API gives correct results, and the rest verify that incorrect use of the API is flagged as such. Errors are flagged by throwing an exception. The test case itself captures the exception and verifies that it is of the correct type.

This has an attractive corollary. While I can verify that my interface represents a valid implementation of the EJB Persistence API by running all the test cases as written, I can also use the test cases that incorrectly use the API to simulate an incorrect client. I do so by modifying the test cases to re-throw any exceptions they catch. In my results in Tables 8.1 through 8.4, I distinguish these two different modes as “correct” and “incorrect” executions. Note that, some test cases do not simulate an incorrect client, and so can only run in the “correct” mode.

All these test cases have been written as unit tests with swift execution in mind. While useful for test cases, they do not necessarily simulate large clients well. In an attempt to simulate larger clients, I have parametrized each test case in two dimensions; the first dimension specifies the maximum number of repetitions of the operation under test, and the second specifies the maximum number of objects created in the test. Because most operations only make sense with newly allocated data (for example, persisting a fresh object), each repetition reallocates up to the maximum number of objects. In this fashion I measure the overhead introduced as the number of operations goes up, and as the number of objects in the mix increases.

These bounds represent constraints on the maximum number of repetitions, not the number of repetitions itself. That is, when I report the run time for, say, **Get** running with 3 objects and 5 repetitions, I am reporting the time required to run that test with each combination of 1 to 3 objects and 1 to 5 repetitions, for a total of 15 runs.

I ran each of the 11 tests twice (in "correct" and "incorrect" modes, here "error state FALSE" and "error state TRUE"), and with the maximum number of objects varying from 1 to 5, and with the maximum number of repetitions varying from 1 to 5; this represents over 500 runs. This is far more than can be conveniently displayed in a table. Accordingly, I have excerpted representative results into Tables 8.1 and 8.2, and present several graphs showing selected results for "correct" execution in Figure 8.3 through Figure 8.5. I present the number of states JPF has visited as a more representative measure of memory consumption than the values reported by JPF; standard memory allocation algorithms request only large blocks of memory from the operating system, and so these figures are too coarse grained for my purposes. I give timing and state data for "incorrect" execution in Tables 8.3 and 8.4 and show some graphs displaying timing results for "incorrect" execution in Figure 8.6; I have omitted the state graphs for "incorrect" execution because, as shown in Table 8.4, there is no increase in the number of states processed in order to detect an error in any of those tests.

My expectations for this data was that "correct" execution would show a polynomial increase in time and memory usage as the number of objects and number of repetitions was increased; specifically it would increase by some function of complexity $O(n^2m^2)$ where n is the number of objects and m is the number of repetitions. For "incorrect" execution, since I halt verification the moment an error is detected it is difficult to predict the run time or memory usage.

The expectation I set for "correct" execution is due to the following reasoning. There are $O(nm)$ test runs made for any combination of n objects and m repetitions. A test run with a maximum of 3 objects and a maximum of 5 repetitions performs $1 + 2 + 3$ object allocations and performs $1 + 2 + 3 + 4 + 5$ operations; in general we will see $\sum_{i=1}^{n} i$ object allocations for each operation, and $\sum_{i=1}^{m} i$ operations. This means that a test run for n objects and m repetitions will see $O(\frac{n(n-1)}{2}) = O(n^2)$ object allocations per operation and similarly $O(\frac{m(m-1)}{2}) = O(m^2)$ operations. If object allocations and API operations are the dominant contributors to test case run time, as seems plausible, then I would see the run time follow a doubly quadratic curve $O(n^2m^2)$. Because JPF stores visited states indefinitely, I would also expect the memory usage to increase by this same parameter.

I find that this expectation is upheld in all my tests, save for the "incorrect" execution. That is, every test using "correct" execution displays a clearly super-linear increase as each parameter is increased. Because JPF aborts execution as soon as any error is found, the execution times for "incorrect" execution is uniformly low (below 2 seconds) and dominated by noise that I cannot control for; for example variances in start up time, cool caches or disk access.

My results also demonstrate the efficiency of my approach. For all tests the run time for repeating the API operation 5 times is less than three times the run time for creating 5 objects, and frequently considerably less. A test repeating an operation 5 times but creating one object will execute 5 operations but also perform 5 object allocations; a test repeating an operation once but creating 5 objects will execute one operation and 5 object allocations. The fact that the execution of these two is comparable implies I have reduced the execution time required to perform API operations to approximately the same time required for object allocation.

Chapter 9

Client and Server Verification for Web Services Using Interface Grammars

I also applied my interface grammars to verification of web services clients and servers; specifically I targeted the Amazon Web Services [3] framework, which is based on the Web Services Description Language [62], or WSDL. WSDL is a language for specifying web service interfaces.

A WSDL specification lists the available operations of a service and the types of the arguments of those operations. Based on the WSDL specification, one can implement a client that calls the operations of the web service. In this framework, conformance to the interface specification becomes very important. If either the client or the server deviate from the interface specification, the client-server interaction will lead to errors. Note that it may not be easy to test the client and server together since they may not belong to the same organization.

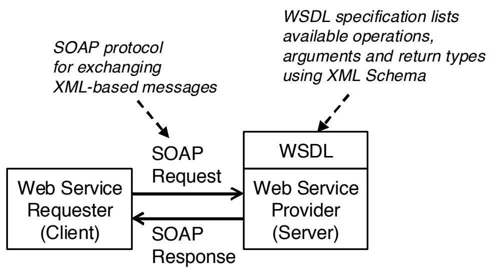

SOAP protocol for exchanging XML-based messages
WSDL specification lists available operations, arguments and return types using XML Schema
SOAP Request
SOAP Response
WSDL
Web Service Requester (Client)
Web Service Provider (Server)

Figure 9.1: Basic architecture for web services

Web services interact with each other by exchanging messages encoded using the eXtensible Markup Language (XML) [64]. XML Schema [65] provides a type system for XML messages and the Simple Object Access Protocol (SOAP) [48] is a standard communication protocol for transmitting XML messages. Each web service has to publish its invocation interface, e.g., network address, ports, operations provided, and the expected XML message format to invoke the service, using WSDL. In its basic form, the web service architecture consists of a simple RPC model where a client invokes operations exported by a service provider using the SOAP protocol as seen in figure 9.1. The WSDL specification serves as the contract between the client and the server that defines the valid interactions.

My web service verification framework is shown in figure 9.2 on the next page. It consists of two tools: a WSDL-to-interface grammar translator and my interface compiler. As the first step the keywordwsdl to interface grammar translator takes a WSDL specification as input and converts it to two interface grammars: one representing

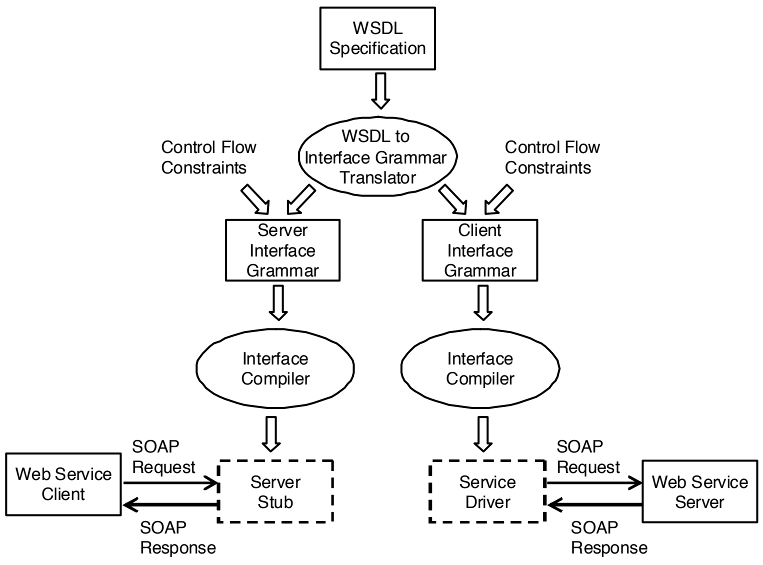


Figure 9.2: My web service verification framework

the behavior of the server and the other one representing the behavior of the client. Then, I use my interface compiler to generate a server stub and a service driver from these interface grammars, respectively.

9.1 Amazon Web Service

One challenge in conducting a case study on WSDL is the fact that most large and mature web service implementations are proprietary and most public WSDL specifications are for small toy examples such as currency converters, temperature converters, etc.

One notable exception to this is the Amazon Web Services (AWS) provided by Amazon.com. AWS is a large framework including various services such as Amazon Elastic Compute Cloud (Amazon EC2) for cloud computing, Amazon Flexible Payments Service (Amazon FPS) for financial transactions, which require payment. The Amazon Associates Web Service (also known as the Amazon E-Commerce Service or "ECS"), on the other hand, is a free service that exposes Amazon's product data with the goal of driving traffic back to Amazon's web sites or sales of Amazon products and services. I chose this service as the target of my web services case study.

The Amazon E-Commerce Service (which I will hence refer to as AWS-ECS) provides access to Amazon's product data through a SOAP interface specified with WSDL. The scope of the full AWS-ECS is enormous since it provides information about the wide variety of goods Amazon.com sells. The WSDL specification for the AWS-ECS lists 40 operations, almost all of which are differing ways of searching Amazon's product database. I focused on what I believed to be the core of the AWS-ECS API, consisting of the `ItemSearch`, `CartCreate`, `CartAdd`,`CartModify`, `CartGet`, and `CartClear` operations. Notably absent is any manner for automatically purchasing items in a cart. Amazon prefers that, after the user finds what he/she is looking for, the AWS-ECS clients direct the user to an Amazon web page for processing purchasing transactions.

I can informally define the semantics of these operations as follows:

- `ItemSearch` searches Amazon's database for items matching some set of keyword parameters. It returns a list of products that match the search criteria.
- `CartCreate` takes an ASIN—a unique identifier for that item in Amazon's database—and a positive integer n and creates a new cart that represents a request for n copies of that item. All AWS-ECS operations require absolute quantities; that is, it is impossible to say 'add one more of this to the cart'; you must first discover the contents of the cart and then send a message updating them accordingly.
- `CartAdd` takes a cart, an ASIN and a positive integer n and adds a new row to the cart requesting n copies of that item. It returns the modified cart and a unique item identifier for the new row in that cart. It is illegal to add an ASIN to a cart that already has a row for that ASIN.
- `CartModify` takes a cart, an item identifier, and a non-negative integer n and alters the row in the cart signified by the item identifier so that it requests instead n copies of the item. The row referred to by the item identifier must exist. If $n = 0$, then the row is deleted entirely. Even though there can only be one row for each ASIN, `CartModify` only takes the item identifier returned by `CartAdd` or `CartCreate`.
- `CartGet` is a query operations that takes a cart and returns the contents of the cart.

- `CartClear` takes a cart and removes everything from it, emptying it out. This operation represents one of only two ways to achieve an empty cart; the other is using `CartModify` with $n = 0$.

One piece of my framework is to convert the above constraints about the operations provided by a web service to an interface grammar that can be used to check the web service providers (i.e., servers) and their clients in terms of conformance to these constraints. I am able to generate a portion of these constraints automatically by my WSDL to interface grammar translator. However, this is not sufficient for all the constraints.

The above six operations also have several control flow constraints that are not stated in the WSDL specification of the AWS-ECS. Except for `ItemSearch` and `CartCreate`, every operation requires a cart already exist. `CartModify` can only be called after an item identifier has been retrieved from `CartCreate` or `CartAdd`. It is illegal to call `CartModify` with anything, including previously valid data, after a `CartClear`. Worse yet, this control flow is data dependent; one `CartModify` after another on the same item identifier can be okay if the first has an $n > 0$; and they can be fine if on different item identifiers.

From the point of view of the WSDL specification, none of this is specified. A WSDL specification only declares that certain operations exist, which take parameters of a certain type; it says nothing about ordering. If I want to verify that the web service operations are called in the right order, it is necessary to provide some sort of auxiliary specification that identifies these control flow constraints. In my framework, I ask the user to specify these control flow constraints directly on the interface grammar generated by our WSDL to interface grammar translator.

Specifically for the AWS-ECS, I have written two grammars, one for clients of the service and one as a client of the service, in figure 9.3 on page 165 and figure 9.4 on page 166.

9.2 Translating WSDL to interface grammars

As I mentioned earlier, a WSDL specification is a list of exposed operations along with the type of the parameters and return values. I have developed a tool to translate a WSDL specification to an interface grammar specification. To generate an interface grammar specification from a WSDL specification, we need to know in what order the operations should occur (i.e., the control flow constraints) and we need to be able to generate instances of the parameters for the operations and verify the return values.

It is relatively easy to express the control flow constraints directly as a grammar. However, generating instances of the parameters and verifying the return values is more difficult and tedious. Since the type information is available in the WSDL specification, there is no user input necessary for specifying such constraints. In fact I developed a translator that automatically generates an interface grammar for such constraints from a given WSDL specification.

9.2.1 Translation from XML Schema to interface grammars

A WSDL specification encodes all types using XML Schema. To automatically generate sample XML documents from an XML Schema using grammars, I use the following translation rules. Since XML Schema itself is very verbose, I use the MSL [11] formalism of Brown, et al., which encodes all of XML Schema in a more compact form. At present my translator does not handle all of XML Schema—omitting, for example, unordered types—merely the portions I found necessary in this work. Accordingly I define a simplified version of MSL whose type expressions can be defined using the following grammar:

$$g \rightarrow b \,\Big|\, t[g_0] \,\Big|\, g_1\{m, n\} \,\Big|\, g_1, \ldots, g_k \,\Big|\, g_1| \ldots |g_k \tag{9.41}$$

Here $g, g_0, g_1, \ldots, g_k$ are all MSL types; b is a basic data type such as Booleans, integers, or strings; t is a tag; and m and n are natural numbers such that $m < n$, or alternatively m may be an arbitrary natural number and n may be ∞.

As to interpretation, I regard $g \rightarrow b$ specifying a basic type; $g \rightarrow t[g_0]$ specifying the sub-element t of g, whose contents are described by the type expression g_0; $g \rightarrow$

$g_1\{m, n\}$ where $n \neq \infty$ specifying an array of g_1s with at least m elements and at most n elements; $g \rightarrow g_1\{m, \infty\}$ specifying an unbounded array of g_1s with at least m elements; $g \rightarrow g_1, \ldots, g_k$ specifying sequencing, with each of the g_is one after the other; and $g \rightarrow g_1 | \ldots | g_k$ specifying alternation, where g is one of the g_is. I denote the language of type expressions generated by equation (9.41) on the preceding page to be $\mathcal{XML}$.

I can generate sample XML documents using these type expressions; but my goal is to communicate with a SOAP server. I could write my own SOAP client library, but this is difficult and time consuming. I chose instead to use Apache Axis. Axis is a very suitable library, but it requires a Java object graph in a very specific form which it will then serialize to XML. Accordingly I create Java objects from type expressions, and do so in the same way that Axis maps WSDL to Java objects.

XML Schema and the Java type system are very different and, hence, mapping from one to the other is not trivial. However, since such a mapping is already provided by Axis, all I have to do is the follow the same mapping that Axis uses. Axis maps type expressions as in equation (9.41) on the previous page to Java as follows:

1. $g \rightarrow b$ is mapped to a Java basic type when possible (for example, with Booleans or strings). Because XML Schema integers are unbounded and Java integers are not, I must use a specialized Java object rather than native integers.

2. $g \rightarrow t[g_0]$ is mapped to a new Java class whose name is the concatenation of the current name and t; this class contains the data in g_0, and will be set to the t field in the current object.

3. $g \rightarrow g_1\{0, 1\}$ is mapped to either **null** or the type mapped by g_1.

4. $g \rightarrow g_1\{m, n\}$ is mapped to a Java array of the type mapped by g_1.

5. $g \rightarrow g_1, \ldots, g_k$ is mapped to a new Java class that contains each of the g_is as fields.

6. $g \rightarrow g_1|\ldots|g_k$ is mapped to a new Java interface that each of the g_is must implement.

This mapping contains omissions; **boolean**, **string**|**int** for example does not map readily to any Java object. These mismatches are inevitable consequences of the mismatch between the XML Schema and Java type systems, and are handled in an ad hoc manner. Fortunately, WSDL specifications in practice are usually deliberately engineered to avoid these sorts of corner cases.

The rules for the WSDL to interface grammar translation is shown in figure 9.5 on page 167. The translation is defined by the function $\mathbf{p}$, which uses the auxiliary functions $\mathbf{r}$ (which gives unique names for type expressions suitable for use in grammar nonterminals) and $\mathbf{t}$ (which gives the name of the new Java class created in the Axis mapping of $g \rightarrow t[g_0]$). By applying $\mathbf{p}[\![g]\!]$ to an XML Schema type expression I compute several grammar rules to create Java object graphs for all possible instances of that type expression; the start symbol is $\mathbf{r}[\![g]\!]$.

An explanation of each rule is in order, followed by an example.

Rule 9.42 on page 167 translates Boolean types by simply enumerating both possible values. Calling $\mathbf{r}[\![g]\!](x)$ with an uninitialized variable x will set x to either **true** or **false**.

Rule 9.43 on page 167 translates natural numbers to a Java instance by starting at 0 and executing an unbounded number of successor operations. This is inefficient; in practice if the number is bounded I can generate it far more efficiently. This does make the set of generated object graphs infinite, but if the number itself is not bounded the type expression itself generates an infinite language. I can accommodate general integers by performing this operation and then choosing a sign.

Rule 9.44 on page 167 translates strings to Java strings. This again starts with an empty string and concatenates an unbounded number of characters onto it, to exhaust the state space. It should be noted that almost no program and certainly no WSDL specification that says it will accept any string really means that; strings are frequently used as unspecified enumerations, have unspecified (and possibly un-specifiable) correlations with other parts of the object graph, have some associated structure they should maintain (as in search queries), etc. Accordingly, refining the automatically generated grammar to something more restricted but also more useful starts by changing these rules.

Rule 9.45 on page 167 translates tags into Java objects. The rule is simple; we figure out which Java type Axis is using for this position using $\mathtt{t}[\![g]\!]$, if it is not already initialized (which can happen if we are applying rule 9.50 on page 167) instantiate it, recursively process its contents, and then set the contents to the t field on the object we are currently working on.

Rule 9.48 on page 167 translates optional elements into Java objects by having two rules, one for **null** and the other to generate the object.

Rule 9.47 on page 167 translates unbounded arrays into Java objects. I use essentially the same operation as in rule 9.43 on page 167; start with the base case of an empty array and concatenate objects onto it until we decide to stop.

Rule 9.48 on page 167 translates bounded arrays into Java objects, by simply generating n rules, one for each potential object. Although I give this simple rule here for readability, in our implementation I handle this case more efficiently.

Rule 9.49 on page 167 translates general arrays, that may have a minimum number of objects greater than 0, to a situation where one of rule 9.47 on page 167 or rule 9.48 on page 167 applies.

Rule 9.50 on page 167 translates sequences into Java objects; we simply apply each of the sub-rules to the object graph under examination in sequence.

Rule 9.51 on page 167 translates alternations into Java objects; we pick one of the sub-rules and apply it. The difficult part of this, making each of $g, g_1, \ldots, g_k$ acceptable Java objects, is handled for us by Apache Axis.

Finally, an example is in order. Consider the following type (a simplified version of the `ResponseGroupInformation` element in the Amazon WSDL specification):

$$t = \textbf{name}[\textbf{string}]\{0,1\}, \textbf{ops}[\textbf{name}[\textbf{string}]\{0,\infty\}] \tag{9.52}$$

In this, we have in sequence an optional **name** tag, which contains a string, and a **ops** tag which contains an unbounded array of strings. Translating this to a grammar, we first apply rule 9.50 on page 167. To the first element of the sequence we apply rule 9.46 on page 167, rule 9.45 on page 167 and finally rule 9.44 on page 167, resulting in the following grammar:

$$a(x) \rightarrow x \leftarrow \text{``''}$$

$$a(x) \rightarrow y \leftarrow c; a(x); x \leftarrow y \| x \quad \text{for every character } c$$

$$b(x) \rightarrow \text{if } x \equiv \textbf{null}\, x \leftarrow \textbf{new}\, \texttt{Name}; a(y); x.\textbf{name} \leftarrow y$$

$$c(x) \rightarrow x \leftarrow \textbf{null}$$

$$c(x) \rightarrow b(x)$$

with start symbol c. For the second element of the sequence we apply rule 9.45 on page 167, rule 9.47 on page 167, rule 9.45 on page 167 and finally rule 9.44 on page 167 to get:

$$
\begin{aligned}
&d(x) \rightarrow x \leftarrow \text{``''}\\
&d(x) \rightarrow y \leftarrow c; d(x); x \leftarrow y \| x \quad \text{for every character } c\\
&e(x) \rightarrow \text{if } x \equiv \textbf{null}\, x \leftarrow \textbf{new}\, \texttt{OpsName}; e(y); x.\textbf{name} \leftarrow y\\
&f(x) \rightarrow x \leftarrow [],\\
&f(x) \rightarrow e(y); f(x); x \leftarrow x \| y\\
&g(x) \rightarrow \text{if } x \equiv \textbf{null}\, x \leftarrow \textbf{new}\, \texttt{Ops}; f(y); x.\textbf{ops} \leftarrow y
\end{aligned}
$$

with start symbol g. Putting the two together with rule 9.50 on page 167 requires only one additional nonterminal,

$$
\begin{aligned}
h(x) &\rightarrow c(x)\\
h(x) &\rightarrow g(x)
\end{aligned}
$$

giving us the entire grammar.

9.3 Client verification

To demonstrate the value of my approach, I performed two distinct classes of experimentation. The first, the client verification I detail here, involves verification of a demonstration client for the Amazon E-Commerce Service (AWS-ECS). We generate a stub for the SOAP communication layer so that we can verify the client without connecting to the AWS-ECS and without any network communication. The second, the server verification, involves connecting directly to AWS-ECS itself and checking the AWS-ECS implementation, which I detail in section 9.4 on page 160.

9.3.1 Amazon Web Service client

The AWS-ECS client used here in my experiments is a demonstration of programming technique written by Amazon. It is called the AWS Java Sample. This client performs no validation on its input data whatsoever. It is intended as a programming example showing how to use the SOAP and REST interfaces, not as something to use. Hence, it serves as a suitable vehicle for us in demonstrating the bug finding capabilities of our approach.

The client consists of a Swing GUI that serves as a thinly veiled interface to the AWS-ECS methods. To verify this client, I wish to use JPF, which cannot handle GUIs; accordingly I have written by hand a simple driver that explores several areas that I want to test. I classify these areas into two major groupings: input errors that the client ought to be catching but doesn't, and control flow errors that represent execution sequences that are locally valid but globally wrong. An example of the former is passing a string when AWS expects an integer; an example of the latter is trying to modify contents of the cart after the cart has been cleared. A proper AWS client should catch these errors and prompt the user to provide appropriate input instead of passing erroneous requests to AWS. If such input validation is not done at the client side, the user would either see a cryptic error message sent back from the AWS, or, worse yet, the client may terminate the session (or even crash) based on the error message sent by AWS. By providing erroneous user input and control sequences to the client, our goal is to discover input and control flow validation errors at the client side.

9.3.2 Input errors

To begin, I analyze three input errors that the client, were it doing proper input validation, would catch. In actual execution, one of these (the typecheck failure) would be caught by the Axis communication layer, but the other two would be communicated to AWS, which would refuse to execute them. The data I gathered for these three errors is summarized in table 9.1 on the next page.

Type	Time (s)	Memory (kB)
Typechecking failure	12.5	25,208
Nonsensical data	11.1	25,208
Uncorrelated data	20.8	43,360

Table 9.1: Input validation errors for the Client.

The typechecking failure here exploits a failure of the client to check that the XML Schema type of some of its inputs is actually valid. This comes up when a user enters a string, when an integer is expected. This does not get caught by the Java compiler, as the data in question remains a string until it reaches the serialization layer in the code. In actual execution this would be caught by Axis.

The nonsensical data failure here attempts to add a nonexistent item to a nonexistent cart with a bad checksum. Since this is a syntactially valid request, it would make it all the way to Amazon's servers before being rejected, whereas my stub catches it much earlier.

Finally, the uncorrelated data failure here involves two method calls. The method calls are in the correct sequence and would constitute a valid sequence, except for the fact that the data associated with the two calls is completely uncorrelated. Specifically, the test searches for an item and then attempts to add another, nonexistent, item to the cart. This again would ordinarily make its way all the way to Amazon. Catching this error requires the data extensions mentioned earlier.

In table 9.1 on the preceding page I show the results of client verification using the JPF model checker and the server server stub that is automatically generated by our interface compiler. Note that executing a verification task like this one without the automatically generated server stub is almost impossible with the existing model checking tools like JPF. Without the server stub, the client has to send a SOAP request to Amazon's servers, which must execute its implementation of the corresponding operation and send the result back.

In my framework, the calls to the AWS-ECS are replaced with calls to the server stub. When the server stub receives an incoming call, it executes the semantic predicates and actions that correspond to the operation that is called and returns the result.

9.3.3 Control flow errors

The preceding errors are useful, but do not demonstrate the full scope of my approach. Accordingly, I have coded the client driver to call AWS methods in an undirected fashion. The driver first initializes the cart, and then proceeds to call available methods in no particular order. Some call sequences can be perfectly valid executions, but some of them can represent errors. For example, modifying the contents of the cart after clearing it is nonsensical. An appropriate AWS client would detect such errors by storing some state information at the client side and warning the user against such erroneous requests.

Type	Depth	Time (s)	Memory (kB)	Errors found
To first error	2	31.8	43,360	0
To first error	3	64.2	62,084	1
To first error	4	49.6	73,456	1
To first error	5	57.3	73,456	1
All errors	2	31.8	43,360	0
All errors	3	77.3	62,084	2
All errors	4	266.8	111,816	15
All errors	5	862.6	229,872	68

Table 9.2: Control flow validation errors for the Client.

In my framework, these types of errors are caught during client verification since these errors trigger semantic predicate violations in the interface grammar.

The data I gathered from these runs is summarized in table 9.2. Some important details concerning these: I have run each test twice, once until JPF detected the first erroneous path, and then once more discovering all possible erroneous paths. With a depth of 2 my driver is incapable of going wrong; accordingly the first error and all errors data for that depth are the same. At a depth of 3, my analysis takes longer to detect the error than it does for a depth of 4; this is because the first paths our driver executed at depth 3 were in fact correct, and some backtracking had to occur before an error could be detected. By contrast, the timing data for runs detecting all errors follow the exponential time increase one would expect.

9.4 Server verification

For server verification, the interface compiler takes the interface specification as input and automatically generates a driver that sends SOAP requests to the web service. This driver is essentially a sentence generator for the input interface grammar. This work was done in collaboration with Muath Alkhalaf; in particular figure 9.6 on page 168 and figure 9.7 on page 168 were obtained as a part of his MS project.

The basic sentence generator algorithm is the same algorithm used for all interface grammars, as in algorithm 1 on page 52. Here $control =$ **component**, because we are using a strict driver for server verification. This becomes a top-down sentence generation algorithm that starts with the start symbol and generates a leftmost derivation by applying a production rule to the non-terminal symbol at the top of the stack until the stack is empty. Note that this algorithm generates the sentences on-the-fly, i.e., while generating a sentence the algorithm is also executing the corresponding test case by making calls to the target web service. The key step here is always in choosing the next production. Here I discuss only random sentence generation.

9.4.1 Server verification using random sentence generation

A random sentence generator chooses the next production in the sentence generation algorithm randomly. In order to assess the effectiveness of this random testing approach, we measured the following coverage criteria:

- *Non-terminal coverage:* If we generate sentences randomly, how many sentences do we need to generate in order to cover all the non-terminals and how long does that take? Note that, 100% non-terminal coverage is achieved when all the non-terminals appear in derivation of some sentence that has been generated so far.

- *Production coverage:* If we generate sentences randomly, how many sentences do we need to generate in order to cover all the productions and how long does that take? Note that, 100% production coverage is achieved when all the productions are used in some sentence that has been generated so far.

In order to do these measurements we ran ten tests. In each of these tests, we ran the generator until it generated 100 sentences, and then did the measurements. Finally, we took the averages of these ten measurements. Figure 9.6 on page 168 and figure 9.7 on page 168 show the results of our experiments. Not surprisingly, in figure 9.6 on page 168, we see that the verification time increases linearly with the number of sentences. In figure 9.7 on page 168 we see that the full production and nonterminal coverage is achieved after generating 41 sentences on the average. Generating 41 sentences and executing the corresponding 41 test sequences takes about 127 seconds. Again, the generation of the sentences and the execution of the test sequences are done at the same time. The average number of steps in derivations generated by the random generator was 17.5, and the average number of SOAP requests that were generated per derivation was 3.2.

9.4.2 Errors in the Amazon Web Service

During the server verification experiments we discovered two errors. These errors correspond to mismatches between the interface grammar specification and the AWS-ECS implementation.

Multiple add error

My initial reading of the AWS-ECS specification led me to believe that it was okay to send multiple ADD requests for the same ASIN. I believed that this would lead to multiple lines in the cart with distinct item IDs, but not otherwise cause trouble. Accordingly, the guards $[\![\text{asin} \notin \mathrm{ran}(\text{cart})]\!]$ in production 9.33 on page 166 and production 9.39 on page 166 were omitted. I learned that this was incorrect when we discovered an assertion violation during server verification, and added those guards. This restriction is not explicitly stated in the AWS-ECS API specification.

Null cart array

The driver checks that the contents of the cart returned by Amazon are precisely those we expect to see. In the AWS-ECS, the items in a cart are stored in a sequence of sequences, which is mapped by the Java layer to a field `cartItems` on the `Cart`. This is an instance of the `CartItems` type, which in its field `cartItem` contains an array of `CartItem` objects. I believed that an empty cart, that is a cart with no items, would have a non null `Cart.cartItems` that contains an array of zero length. However in the AWS-ECS implementation, this is translated as a null `Cart.cartItems`, so I was forced to change our semantic predicate accordingly. This is not explicitly stated in the API documentation either, although it is present in the WSDL specification. Although this error can in principle occur anywhere, we encountered it specifically in production 9.29 on page 166 (which deletes an item from the cart, possibly resulting in an empty cart) and production 9.35 on page 166 (which clears the cart).

The experiments I report in the previous section were conducted after we changed the predicates mentioned above to fix these two errors. We also conducted experiments in the faulty versions where above errors were present. We ran the random sentence generator ten times, stopping each time as soon as we discovered the bug, and took the averages of both time and number of sentences. For the assertion violation (the multiple add error) the error was discovered after an average of 3.6 runs and took 2.5 seconds. For the null cart array error it took on average 5.2 runs and 10.1 seconds to discover the error.

The errors mentioned above can either be considered an error in the AWS-ECS specification or an error in the AWS-ECS implementation. Eventually the goal of both client and server verification is to catch the semantic mismatches between the client's and server's understanding of the web service interface specification. In my approach this interface specification is the interface grammar specification. During client verification I look for mismatches between the interface grammar specification and client implementation and during server verification I look for mismatches between the interface grammar specification and the server implementation. As our results demonstrate, this approach is effective in identifying both types of mismatches.

$$
\begin{aligned}
\textit{start} &\rightarrow \textit{search}(\text{asin}); \textit{cart}(\text{asin}) && (9.1)\\
&\mid \epsilon && (9.2)\\
\textit{search}(\text{asin}) &\rightarrow !\textsc{search}(); \text{¡}\textsc{search}(\text{asin}); \textit{search}'(\text{asin}) && (9.3)\\
\textit{search}'(\text{asin}) &\rightarrow !\textsc{search}(); \text{¡}\textsc{search}(\text{asin}); \textit{search}'(\text{asin}) && (9.4)\\
&\mid \epsilon && (9.5)\\
\textit{cart}(\text{asin}) &\rightarrow !\textsc{create}(\text{asin}); \text{¡}\textsc{create}(\text{cart}, \text{item}); && (9.6)\\
&\qquad \textit{permute}(\text{cart}, \text{item}); \textit{clear}(\text{cart}) &&\\
&\mid \epsilon && (9.7)\\
\textit{permute}(\text{cart}, \text{item}) &\rightarrow !\textsc{get}(\text{cart}); \text{¡}\textsc{get}(\text{cart}); \textit{permute}(\text{cart}, \text{item}) && (9.8)\\
&\mid \langle\!\langle \textsc{choose}\, n > 0 \rangle\!\rangle; !\textsc{modify}(\text{cart}, \text{item}, n); && (9.9)\\
&\qquad \text{¡}\textsc{modify}(\text{cart}); \textit{permute}(\text{cart}, \text{item}) &&\\
&\mid !\textsc{modify}(\text{cart}, \text{item}, 0); \text{¡}\textsc{modify}(\text{cart}) && (9.10)\\
&\mid \textit{search}(\text{asin}); \textit{permute}'(\text{cart}, \text{asin}); && (9.11)\\
&\qquad \textit{permute}(\text{cart}, \text{item}) &&\\
&\mid \epsilon && (9.12)\\
\textit{permute}'(\text{cart}, \text{asin}) &\rightarrow [\![\text{asin} \notin \mathrm{ran}(\text{cart})]\!]; !\textsc{add}(\text{cart}, \text{asin}); && (9.13)\\
&\qquad \text{¡}\textsc{add}(\text{cart}, \text{item}); \textit{permute}(\text{cart}, \text{item}) &&\\
&\mid \epsilon && (9.14)\\
\textit{clear}(\text{cart}) &\rightarrow !\textsc{clear}(\text{cart}); \text{¡}\textsc{clear}(\text{cart}); \textit{clear}(\text{cart}) && (9.15)\\
&\mid !\textsc{get}(\text{cart}); \text{¡}\textsc{get}(\text{cart}); \textit{clear}(\text{cart}) && (9.16)\\
&\mid \textit{search}(\text{asin}); \textit{clear}'(\text{cart}, \text{asin}) && (9.17)\\
&\mid \epsilon && (9.18)\\
\textit{clear}'(\text{cart}, \text{asin}) &\rightarrow [\![\text{asin} \notin \mathrm{ran}(\text{cart})]\!]; !\textsc{add}(\text{cart}, \text{asin}); && (9.19)\\
&\qquad \text{¡}\textsc{add}(\text{cart}, \text{item}); \textit{permute}(\text{cart}, \text{item}); &&\\
&\qquad \textit{clear}(\text{cart}) &&\\
&\mid \epsilon && (9.20)
\end{aligned}
$$

Figure 9.3: Interface grammar for a AWS-ECS client

$$\begin{aligned}
\textit{start} &\rightarrow \textit{search}(\text{asin}); \textit{cart}(\text{asin}) && (9.21)\\
&\mid\ \epsilon && (9.22)\\
\textit{search}(\text{asin}) &\rightarrow ?\textsc{search}(); \langle\!\langle \text{asin} \leftarrow \textsc{newtag} \rangle\!\rangle; ¿\textsc{search}(\text{asin});\\
&\qquad\quad \textit{search}'(\text{asin}) && (9.23)\\
\textit{search}'(\text{asin}) &\rightarrow ?\textsc{search}(); \langle\!\langle \text{asin} \leftarrow \textsc{newtag} \rangle\!\rangle; ¿\textsc{search}(\text{asin});\\
&\qquad\quad \textit{search}'(\text{asin}) && (9.24)\\
&\mid\ \epsilon && (9.25)\\
\textit{cart}(\text{asin}) &\rightarrow ?\textsc{create}(\text{asin}); \langle\!\langle \text{item} \leftarrow \textsc{newtag} \rangle\!\rangle; && (9.26)\\
&\qquad\quad \langle\!\langle \text{cart} \leftarrow \{\text{item} \mapsto \text{asin}\} \rangle\!\rangle; ¿\textsc{create}(\text{cart}, \text{item});\\
&\qquad\quad \textit{permute}(\text{cart}, \text{item}); \textit{clear}(\text{cart})\\
&\mid\ \epsilon && (9.27)\\
\textit{permute}(\text{cart}, \text{item}) &\rightarrow ?\textsc{get}(\text{cart}); ¿\textsc{get}(\text{cart}); \textit{permute}(\text{cart}, \text{item}) && (9.28)\\
&\mid\ ?\textsc{modify}(\text{cart}, \text{item}, 0); [\![\text{item} \in \mathrm{dom}(\text{cart})]\!]; && (9.29)\\
&\qquad\quad \langle\!\langle \text{cart} \leftarrow \text{cart} \setminus \{\text{item} \mapsto \top\} \rangle\!\rangle; ¿\textsc{modify}(\text{cart})\\
&\mid\ ?\textsc{modify}(\text{cart}, \text{item}, n); [\![n > 0 \wedge \text{item} \in \mathrm{dom}(\text{cart})]\!]; && (9.30)\\
&\qquad\quad ¿\textsc{modify}(\text{cart})\\
&\mid\ \textit{search}(\text{asin}); \textit{permute}'(\text{cart}, \text{asin}); \textit{permute}(\text{cart}, \text{item}) && (9.31)\\
&\mid\ \epsilon && (9.32)\\
\textit{permute}'(\text{cart}, \text{asin}) &\rightarrow ?\textsc{add}(\text{cart}, \text{asin}); [\![\text{asin} \notin \mathrm{ran}(\text{cart})]\!]; && (9.33)\\
&\qquad \langle\!\langle \text{item} \leftarrow \textsc{newtag} \rangle\!\rangle; \langle\!\langle \text{cart} \leftarrow \text{cart} \cup \{\text{item} \mapsto \text{asin}\} \rangle\!\rangle;\\
&\qquad ¿\textsc{add}(\text{cart}, \text{item}); \textit{permute}(\text{cart}, \text{item})\\
&\mid\ \epsilon && (9.34)\\
\textit{clear}(\text{cart}) &\rightarrow ?\textsc{clear}(\text{cart}); \langle\!\langle \text{cart} \leftarrow \emptyset \rangle\!\rangle; ¿\textsc{clear}(\text{cart}); \textit{clear}(\text{cart}) && (9.35)\\
&\mid\ ?\textsc{get}(\text{cart}); ¿\textsc{get}(\text{cart}); \textit{clear}(\text{cart}) && (9.36)\\
&\mid\ \textit{search}(\text{asin}); \textit{clear}'(\text{cart}, \text{asin}) && (9.37)\\
&\mid\ \epsilon && (9.38)\\
\textit{clear}'(\text{cart}, \text{asin}) &\rightarrow ?\textsc{add}(\text{cart}, \text{asin}); [\![\text{asin} \notin \mathrm{ran}(\text{cart})]\!]; && (9.39)\\
&\qquad \langle\!\langle \text{item} \leftarrow \textsc{newtag} \rangle\!\rangle; \langle\!\langle \text{cart} \leftarrow \text{cart} \cup \{\text{item} \mapsto \text{asin}\} \rangle\!\rangle;\\
&\qquad ¿\textsc{add}(\text{cart}, \text{item}); \textit{permute}(\text{cart}, \text{item}); \textit{clear}(\text{cart})\\
&\mid\ \epsilon && (9.40)
\end{aligned}$$

Figure 9.4: Interface grammar for a AWS-ECS server

$$\mathbf{p} : \mathcal{XML} \to \mathbf{Prod}$$
$$\mathbf{r} : \mathcal{XML} \to \mathbf{NT}$$
$$\mathbf{t} : \mathcal{XML} \to \mathbf{Type}$$

$$\mathbf{p}[\![g \to \textbf{boolean}]\!] = \{\mathbf{r}[\![g]\!](x) \to x \leftarrow \textbf{true}, \mathbf{r}[\![g]\!](x) \to x \leftarrow \textbf{false}\} \tag{9.42}$$

$$\mathbf{p}[\![g \to \textbf{natural}]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to x \leftarrow \mathbf{0}, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g]\!](x); x \leftarrow x+1 \end{array}\right\} \tag{9.43}$$

$$\mathbf{p}[\![g \to \textbf{string}]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to x \leftarrow \text{``''}, \\ \mathbf{r}[\![g]\!](x) \to y \leftarrow c; \mathbf{r}[\![g]\!](x); x \leftarrow y \| x \\ \hfill \text{for every character } c \end{array}\right\} \tag{9.44}$$

$$\mathbf{p}[\![g \to t[g']]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to \text{if } x \equiv \textbf{null}\, x \leftarrow \textbf{new}\, \mathbf{t}[\![g]\!]; \\ \hfill \mathbf{r}[\![g']\!](y); x.t \leftarrow y \end{array}\right\} \cup \mathbf{p}[\![g']\!] \tag{9.45}$$

$$\mathbf{p}[\![g \to g'\{0,1\}]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to x \leftarrow \textbf{null}, \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](x), \end{array}\right\} \cup \mathbf{p}[\![g']\!] \tag{9.46}$$

$$\mathbf{p}[\![g \to g'\{0,\infty\}]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to x \leftarrow [], \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y); \mathbf{r}[\![g]\!](x); x \leftarrow x \| y \end{array}\right\} \cup \mathbf{p}[\![g']\!] \tag{9.47}$$

$$\mathbf{p}[\![g \to g'\{0,n\}]\!] = \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to x \leftarrow [], \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y); x \leftarrow [y], \\ \qquad \cdots \\ \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y_1); \ldots; \mathbf{r}[\![g']\!](y_n); \\ \hfill x \leftarrow [y_1, \ldots, y_n] \end{array}\right\} \cup \mathbf{p}[\![g']\!] \tag{9.48}$$

$$\begin{aligned} \mathbf{p}[\![g \to g'\{m,n\}]\!] = & \left\{\begin{array}{l} \mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g']\!](y_1); \ldots; \mathbf{r}[\![g']\!](y_m); \\ \hfill \mathbf{r}[\![g'']\!](x); x \leftarrow [y_1, \ldots, y_m] \| x \end{array}\right\} \\ & \cup \mathbf{p}[\![g'']\!] \cup \mathbf{p}[\![g']\!] \qquad \text{where } g'' \to g'\{0, n-m\} \end{aligned} \tag{9.49}$$

$$\mathbf{p}[\![g \to g_1, \ldots, g_k]\!] = \{\mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g_1]\!](x); \ldots; \mathbf{r}[\![g_k]\!](x)\} \cup \bigcup_{i=1}^{k} \mathbf{p}[\![g_i]\!] \tag{9.50}$$

$$\mathbf{p}[\![g \to g_1 | \ldots | g_k]\!] = \bigcup_{i=1}^{k} \{\mathbf{r}[\![g]\!](x) \to \mathbf{r}[\![g_i]\!](x)\} \cup \mathbf{p}[\![g_i]\!] \tag{9.51}$$

Here for a nonterminal g, $\mathbf{p}[\![g]\!]$ is the set of associated grammar rules, $\mathbf{r}[\![g]\!]$ is a unique name suitable for a grammar nonterminal, and $\mathbf{t}[\![g]\!]$ is the unique Java type for that position in the XML Schema grammar.

Figure 9.5: MSL to interface grammar translation rules

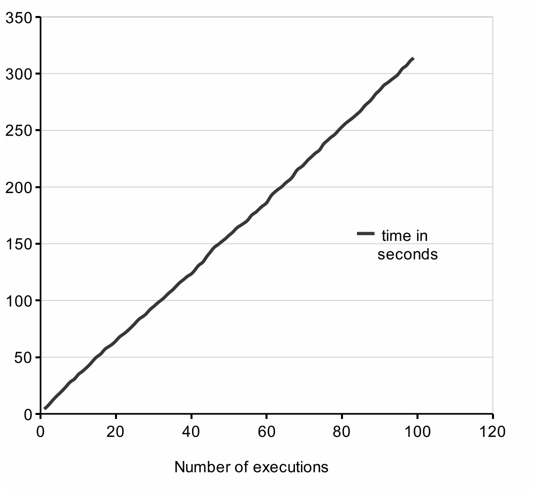


Figure 9.6: The amount of time it takes to test Amazon's AWS-ECS implementation using the test sequences generated by the random sentence generator

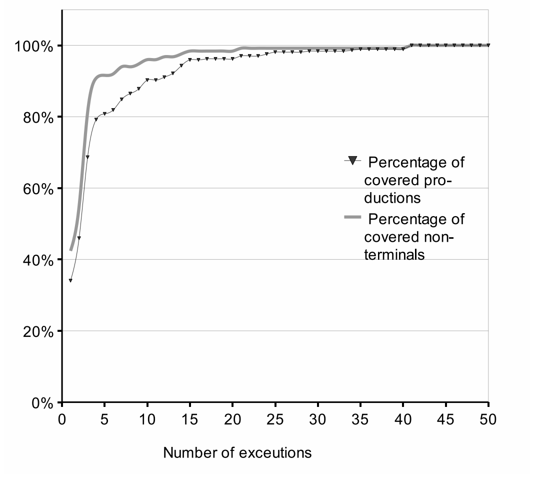


Figure 9.7: Non-terminal and production coverage obtained using the random sentence generator

Chapter 10

Related Work

My work touches on several related fields; some related work is presented here, along with the differences between it any my own efforts.

10.1 Grammar-based testing

There has been some prior work relating to grammar based testing, although none of it attempts to model components of a program with a grammar. Purdom [44] presented a fast algorithm for generating the minimal set of test cases required to achieve production coverage of a grammar, targetting parsers specifically. Because my grammars are interactive, this algorithm cannot be directly applied, but in future work I intend to adapt it.

Lämmel and Schulte [33] describe several techniques for limiting the combinatorial explosion of grammar based testing. Their technique uses a grammar to generate test cases; my work uses a grammar to act as an interactive component of a system. The idea of exploring different coverage criteria, and in particular of adapting their combinatorial coverage technique is appealing but can be difficult due to the interactive nature of my stubs; one cannot achieve even full production coverage (which they call rule coverage) if the host program does not cooperate.

Maurer [37, 38] also generates test data with an enhanced context free grammar for his DGL tool. The same differences as with Lämmel and Schulte's work apply; my tool generates interactive stubs, Maurer's generates test data. Maurer's tool also permits variables, which is an interesting precursor to our rule parameters; however he does not attempt to preserve the lexcial scoping of his variables, as we must with ours.

Offutt, Ammann and Liu [43] describe how mutation testing can be regarded as a type of grammar based testing, and give several useful coverage algorithms. I do not consider mutation at this time, and in any case their technique concentrates on test cases whereas mine is interactive.

Bauer and Finger [7] generate test cases using a regular grammar, which is strictly less powerful than mine and cannot accomodate recursion. As well, their technique generates noninteractive test cases.

Duncan and Hutchison [17] use attributed grammars to generate test cases. There are similarities in their attributed grammars and my interface grammars; for example, we both permit run-time guards, and in many cases their inherited and synthesized attributes can be made equivalent to my rule parameters. However their technique remains focused on test case generation and mine on interactive stubs.

Sirer and Bershad [46] have developed a grammar based test tool *lava,* with a focus on validating their Java Virtual Machine. Their tool has two different roles, one as a straightforward test case generator and another that makes minute permutations in an attempt to discern hidden flaws. As befits a test case generator, they include certificates to solve the oracle problem, which describe the intended result of the test case. The same interactive-versus-generative considerations above apply, and I can mimic their certificates using semantic predicates; nonetheless in future work I intend to include some form of test certificate in my tool.

10.2 Automated environment generation

There has been some work on automated environment generation, in contrast to my own semiautomated style. In [53], Tkachuk, Dwyer, et al., describe a way to derive the driving system automatically. In [31], Khurshid, Păsăreanu, et al., describe a technique for using JPF to generate test cases. In [63], Xie, Marinov, et al., describe a similar technique using exhaustive runs for the purpose of generating test cases. These techniques are very interesting but in their current form they would not have been useful in my case studies because they explore the program only to a bounded depth in the program's call tree. These limitations have led to research in semiautomated environment generation, which can address some of the limitations of a fully automated approach.

10.3 Interface specification

The use of finite state machines for specification, verification and extraction of interfaces have been studied extensively [2, 8, 9, 14, 16, 60]. Finite state machines cannot specify nested-call structures such as the recursive transaction example I have used in this chapter. The interface grammars I propose in this chapter enables us to specify such interactions. Moreover, I believe that the semantic predicates and actions that are allowed in my interface grammars are necessary to model interfaces of complex components. Another factor that differentiates my work from that of Whaley et al. [60] or Alur et al. [2] is that I do not extract interfaces; rather, I use interface grammar specifications to check both interface conformance and also to achieve modular verification.

The Specification Language for Interface Checking (SLIC) is used to specify interface constraints in the SLAM project [5, 6]. In SLIC, interfaces are specified using state machines. The transitions of state machines are associated with C statements that can be used to specify additional constraints on the interface. As with the other state machine based approaches discussed above, the approach used by SLIC is not appropriate for specification of nested call sequences.

In Betin-Can's work [8–10], finite state interface specifications are used to achieve modular verification where behavior verification and interface verification are executed as two separate steps. Interface grammars as proposed here provide a richer language for specification of interfaces and can be integrated to the modular verification approach used in that work.

10.4 Environment generation

Environment generation is a critical problem for achieving modularity in software model checking and has been studied before. Godefroid et al. [22] present techniques for automatically closing environments of open reactive programs by automatically creating the most general environment for the program using dataflow analysis. In contrast, Tkachuk and Dwyer [51] investigate automatically generating environments for components using side effect and points-to analyses for modular model checking. I use a semi-automated approach where the user writes an interface grammar and the interface grammar is automatically compiled to a component stub for modular verification. I believe that for specification of rich interfaces such as the EJB interface discussed in this paper it is necessary to get user input in order to restrict the behaviors allowed by the interface.

The Bandera environment generator [52] from Tkachuk et al. also uses a semi-automated approach in which environment models are automatically synthesized from environment assumptions. The environment assumptions are given as LTL formulas or regular expressions specifying ordering of program actions which are unit method calls or field assignments that can be executed by the environment. My approach based on interface grammars enables us to specify nested call sequences that cannot be expressed using formalisms, such as LTL or regular expressions, that can be recognized by finite state machines. Also rather than focusing on environment generation, I am focusing on specification of interfaces. Of course, these are closely related concepts since the interfaces of components that interact with a program forms the environment of that program. However, I believe that it is more likely for developers to write interface specifications for different components rather than writing an environment for a particular program.

Chapter 11

Conclusion and Future Work

I have defined and written tools for a new formalism that makes automated verification of large systems easier. Furthermore I have applied this formalism to several nontrivial examples, including verification of EJB clients and verification of clients and servers for web services. These investigations have shown that this formalism is effective and useful. My experiments suggest that using interface grammars does not introduce significant overhead into the verification process, and it significantly reduces development time as noted in chapter 8 on page 126. Moreover interface grammars make a useful intermediate target for additional verification techniques, like verifying web services. The parser for these interface grammars is sufficiently flexible that it can be adapted to run in a normal JVM, as shown in 9.4. I feel these results demonstrate that interface grammars represent a useful formalism for environment generation, my goal in this work.

Interface grammars as presented here have some important limitations. The most significant one is the single-threaded nature of the current semantics. Additionally I currently only support a small subset of JML. I suggest these areas and some others for future work. Specifically:

- Concurrency. The current interface grammar system is designed for single-threaded systems. Several key assumptions would need to be revisited before support for concurrent components would work. Yet, highly concurrent systems are the most interesting ones to model check, as they are prone to difficult to find errors.

- Supporting more of JML. The subset I currently support is enough to prove the value of the approach and enough to do useful work (as seen in chapter 7 on page 87) but it would be very useful to support additional JML constructs; for example, the `\fresh` predicate or the existential predicates.

- An interface grammar editor. I wrote a simple Emacs mode for my own use, but an Eclipse plugin would be attractive. Additionally some sort of debugger would have been invaluable during development; I did not have time to look into this myself, but it would make an excellent medium-sized project.

- XML document generation. An XML document is usually defined as a recursive structure; accordingly some grammar-based formalism would seem appropriate.

Additionally DOM and SAX are well defined, standardized interfaces for reading these documents, and so would provide a natural cleavage point for applying interface grammars. Some of this was needed for investigating web services in chapter 9 on page 141.

- Metric support. We already investigated production coverage of an interface grammar in section 9.4 on page 160; additional metrics would be interesting to examine and may require tool support.
- Support for additional model checkers. Bogor [45] would probably warrant this treatment.
- Composition of interface grammars.
- Investigate moving portions of the interface grammar parser into the model checker. JPF is extensible, and it may be helpful to move some of the parser code, which is not specific to any particular grammar and which performs a fair amount of computation, into the code of the model checker itself.
- Investigation of other large systems.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages, (POPL 2005)*, 2005.

[3] Amazon web services. http://solutions.amazonwebservices.com/.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[5] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the SPIN Workshop*, pages 103–122, 2001.

[6] T. Ball and S. K. Rajamani. SLIC: A Specification Language for Interface Checking. Technical Report MSR-TR-2001-21, Microsoft Research, Jan. 2002.

[7] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Proceedings of the 4th International Conference on Software Engineering*, pages 425–432, Munich, Germany, Sept. 1979. ACM Special Interest Group on Software Engineering.

[8] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 248–257, 2004.

[9] A. Betin-Can, T. Bultan, and X. Fu. Design for verification for asynchronously communicating web services. In *Proceedings of the 14th International World Wide Web Conference (WWW 2005)*, pages 750–759, 2005.

[10] A. Betin-Can, T. Bultan, M. Lindvall, S. Topp, and B. Lux. Application of design for verification with concurrency controllers to air traffic control software. In *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE 2005)*, 2005.

[11] A. Brown, M. Fuchs, J. Robie, and P. Wadler. MSL: a model for W3C XML Schema. In *Proceedings of the 10th International World Wide Web Conference*,

pages 191–200. ACM Special Interest Group on Hypertext, Hypermedia and Web, ACM, 2001.

[12] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80, pages 73–89, Elsevier, June 2003.

[13] S. K. C. Boyapati and D. Marinov. Korat: Automated testing based on java predicates. In *Proc. ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, 2002.

[14] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 428–441, 2002.

[15] O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematicas Structures in Computer Science*, 4(2):361–391, 1992.

[16] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[17] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering*, pages 170–178, New York, NY, USA, Mar. 1981. ACM Special Interest Group on Software Engineering.

[18] M. Fowler. *Analysis Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[19] P. Fradet and D. le Métayer. Shape types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, Jan. 1997. ACM SIGPLAN and ACM SIGACT, ACM Press.

[20] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Jan. 1997.

[21] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–186. ACM Press, 1997.

[22] P. Godefroid, C. Colby, and L. Jagadeesan. Automatically closing open reactive programs. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 1998)*, pages 345–357, 1998.

[23] R. Griswold and M. Griswold. *The Icon Programming Language*. Prentice-Hall, New Jersey, NJ, 1983.

[24] D. A. Hall. jga: Generic algorithms for Java.

[25] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. White. Formal analysis of the remote agent before and after flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[26] G. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[27] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming language design and implementation*, Dec. 2004.

[28] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pages 39–49, 2007.

[29] G. Hughes, S. P. Rajan, T. Sidle, and K. Swenson. Error detection in concurrent java programs. In *Proceedings of the Workshop on Software Model Checking (SoftMC 2005)*, volume 144, pages 45–58. Electronic Notes in Theoretical Computer Science, Feb. 2006. Issue 3.

[30] A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*, chapter 6. The Pragmatic Programmers, Sept. 2003.

[31] S. Khurshid, C. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Warsaw, Poland, Apr. 2003.

[32] H. Kilov, B. Rumpe, and I. Simmonds, editors. *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[33] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In U. Uyar, M. Fecko, and A. Duale, editors, *Proceedings of the 18th IFIP International Conference on Testing Communicating Systems (TestCom 2006)*, volume 3964 of *LNCS*, New York, NY, USA, May 2006. Springer-Verlag.

[34] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[35] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, May 2008.

[36] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, 1992.

[37] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Softw.*, 7(4):50–55, 1990.

[38] P. M. Maurer. The design and implementation of a grammar-based data generator. *Software Practice and Experience*, 22(3):223–244, Mar. 1992.

[39] B. Meyer. Applying design by contract. *IEEE Computer*, pages 40–51, Oct. 1992.

[40] A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. Technical report, PUC-Rio, 2004.

[41] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[42] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[43] J. Offutt, P. E. Ammann, and L. L. Liu. Mutation testing implements grammar-based testing. In *Proceedings of the 2nd Workshop on Mutation Analysis*, Nov. 2006.

[44] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, 1972.

[45] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th*

European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, pages 267–276, New York, NY, USA, 2003. ACM Press.

[46] E. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL '99)*, pages 1–13, Austin, TX, US, 1999.

[47] J. M. Siskind and D. A. McAllester. SCREAMER: A portable efficient implementation of nondeterministic COMMON LISP. Technical Report IRCS–93–03, University of Pennsylvania Institute for Research in Cognitive Science, 1993.

[48] Simple object access protocol (soap) 1.1. W3C Note 08, http://www.w3.org/TR/SOAP/, May 2000.

[49] Sun Java Community Process. *JavaBeans Specification*, 1997. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[50] Sun Java Community Process. *Enterprise Java Beans 3.0 Specification*, May 2006. JSR-000220.

[51] O. Tkachuk and M. B. Dwyer. Adapting side-effects analysis for modular program model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE)*, pages 188–197, 2003.

[52] O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking. In *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, pages 116–129, 2003.

[53] O. Tkachuk, M. B. Dwyer, and C. S. Păsăreanu. Automated environment generation for software model checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Oct. 2003.

[54] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder—second generation of a Java model checker. In *Proc. of Post-CAV Workshop on Advances in Verification*, Chicago, July 2000.

[55] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[56] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[57] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of International Symp. on Software Testing*, 2004.

[58] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.

[59] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–Sept. 1998.

[60] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, 2002.

[61] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.

[62] Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[63] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005*, 2005.

[64] Extensible markup language (XML) 1.0 (second edition). W3C, http://www.w3.org/TR/REC-xml, 2000.

[65] XML Schema part 2: Datatypes. W3C Recommendation, http://www.w3.org/TR/xmlschema-2/, May 2001.

[66] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, 2004.