Interface Grammars for Modular Software Model Checking*

Graham Hughes and Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{graham,bultan}@cs.ucsb.edu

ABSTRACT

We propose an interface specification language based on grammars for modular software model checking. In our interface specification language, component interfaces are specified as context free grammars. An interface grammar for a component specifies the sequences of method invocations that are allowed by that component. Using interface grammars one can specify nested call sequences that cannot be specified using interface specification formalisms that rely on finite state machines. Moreover, our interface grammars allow specification of semantic predicates and actions, which are Java code segments that can be used to express additional interface constraints. We have built an interface compiler that takes the interface grammar for a component as input and generates a stub for that component. The resulting stub is a table-driven parser generated from the input interface grammar. Invocation of a method within the component becomes the lookahead symbol for the stub/parser. The stub/parser uses a parser stack, the lookahead, and a parse table to guide the parsing. The semantic predicates and semantic actions that appear in the right hand sides of the production rules are executed when they appear at the top of the stack. We conducted a case study by writing an interface grammar for the Enterprise JavaBeans (EJB) persistence interface. Using our interface compiler we automatically generated an EJB stub using the EJB interface grammar. We used the JPF model checker to check EJB clients using this automatically generated EJB stub. Our results show that EJB clients can be verified efficiently using our approach.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]

*This work is supported by NSF grants CCF-0614002 and CCF-0341365.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'07, July 9–12, 2007, London, England, United Kingdom.
Copyright 2007 ACM 978-1-59593-734-6/07/0007 ...$5.00.

Keywords

interface grammars, modular verification, model checking

General Terms

Verification

1. INTRODUCTION

The application of model checking techniques directly to programs [11, 16, 4] has shown promise for specific verification tasks, such as checking for concurrency errors [16] or checking device drivers for interface violations [4]. However, there are two related problems that hinder applicability of model checking to software in a wider scale: 1) state space explosion (i.e., exponential increase in the search space by increasing number of variables and concurrent components) limits the scalability of model checking techniques and 2) environment generation (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.

Our focus in this paper is model checking Java programs. The limitations we mentioned above are apparent in Java Path Finder (JPF) [16], which is a model checker for Java programs. JPF cannot handle native calls in Java programs. Hence, in order to use JPF for verification of Java programs, one has to write environment models for any component that uses native code, which is a daunting task.

Note that inability to handle native code is not only a limitation specific to JPF, but it is 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. 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

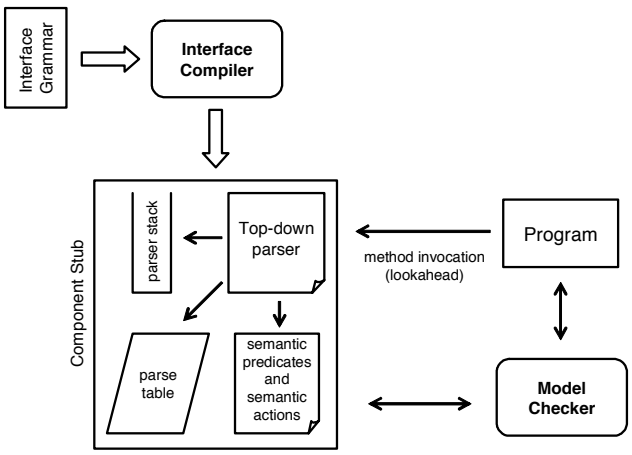

Figure 1: An overview of our approach

(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.

In this paper, we propose a semi-automated approach to attack the above mentioned problems. We propose an interface specification language and require the users to write interface specifications for components that are outside the scope of the current verification effort. Our 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, our interface specification language allows the users to escape to Java and write semantic predicates or actions in Java, specifying the behavior of the component (similar to the approach used by parser generators such as Yacc [13]). We believe that our 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.

Figure 1 shows an overview of our approach. We have built an *interface compiler* that takes an interface grammar as input and automatically generates a stub for the corresponding component. The component stub is a table-driven top-down parser that parses the sequence of incoming method calls (i.e., the method invocations) based on the grammar provided in the interface specification. During execution, the stub/parser executes the semantic predicates and actions at the appropriate times based on their placement in the productions of the interface grammar. If the program that uses the component violates the component's interface, then the component stub either reports a parse error (that corresponds to an error in the call sequence) or a semantic predicate violation (which can correspond to an error in an argument that is passed to the component).

In order to write compact interfaces it is necessary to support nondeterminism in an interface specification language. A stub generated from an interface specification of a component should generate an over-approximation of the behavior of that component. The use of nondeterminism allows specification of a set of behaviors in a concise manner. Our interface specification language provides a nondeterministic switch operator, i.e., two or more switch cases evaluate to true, then one of them is selected nondeterministically.

We assume that the target software model checker provides a *nondeterministic choice* primitive. Our current interface compiler uses the nondeterministic choice primitives provided by JPF. We can easily modify our interface compiler to support other model checkers as long they support nondeterministic choice. During verification, the model checker exhaustively checks all possible choices that can result from the use of nondeterministic choice primitives. While generating the stub code, our interface compiler converts the nondeterministic choices in the interface grammar to calls to the nondeterministic choice primitive of the model checker. This means that all possible behaviors provided by the interface will be checked by the model checker during verification of the program with the automatically generated stub.

Our approach enables model checking to be executed in a modular fashion by replacing different components in the software system with environment models generated from their interfaces. We conducted a case study to demonstrate our approach using Enterprise JavaBeans (EJB) Persistence API clients. We wrote an interface grammar for the Persistence API and verified Persistence API clients using a stub automatically generated from this interface. Our experimental results demonstrate that interface grammars can be used effectively in modular verification.

The rest of the paper is organized as follows. Section 2 provides an overview of our interface specification language. Section 3 discusses the interface compiler. Section 4 discusses the EJB case study. Section 5 includes a discussion on related work, and Section 6 concludes the paper.

2. INTERFACE GRAMMARS

We propose interface grammars as a language for specification of component interfaces. The core of an interface grammar is a set of production rules that define a Context Free Grammar (CFG). This CFG 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 CFG 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:

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

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 has to 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 we gave above does not require the power of context free 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:

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

Note that this interface specification allows nesting of matching `begin` and `commit` or `rollback` calls and, hence, 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 CFG 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).

Figure 2 shows the transaction and recursive transaction classes from the EJB interface specification. In addition to the `begin`, `commit` and `rollback` methods we discussed in the simplified interface grammar examples above, these grammars include another method called `setRollbackOnly` and two query methods `isActive` and `getRollbackOnly`. The method `setRollbackOnly` can only be invoked if there is an active transaction, and after it is invoked, the only way to finish the transaction is to invoke `rollback`. The specification shown in Figure 2 includes one grammar for the `transaction` class and an additional grammar for the `recursive_transaction` class. The nonterminals used in the transaction grammar are `start`, `inactive`, `active`, and `rollback_only` and the nonterminals for the recursive transaction grammar are `start`, `base` and `tail`. By default `start` is the start symbol.

Note that the interface grammars shown in Figure 2 are based on the simple grammars we discussed above. An interesting difference between the transaction and recursive transaction grammars in Figure 2 is the way they handle roll-back-only status. In the transaction grammar, roll-back-only status is handled at the grammar level by using a nonterminal that corresponds the case when roll-back-only is set. In the recursive transaction grammar, roll-back-only status is handled with semantic predicates and semantic actions. Our interface specification language supports both of these approaches. Note that relying on just grammar rules to keep such state information would produce a large number of nonterminals. On the other hand relying only on semantic predicates and actions would degenerate the interface specification into a hand written Java stub.

In Figure 3 we show a (simplified) grammar defining the abstract syntax of our interface grammar language. We denote *nonterminal* and `terminal` symbols and Java CODE and IDENTIFIERS with different fonts. The symbols `<<` and `>>` are used to enclose Java statements and expressions. Incoming method calls to the component (i.e., method invocations) are shown with adding the symbol `?` to the method name as a prefix. Outgoing method calls (i.e., method calls by the component) are shown with adding the symbol `!` to the method name as a prefix. In the grammar shown in Figure 3, we use "*" to denote zero or more repetitions of the preceding symbol, and "?" to denote that the preceding symbol can appear zero or one times.

An interface grammar consists of a set of class interfaces (not to be confused with Java interfaces) (represented in rule (1) in Figure 3). The interface compiler generates one stub class for each class interface. Each class interface consists of a set of semantic actions and a set of production rules that define the CFG for that class (rules (2), (3) and (4)). A *semantic action* is simply a piece of Java code that is inserted to the stub class that is generated for the component (rule (20)). A *rule* corresponds to a production rule in the interface grammar. Each rule has a name and a block (rule (5)). A rule block consists of a sequence of statements (rule (6)). Each statement can be a rule application, a semantic action, a declaration, a choose block, a method invocation, a method return or a method call (rules (7)-(14)). A semantic action corresponds to a piece of Java code that is executed when the parser sees the nonterminal that corresponds to that semantic action at the top of the parse stack. A *rule application* corresponds to the case where a nonterminal appears on the right hand side of a production rule. A *declaration* corresponds to a Java code block where a variable is declared and is assigned a value (rule (21)). A *choose block* is simply a switch statement (rules (11) and (15)). A selector for a switch case can either be a method invocation (i.e., an incoming method call), a semantic predicate or the combination of both (rules (16) and (17)). A switch case is selected if the semantic predicate is true and if the lookahead token matches to the method invocation for that switch case. A *method return* simply corresponds to a return statement in Java. When the component stub receives a method invocation from the program, it first calls the interface parser with the incoming method invocation, which is the lookahaed token for the interface parser. When the parser returns, the component stub calls the interface parser again, this time with the token which corresponds to the method return. Finally, a *method call* is simply a call to another method by the stub.

```
class transaction
          implements EntityTransaction {
  << entity_manager m; ... >>;
  rule start { apply inactive; }
  rule inactive {
    choose {
      case ?begin(): {
        !<< m >>.begin();
        return begin; apply active;
      }
      case ?isActive(): {
        return isActive << false >>;
        apply active;
      }
      case ?getRollbackOnly(): {
        return getRollbackOnly << false >>;
        apply active;
      }
      case : { }
    }
  }
  rule active {
    choose {
      case ?commit(): {
        !<< m >>.commit();
        return commit; apply inactive;
      }
      case ?commit(): {
        !<< m >>.rollback();
        << throw new RollbackException(); >>
      }
      case ?setRollbackOnly(): {
        return setRollbackOnly;
        apply rollback_only;
      }
      case ?isActive(): {
        return isActive << true >>;
        apply active;
      }
      case ?getRollbackOnly(): {
        return getRollbackOnly << false >>;
        apply active;
      }
      case ?rollback(): {
        !<< m >>.rollback(); return rollback;
        apply inactive;
      }
    }
  }
  rule rollback_only {
    choose {
      case ?setRollbackOnly(): {
        return setRollbackOnly;
        apply rollback_only;
      }
      case ?isActive(): {
        return isActive << true >>;
        apply rollback_only;
      }
      case ?getRollbackOnly(): {
        return getRollbackOnly << true >>;
        apply rollback_only;
      }
      case ?rollback(): {
        !<< m >>.rollback(); return rollback;
        apply inactive;
      }
    }
  }
}
```

```
class recursive_transaction
          implements EntityTransaction {
  << ... >> ;
  rule start { apply base; }
  rule base {
    choose {
      case ?begin: {
        << level++; >>;
        !<< m >>.begin();
        return begin;
        apply base;
        apply tail;
        apply base;
      }
      case ?setRollbackOnly(): {
        << isRollbackOnly = true; >>;
        return setRollbackOnly;
        apply base;
      }
      case ?isActive(): {
        return isActive << level > 0 >>;
        apply base;
      }
      case ?getRollbackOnly(): {
        return getRollbackOnly << isRollbackOnly >>;
        apply base;
      }
      case : { }
    }
  }
  rule tail {
    choose {
      case ?commit() << !isRollbackOnly >> : {
        !<< m >>.commit();
        << decrement(); >>;
        return commit;
      }
      case ?commit() << !isRollbackOnly >> : {
        !<< m >>.rollback();
        << decrement(); >>;
        << throw new RollbackException();  >>
      }
      case ?setRollbackOnly(): {
        << isRollbackOnly = true; >>;
        return setRollbackOnly;
        apply tail;
      }
      case ?isActive(): {
        return isActive << true >>;
        apply tail;
      }
      case ?getRollbackOnly(): {
        return getRollbackOnly << isRollbackOnly >>;
        apply tail;
      }
      case ?rollback(): {
        !<< m >>.rollback();
        << decrement(); >>;
        return rollback;
      }
    }
  }
}
```

Figure 2: A portion of the EJB interface grammar that specifies the interface constraints about the transactions and recursive transactions

```
(1)  main        →  class *

(2)  class       →  class CLASSID { item * }

(3)  item        →  semact ;
(4)              |  rule

(5)  rule        →  rule RULEID block

(6)  block       →  { statement * }

(7)  statement   →  block
(8)              |  apply RULEID ;
(9)              |  semact ;
(10)             |  declaration ;
(11)             |  choose { cbody * }
(12)             |  ? MINVOCATION ;
(13)             |  return MRETURN semexpr ? ;
(14)             |  ! MCALL ;

(15) cbody       →  case select ? : { statement * }

(16) select      →  ? MINVOCATION sempred ?
(17)             |  sempred

(18) sempred     →  << EXPR >>

(19) semexpr     →  << EXPR >>

(20) semact      →  << STATEMENT >>

(21) declaration →  << TYPE ID = EXPR >>
```

Figure 3: Abstract syntax for the interface grammar language

3. INTERFACE GRAMMAR COMPILER

We have implemented a compiler for our interface grammars, targeting the Java language. Our interface compiler executes in three major steps:

1. Parse the input interface grammar specification and construct an abstract syntax tree;
2. Convert this abstract syntax tree into a Context Free Grammar (CFG);
3. Output a parser for this resulting CFG.

Our interface compiler generates a stub for each class in the interface specification. At run time, the stub for a class calls the parser that is generated based on the interface grammar of that class, with the method calls it witnesses. Below, we describe the conversion process from interface grammars to context free grammars, generation of parsers for the resulting context free grammars, and the runtime system for the automatically generated parser/stubs.

Our method generates a context free grammar from the interface grammar specification and at runtime the automatically generated stub uses this grammar to parse method invocations. We chose to use a modified LL(1) algorithm [1] as the basis for our parser, both for its familiarity and for its relative ease of implementation. There are a number of different potential ways to parse an LL(1) grammar, but for the purposes of this discussion we will distinguish two; the recursive descent parser and the table driven parser. Both of these approaches have similar efficiency. 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 us). An advantage of using a recursive descent parser in our context is the fact that we can insert the semantic predicates and actions to the methods of the recursive descent parser, whereas for a table driven parser, we must find a way to represent semantic actions or semantic predicates as data. The most important difference for our purposes, however, is where the tokens come from.

We 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. We require the code-calls convention, because we are writing stubs for components that will have their methods invoked by user code. It is very difficult to write a single threaded recursive descent parser using the code-calls convention in Java, 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. We can 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. More importantly this additional concurrency would increase the state space and degrade the performance of the target Java model checker, i.e., it would contribute to the problem that we wish to solve in the first place. Based on these concerns our interface compiler generates table-driven parsers for interface grammars.

3.1 Compile-time computation

The goal of our interface compiler is to translate an interface grammar into a number of Java classes. First, our interface compiler uses the ANTLR tool [3] to parse the input interface specification and construct an abstract syntax tree representing the input specification.

Generating the context free grammar: The second major step of computation in our interface compiler involves converting the abstract syntax tree generated by the parser into a context free grammar. In addition to nonterminal and terminal symbols, the resulting context free grammar also contains semantic predicates and actions. The terminal symbols of the resulting context free grammar are the method invocations and the method returns for each method m in the interface that we are stubbing, and we represent these with the symbols $?m$ and $¿m$, respectively. Note that, $?m$ represents an external client calling m, and $¿m$ represents a return from such a call.

Formally, we define a context free interface grammar G as a tuple

$$G = \langle NT, T, SA, SP, P \rangle$$

where NT is the set of nonterminals, T is the set of terminals, SA is the set of semantic actions, and SP is the set of semantic predicates. $P \subseteq NT \times (NT \cup T \cup SA \cup SP)^*$ is the set of productions where each production consists of one nonterminal symbol (left hand side of the production) and a sequence of nonterminal and terminal symbols and semantic actions and predicates (right hand side of the production).

In Figure 4 we give an attribute grammar based on the abstract syntax of our interface specification language shown in Figure 3. This attribute grammar shows how interfaces written in our interface specification language are converted

to a context free interface grammar. In the attribute grammar shown in Figure 4, for every symbol s, $s.t \in (NT \cup T \cup SA \cup SP)^*$ is a sequence of nonterminal and terminal symbols and semantic actions and predicates, and $s.p \subseteq NT \times (NT \cup T \cup SA \cup SP)^*$ is a set of productions. We use $\|$ to denote concatenation of sequences, $\langle\langle x \rangle\rangle$ to denote a semantic action containing x, and $[\![x]\!]$ to denote a semantic predicate containing x.

Constructing the parse table and computing the *first* and *follow* sets: After we construct a context free interface grammar from the abstract syntax tree of the interface specification using the attribute grammar rules given in Figure 4, the next step is to construct the LL(1) parser table for the resulting context free interface grammar. We need to modify the standard LL(1) parse table construction algorithm due to two reasons:

- We have semantic predicates that can influence the parse, by disallowing certain productions;
- We want to support nondeterministic choice in interface specifications which will be resolved by the target model checker's search heuristics at runtime.

Accordingly, for a given grammar $G = \langle NT, T, SA, SP, P \rangle$ our parsing table is a function $t : NT \times T \to \mathcal{P}(SP \times P)$.

Note that, in normal LL(1) parsing, given a nonterminal (at the top of the parse stack) and a terminal (the lookahead) the parsing table should return a single production; more than one production in one cell of the table indicates that the grammar is not LL(1). We relax this restriction since we allow semantic predicates and since we allow some nondeterminism in the interface specifications with the nondeterministic `choose` construct. When semantic predicates are added, some productions may not be available at runtime since the semantic predicate that is guarding that production may evaluate to false. Accordingly we pair the semantic predicate controlling when a production is available with the production in the parsing table. Finally since we permit some nondeterminism based on the nondeterministic choose operator, we relax the parsing table still further. The parsing tables we construct consist of lists of pairs of semantic predicates and productions. More than one production can be available given the nonterminal at the top of the parser stack and the lookahead token; that is, more than one pair's semantic predicate can evaluate to true. The semantics of that event are discussed in Section 3.2.

To compute the parse table t, we need two auxiliary functions *first* and *follow* [1], which we compute using the algorithms shown in Figures 5 and 6. Because we are dealing with code that writes code, we introduce some conventions to make our presentation simpler:

- $\langle\langle x \rangle\rangle$ means "code that will output x".
- $[\![x]\!]$ is the predicate that, when evaluated, computes x.
- ϵ is the empty list.
- $\square$ is the end of input token.
- $\langle\langle \ldots \$x \ldots \rangle\rangle$ means that x should be substituted into the generated code.

The parse table is constructed using the *first* and *follow* functions based on the standard LL(1) parse table construction algorithm [1], except, as we discussed above, we allow multiple productions to be inserted to a single cell of the parser table. The resulting parse tables is embedded directly in the code we generate. We then generate stubs for each method in the Java interface we are implementing; the details of the stub code will be discussed in Section 3.3.

Closures and scoping: There remain some complications. Our Java escapes here are code, but need to be encoded as data for the runtime parser. We have ignored this so far, using $\langle\langle x \rangle\rangle$ and $[\![x]\!]$. To encode our code as data, we wrap all Java escapes using anonymous inner classes, and refer to these as closures; the code is then simply

```
new Closure () {
    public Object apply () {
        $code
    }
}
```

Predicates can be constructed 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 we must make decisions on. If we were using a recursive descent parser, we could exploit the Java compiler's scoping, but we have dismissed that possibility above; accordingly we have to track it ourselves with our own symbol table.

We have implemented a symbol table at runtime with five important methods. The methods *openscope* and *closescope* open a new scope and close the most recent scope, respectively. The method *bind*(n) introduces n as a new variable in the topmost scope. The method *get*(n) searches the symbol table for the most recently bound n and returns its associated value, and the method *put*(n, o) is similar but sets that associated value.

If we used the variable names as keys, this would result in dynamic scoping, whereas our goal is to implement lexical scoping. Accordingly, we assign to each variable declaration in the program a unique number, and use that as a key. We must also keep track of the declarations that are visible both before and after every Java escape. Given this, we can now alter the body of the Java closure code to be as follows:

```
for declaration ∈ visibleSymbolsBefore do
    ⟨⟨$(declaration.type), $(declaration.name)
      = symbols.get ($(declaration.id));⟩⟩
od
⟨⟨$code⟩⟩
for declaration ∈ visibleSymbolsAfter do
    ⟨⟨symbols.put ($(declaration.id),
                   $(declaration.name));⟩⟩
od
```

We must also introduce opening scopes, closing scopes, and binding into our generated grammar; Figure 4 includes this already.

3.2 Runtime computation

Now that we have produced Java code with a parse table, we must discuss the runtime environment that completes our stub. We can consider each class in isolation since they are independent and contain independent parsing tables.

The core parsing algorithm is given in Figure 7. Each stub we generate contains code that corresponds to an implemen-

$$
\begin{array}{lll}
rule \rightarrow \texttt{rule}\ \text{RULEID}\ block & & rule.p := \{(\text{RULEID}, block.t)\} \cup block.p \\
block \rightarrow \{\ statement^*\ \} & & block.t := \langle\!\langle openscope() \rangle\!\rangle \parallel \Big|\Big|_i statement_i.t \parallel \langle\!\langle closescope() \rangle\!\rangle \\
 & & block.p := \bigcup_i statement_i.p \\
statement \rightarrow block & & statement.t := block.t \\
 & & statement.p := block.p \\
statement \rightarrow \texttt{apply}\ \text{RULEID}\ ; & & statement.t := \text{RULEID} \\
statement \rightarrow semact & & statement.t := \langle\!\langle semact.statement \rangle\!\rangle \\
statement \rightarrow declaration\ ; & & statement.t := \langle\!\langle \mathsf{bind}(declaration.id);\ declaration.id\ \texttt{=}\ declaration.expr\ \texttt{;} \rangle\!\rangle \\
statement \rightarrow \texttt{choose}\ \{\ cbody^*\ \} & & statement.t := statement.id \\
 & & statement.p := \bigcup_i \{(statement.id, cbody_i.t)\} \cup cbody_i.p \\
statement \rightarrow \texttt{?}\ \text{MINVOCATION}\ ; & & statement.t := ?\text{MINVOCATION} \\
statement \rightarrow \texttt{return}\ \text{MRETURN}\ ; & & statement.t := ¿\text{MRETURN} \\
statement \rightarrow \texttt{return}\ \text{MRETURN}\ semexpr; & & statement.t := \langle\!\langle \texttt{result = }\ semexpr.expr \rangle\!\rangle \parallel ¿\text{MRETURN} \\
statement \rightarrow \texttt{!}\ \text{MCALL}; & & statement.t := \langle\!\langle\ \text{MCALL}();\ \rangle\!\rangle \\
cbody \rightarrow \texttt{case}\ select?:\{\ statement^*\ \} & & cbody.t := \text{SELECT}.t \parallel \langle\!\langle openscope() \rangle\!\rangle \parallel \Big|\Big|_i statement_i.t \parallel \langle\!\langle closescope() \rangle\!\rangle \\
 & & cbody.p := \bigcup_i statement_i.p \\
select \rightarrow ?\ \text{MINVOCATION}\ sempred & & select.t := ?\text{MINVOCATION} \parallel [\![sempred.expr]\!] \\
select \rightarrow ?\ \text{MINVOCATION} & & select.t := ?\text{MINVOCATION} \\
select \rightarrow sempred & & select.t := [\![sempred.expr]\!]
\end{array}
$$

Here, $\parallel$ denotes sequence concatenation. Unless otherwise specified, for any symbol s, $s.p = \emptyset$. For *statement*, the attribute *statement.id* is a unique identifier for that statement that can serve as a nonterminal.

Figure 4: Translation from syntax tree to context free grammar

tation of this algorithm. The algorithm shown in Figure 7. is fairly similar to the standard LL(1) table-driven parsing algorithm with the addition of semantic actions and semantic predicates, but the *choose* function merits discussion. In the event that multiple productions are available and legal, we need to choose one of them; here we use the model checker's nondeterministic choice primitive. Also, **fail** causes an exception to be thrown, or an assertion to be violated, as befits the situation.

3.3 Stubbing methods

Armed with this algorithm we can finally discuss the methods to be stubbed out; these become

```
public $returnType(stub)
       $name(stub)($arguments(stub)) {
    arguments = [$arguments(stub)];
    result = null; exception = null;
    parser.witness (?$name(stub));
    try {
        parser.witness (¿$name(stub));
    } catch (Exception e) {
        parser.tossUntil (¿$name(stub));
        exception = e;
    }
    if (exception != null) throw exception;
    return ($returnType(stub)) result;
}
```

```
proc witness(t) ≡
  while stack.top() ≠ t do
        o := stack.pop();
        if o ∈ SP ∧ ¬o.apply() then fail
        elsif o ∈ SA then o.apply();
        else productions := table(o, t);
             viable := {(p, prod :(p, prod) ∈ productions
                                  ∧ p.apply()};
             chosen := choose(viable);
             stack.addAll(chosen);
        fi
  od
  stack.pop();
```

Figure 7: Parsing algorithm

The way we deal with `result` and `exception` deserves commentary. Throwing exceptions in an uncontrolled manner can cause the parse information to be destroyed; for example it may not consume the ¿ tokens properly. We have to have some support for this in case of exceptions in the

```
first := {(n, ∅) : n ∈ NT} ∪ {([], {([[true]], ϵ)})};
do for each production P = X → Y1Y2 ... Yn do
        if P = X → ϵ then continue fi
        s := first(Y1Y2 ... Yn);
        p := [[true]];
        for each Yi do
            d := false;
            if Yi ∈ SP then p := p ∧ Yi;
            elsif Yi ∈ SA then ;
            elsif Yi ∈ T then target := first(Yi);
                                if ϵ ∈ target then putative := {(q, t) : (q, t) ∈ target ∧ t ≠ ϵ}
                                                    if s ≠ putative ∧ s ≠ target
                                                       then first(Y1Y2 ... Yn) := putative; fi
                                else if s ≠ target then first(Y1Y2 ... Yn) := target fi
                                     d := true;
                                fi
            else if Yi ∉ s then first(Y1Y2 ... Yn) := {(p, Yi)}; fi
                 d := true;
            fi
            if ¬d then first(Y1Y2 ... Yn) := first(Y1Y2 ... Yn) ∪ {(p, ϵ)}; fi
        od
    od
    for each production P = X → Y1Y2 ... Yn do
        first(P) := first(P) ∪ first(Y1Y2 ... Yn);
    od
od while any element in first has changed;
```

Figure 5: Algorithm for computing *first* sets

Java escapes, but arguably these are errors anyway; we recover in this situation by throwing away everything on the parse stack until we reach the ¿ we were expecting, and then propagate the exception. But not all exceptions are errors: a faithful representation of the interface may require that exceptions be thrown, and we must then throw them in a manner consistent with our parsing algorithm. Accordingly to handle this we store the exception in a member variable and then throw it at the end of the stub method.

Return values are similar; return in Java only works for the most immediate enclosing method. Accordingly we use the same technique we use for handling exceptions; we store the return value in a member variable and then return it at the end of the stub method.

4. VERIFICATION OF EJB CLIENTS

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

4.1 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; we are 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, which is fetched 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 code sequences like `em.getTransaction().begin();` are a common idiom.

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. We do not consider a simulation of the query lan-

$follow := \{(n, \emptyset) : n \in NT\} \cup \{([], \{([\![true]\!], \Box)\})\}$;
do for each production $P = X \to Y_1 Y_2 \ldots Y_n$ **do**
  for each sublist $Y_1 Y_2 \ldots Y_n, Y_2 \ldots Y_n, \ldots, Y_i \ldots Y_n, \ldots, Y_n$ **do**
    if $Y_i \in NT$ **then if** $i = n$ **then** $s := []$; $f := \{([\![true]\!], \epsilon)\}$
      else $s := Y_{i+1} \ldots Y_n$;
        $f := first(s)$;
        if $\exists p : (p, \epsilon) \in f$ **then** $f := f \cup follow(X)$ **fi**
        $follow(Y_i) := follow(Y_i) \cup f$;
      fi
    fi
  od
od
od while any element in *follow* has changed;

Figure 6: Algorithm for computing *follow* sets

guage in this paper, largely because simulating it properly would require a full string parser for the SQL-like syntax. Our interface specification also does not model concurrent update operations or the XML extension defined by the Persistence API.

4.2 Persistence API clients

The normal life cycle of a Persistence API client is to use an `EntityManagerFactory`, to retrieve an `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 may load the entire tree into memory one node at a time by requesting parents and children.

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 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 we applied our interface grammar tools to the Persistence API. We can also use our 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, we have written interface grammars for each of the `EntityManagerFactory`, `EntityManager` and `EntityTransaction` interfaces. Portions of these grammars are shown in Figure 2. 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.

4.3 Experiments

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

We analyze the following test cases:

- `EntityManagerTest.testContains` 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` ensures that objects managed by the `EntityManager` transition to the detached state after a `clear`.
- `EntityManagerTest.testPersistNoneGenerator` ensures that a simple object is equal to itself after it has been persisted and reloaded.
- `EntityManagerTest.testIsOpen` verifies that an `EntityManager` is open upon creation and stays that way until it is closed.
- `AssociationTest.testBidirOneToOne` verifies that persisting one half of a bidirectional association will persist the other half as well.
- `AssociationTest.testMergeAndBidirOneToOne` verifies that the bidirectional association works even with detached objects.

- `CallbacksTest.testCallBackListenersHeirarchy` verifies that methods tagged with `@PrePersist` are called when the object in question is persisted.
- `CallbacksTest.testException` 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` verifies that asking for objects using the wrong primary key type is an illegal operation.
- `ExceptionTest.testEntityNotFoundException` verifies that nonexistent objects fetched with `getReference` should raise exceptions when they are referred to.
- `InheritanceTest.testFind` verifies that if A is a subclass of B, persisting an instance of A and asking for all Bs should retrieve the first object.
- `testAlwaysTransactionalOperations` method of `FlushAndTransactionTest` class checks that flushes and locks are only valid from within transactions.

We conducted our experiments using a Linux machine with 2.8 GHz Pentium 4 with 2 gigabytes of memory. We ran each of the test cases listed above in two configurations on our testbed. We first ran the test case normally, to test the validity of our interface. We achieved a 100% success rate in this, showing that our interface successfully simulates the behavior of EJB Persistence API. Then we ran it again in a configuration where any exceptions seen in the parsing trigger fatal runtime errors; these test whether the clients are faithful to the Persistence API. Of our test cases, some 5 of the 12 deliberately perform illegal operations, which are caught. Our results are shown in Table 1.

We were unable to measure run times in increments of less than a second, as JPF itself does not support getting the current time from within the running program, and provides only second and megabyte resolution for its output. Even with this limited resolution, our results demonstrate that using our approach verification of both nontrivial client code and the interface specification can be done efficiently using relatively little memory and execution time.

5. RELATED WORK

Use of finite state machines for specification, verification and extraction of interfaces have been studied extensively [10, 9, 17, 2, 6, 7]. Finite state machines cannot specify nested-call structures such as the recursive transaction example we use in this paper. The interface grammars we propose in this paper enables us to specify such interactions. Moreover, we believe that the semantic predicates and actions that are allowed in our interface grammars are necessary to model interfaces of complex components. Another factor that differentiates our work from that of [17, 2] is that we do not extract interfaces, rather, we 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 [4, 5]. 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 [6, 7, 8], finite state interface specifications are used to achieve modular verification where behavior verification and interface verification are executed as two separate steps. Interface grammars proposed here provide a richer language for specification of interfaces and can be integrated to the modular verification approach used in [6, 7, 8].

Environment generation is a critical problem for achieving modularity in software model checking and has been studied before. [12] presents techniques for automatically closing environments of open reactive programs by automatically creating the most general environment for the program using dataflow analysis. [14], on the other hand, investigates automatically generating environments for components using side effect and points-to analyses for modular model checking. We 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. We 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 discussed in [15] 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. Our 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, we are 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, we believe that it is more likely for developers to write interface specifications for different components rather than writing an environment for a particular program.

6. CONCLUSIONS

We have proposed and implemented a new framework for conducting modular software model checking based on interface grammars. We proposed an interface specification language based on interface grammars and we built a compiler that automatically generates stubs for components using interface specifications written in our interface specification language. We have used this tool to conduct model checking relating to the key interfaces of the Enterprise JavaBeans Persistence API, and have demonstrated that our approach is feasible and efficient. In future work, we would like to apply our interface grammars to model checking of concurrent programs, as well as the generation of object graphs for model checking.

7. REFERENCES

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

Test case	Interface verification		Client verification		
	Run time	Memory used	Run time	Memory used	Errors detected
AssociationTest					
testBidirOneToOne	2 s	15 MB	2 s	16 MB	
testMergeAndBidirOneToOne	2 s	15 MB	2 s	16 MB	
CallbacksTest					
testCallBackListenersHeirarchy	2 s	15 MB	2 s	15 MB	
testException	2 s	15 MB	2 s	15 MB	yes
EntityManagerTest					
testClear	2 s	15 MB	2 s	15 MB	
testContains	3 s	26 MB	2 s	15 MB	yes
testIsOpen	2 s	15 MB	2 s	15 MB	
testPersistNoneGenerator	2 s	15 MB	2 s	15 MB	
ExceptionTest					
testEntityNotFoundException	2 s	15 MB	2 s	15 MB	yes
FlushAndTransactionTest					
testAlwaysTransactionalOperations	2 s	15 MB	2 s	15 MB	yes
GetReferenceTest					
testWrongIdType	2 s	15 MB	2 s	15 MB	yes
InheritanceTest					
testFind	2 s	15 MB	2 s	15 MB	

Table 1: Run time and memory usage for test cases on stubbed interface

[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] ANother Tool for Language Recognition (ANTLR). http://www.antlr.org/.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.