Interface Grammars for Modular Software Model Checking

Graham Hughes, Tevfik Bultan

***Abstract*—Verification techniques that rely on state enumeration (such as model checking) face two important challenges: 1) *State-space explosion:* exponential increase in the state space with the increasing number of components. 2) *Environment generation:* modeling components that are either not available for analysis, or that are outside the scope of the verification tool at hand. We propose a semi-automated approach for attacking these two problems. In our approach, interfaces for the components that are outside the scope of the current verification effort are specified using an interface specification language based on grammars. Specifically, 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. 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. The stub/parser generated from the interface grammar of a component can be 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. 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 with JPF using our approach, whereas they cannot be verified with JPF directly since JPF cannot handle EJB components.

***Index Terms*—specification languages, model checking, interface grammars, modular verification**

I. Introduction

MODEL checking is an algorithmic verification technique that exhaustively explores the state space of a system in order to check for violations of temporal properties [1]. Earlier model checkers, with their dedicated specification languages, have been used for specification and analysis of numerous hardware and software systems [2], [3]. More recently, the application of model checking techniques directly to programs [4]–[11] has shown promise for specific verification tasks, such as checking for concurrency errors [5] or checking device drivers for interface violations [6]. However, software model checking is not yet mature enough to be widely used in software development. There are two related problems that hinder the applicability of model checking to software in a wider scale: 1) state space explosion (i.e., the 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 the parts of the 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.

These limitations are shared by all software model checking techniques and tools. In particular, they are apparent in the Java Path Finder (JPF) [5], 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.

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 (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

Both authors are affiliated with the Computer Science Department of the University of California, Santa Barbara, CA, 93106 USA. Email: {graham, bultan}@cs.ucsb.edu

This work is supported by NSF grants CCF-0614002 and CCF-0716095.

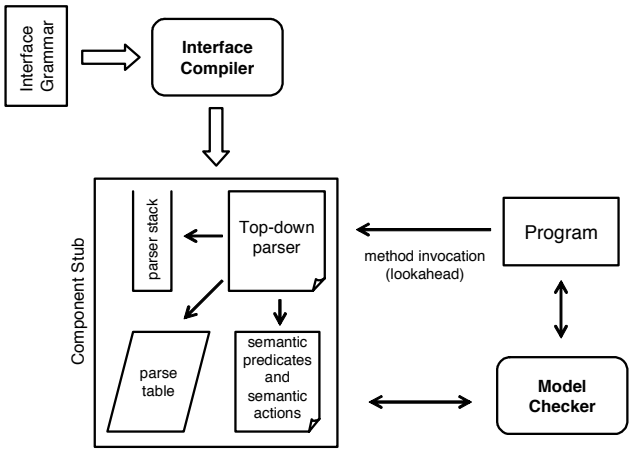

Fig. 1. An overview of our approach

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 [12]). 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.

Fig. 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—which corresponds to an error in the call sequence—or a semantic predicate violation—which corresponds 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. To address this need, our interface specification language provides a nondeterministic switch operator; that is, if two or more switch cases evaluate to true, then one of them is selected for execution nondeterministically.

We assume that the target software model checker provides a *nondeterministic choice* primitive. Support for nondeterministic choice primitives in software model checking is common [4] since it is a useful tool for environment modeling [13], i.e., it can be used for developing environment models that over-approximate the behaviors of the components that are outside the scope of the model checker. 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. Our current approach has two limitations: 1) We focus on client-side verification and do not check the server-side interface conformance. 2) Our interface grammars and interface compiler do not handle component interfaces with call-backs. We will clarify these limitations with a hypothetical scenario. Assume that we wish to check the behavior of a component A which calls the methods of another component B. Also, assume that, we are unable to include component B in our verification effort (either we may not have access to component B, or it may not be possible to represent component B in a form that can be analyzed with the available verification tool). In this scenario, our approach works as follows: We ask the user to write an interface grammar that summarizes (i.e., over-approximates) the behavior of component B. Our interface compiler converts this interface grammar to an executable stub for component B. Then, we check the behavior of component A by replacing component B with the automatically generated stub. Due to limitation 2, currently, our tool only handles scenarios where component A calls the methods of component B but there is no call-back from component B to component A. Furthermore, we do not currently check the conformance of component B to the interface specification. We believe that both of these limitations can be addressed in the future by extending our interface grammar language and compiler to a bi-directional interface language and a bi-directional interface compiler.

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 II provides simple interface grammar examples and a formal model for interface grammars. Section III presents our interface specification language. Section IV discusses the interface compiler. Section V discusses the EJB case study. Section VI includes a discussion on related work, and Section VII concludes the paper.

II. 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 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 [14] 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 we 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

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

Note that this interface specification allows nesting of matching `begin` and `commit` or `rollback` calls and, 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, we add to our 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`. We 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 $l \equiv 0$ we can reset r to false. If we denote the semantic action containing the code `x` as $\langle\!\langle \texttt{x} \rangle\!\rangle$ and the semantic predicate evaluating the code `p` as $[\![\texttt{p}]\!]$, then the amended grammar looks as follows:

Grammar 3. Nested transaction grammar with semantic elements

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

To summarize, the call sequences specified by Grammar 1 above can also be specified using a finite state machine. However, the call sequences for recursive transactions specified by Grammars 2 and 3 cannot be specified using finite state machines.

A. Formal Interface Grammars

The above description of interface grammars has been somewhat informal; we have not defined the effects of the

semantic elements used in Grammar 3 nor have we established what sentences belong to such a grammar. Although the methods used in the Grammars 1, 2, and 3 do not have any arguments and do not return any values, we do not restrict interface grammars to specification of such sequences. I.e., interface grammars have the ability to specify constraints on the method call arguments and the return values. Hence, there are two crucial differences between the standard context free grammars and the interface grammars: 1) Instead of specifying a set of sentences that correspond to sequences of terminal symbols, interface grammars specify sequences of method calls and returns, where the method arguments and return values have to be taken into account, 2) Interface grammars have semantic actions and predicates, and without specifying the scoping and execution semantics for these semantic actions and predicates the semantics of interface grammars cannot be formalized. Here we present a formal definition for interface grammars that clarifies both of these differences.

We define our grammars based on recognizing possible execution traces. We restrict our 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. We denote the initiation of a method call from a program into the component for the method a by $?a$, and we denote the termination of that method call—that is, the return—by $¿a$. For accurate modeling of a component, we must also track the method arguments and the return values. We assume that methods have one argument and one return value; multiple arguments are represented as a tuple. We 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 $¿a[y]$. We frequently write the initiation of a method call that takes no arguments as $?a[]$ or $?a[\perp]$, and the termination of a method call that has a void return value as $¿a[]$ or $¿a[\perp]$.

We can use these traces to formalize interface grammars. We do so here, using notation based on that of Nielson, et al. [15]. Briefly, the fundamental sets we deal with here are written in **boldface**; v^* means the Kleene closure of v; $f[x \mapsto y]$ is the function that maps x to y and otherwise behaves as f; $f[x \mapsto y, a \mapsto b] = f[x \mapsto y][a \mapsto b]$; $f \odot g$ is the function that behaves as f for all values in $\mathrm{dom}(f)$ and behaves as g otherwise; and $[]$ is a function with empty domain and range.

We require several sets to define the semantics of our grammars. Accordingly, we 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; $?a$, $¿a \in \mathbf{\Sigma}$ are members of the alphabet of method calls and returns, where $?a$ denotes a call to a method a and $¿a$ denotes the return from a method a; $\xi \in \mathbf{Loc}$ is a member of the set of locations where we may store values of variables; $x, y \in \mathbf{Var}$ are variables; and $\mathbf{Dom}$ is an unconstrained domain set that represents the values the variables can take. Using these we define the following sets:

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

Here ρ, ς and σ are partial functions; every location may not be assigned a value, nor must every variable be bound to a location. In practice we will construct elements of $\mathbf{State}$ by composing elements of $\mathbf{Env}$ and $\mathbf{Store}$; so $\sigma = \varsigma \circ \rho$. We use $\uparrow$ and $\downarrow$ to denote opening and closing a new scope, respectively.

Unfortunately the above definition of $\mathbf{\Sigma}$ is not really sufficient; specifically we must record, as a part of our traces, the method arguments and method return values. We define the set $\mathbf{\Sigma}_\circ = \mathbf{\Sigma} \times \mathbf{Var}$ to be the set of symbols combined with variable names to denote the arguments or return values as appropriate, and $\mathbf{\Sigma}_\bullet = \mathbf{\Sigma} \times \mathbf{Dom}$ to be the set of symbols combined with values that represent the record of a trace. To ease the presentation, we 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 we need a $\mathbf{Sym}_\circ = \mathbf{NT} \cup \mathbf{\Sigma}_\circ \cup \mathbf{Action} \cup \mathbf{Pred} \cup \mathbf{Decl} \cup \{\uparrow, \downarrow\}$ and $\mathbf{Prod}_\circ = \mathcal{P}(\mathbf{NT} \times \mathbf{Sym}_\circ^*)$.

From this we 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. We would like to define derivation for this grammar, so that we can fully describe whether a sentence is in the language of the grammar. We define single-step derivation ($\Rightarrow$) and ultimate derivation ($\Rightarrow^*$) in Fig. 2.

The signatures for $\Rightarrow$ and $\Rightarrow^*$ are as follows. For the arguments, 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 we have yet to process $\mathbf{Sym}_\circ^*$. The result is either a string of trace symbols $\mathbf{\Sigma}_\bullet^*$, denoting that we have applied Equation (1) and are in a state where we can accept the string; or a string like the argument, meaning more work has to be completed before we can accept a derivation. We must keep track of the environment and store at all for the semantic predicates to function, and we must keep track of them separately rather than conflating them into a single $\mathbf{State}$ to permit lexical scoping of declarations.

Equation (1) defines the derivation when the end of a string has been reached; we drop the scoping information ρ and the store information ς and accept.

Equation (2) defines the derivation rule for a semantic action

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

$$A\,\{\rho,\varsigma\} \;\Rightarrow A \tag{1}$$

$$\frac{\mathrm{dom}(\mathrm{dom}(f)) \subseteq \mathrm{dom}(\rho) \quad \varsigma' = (\rho^{-1} \circ f(\varsigma \circ \rho)) \odot \varsigma}{A\,\{\rho,\varsigma\}\,\langle\!\langle f \rangle\!\rangle B \Rightarrow A\,\{\rho,\varsigma'\}\,B} \tag{2}$$

$$\frac{\mathrm{dom}(\mathrm{dom}(p)) \subseteq \mathrm{dom}(\rho) \quad p(\varsigma \circ \rho) = \mathbf{true}}{A\,\{\rho,\varsigma\}\,[\![p]\!]\,B \Rightarrow A\,\{\rho,\varsigma\}\,B} \tag{3}$$

$$\frac{\xi \notin \mathrm{dom}(\varsigma)}{A\,\{\rho,\varsigma\}\,\Delta x\,B \Rightarrow A\,\{\rho[x \mapsto \xi], \varsigma[\xi \mapsto \bot]\}\,B} \tag{4}$$

$$\frac{\langle S, s_1 \ldots s_n\rangle \in \mathbf{P}}{A\,\{\rho,\varsigma\}\,S\,B \Rightarrow A\,\{\rho,\varsigma\}\,s_1 \ldots s_n\,B} \tag{5}$$

$$\frac{\{\rho,\varsigma\}\,B \Rightarrow^*\, B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\;\uparrow B \downarrow C \Rightarrow A\,B'\,\{\rho,\varsigma'\}\,C} \tag{6}$$

$$\frac{\xi_1, \xi_2 \notin \mathrm{dom}(\varsigma) \quad \{\rho[x \mapsto \xi_1, y \mapsto \xi_2], \varsigma[\xi_1 \mapsto v, \xi_2 \mapsto \bot]\}\,B \Rightarrow^*\, B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\,?a(x)\,B\,¿a(y)\,C \Rightarrow A\,?a[v]\,B'\,¿a[(\varsigma' \circ \rho')(y)]\,\{\rho,\varsigma'\}\,C} \tag{7}$$

$$\frac{\exists u_1, \ldots u_n : A\,\{\rho,\varsigma\}\,B\,C \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n \Rightarrow A\,B'\,\{\rho',\varsigma'\}\,C}{A\,\{\rho,\varsigma\}\,B\,C \Rightarrow^*\, A\,B'\,\{\rho',\varsigma'\}\,C} \tag{8}$$

$$\frac{\exists u_1, \ldots u_n : A\,\{\rho,\varsigma\}\,B \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_n \Rightarrow A\,B'\,\{\rho',\varsigma'\}}{A\,\{\rho,\varsigma\}\,B \Rightarrow^*\, A\,B'} \tag{9}$$

Fig. 2. Derivation rules

$\langle\!\langle f \rangle\!\rangle$. Since $\langle\!\langle f \rangle\!\rangle$ 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 syntactically possible for f to refer to variables that are not in $\mathrm{dom}(\rho)$, that is variables that are not in scope, we 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, we need an updated store, ς', rather than a state. We 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 we must adjoin it to the old store with $\odot$. We justify the existence of ρ^{-1} by noting that ρ is one-to-one; it is only modified in Equations (7) and (4), and there ξ, ξ_1 and ξ_2 are constrained to be values that have never been in $\mathrm{ran}(\rho)$.

It is conceivable that $\varsigma \circ \rho$ may not be well defined; that is, $\mathrm{ran}(\rho) \not\subseteq \mathrm{dom}(\varsigma)$. We 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 we add a new element to $\mathrm{ran}(\rho)$—that is, each time we map a variable to a new location—we map that same location to a value in ς. So $\varsigma \circ \rho$ is always well defined.

Equation (3) is similar to Equation (2), 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 we insist be a subset of the variables in scope. Similarly we insist that the predicate p be true in the current state.

Equation (4) deals with variable declarations. We 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. We must update ρ to reflect that x has been bound to the location ξ, and then bind the location ξ to $\bot$, reflecting that it has not yet been assigned a value.

Equation (5) defines nonterminal substitution; for any production $S \rightarrow s_1 \ldots s_n$ in $\mathbf{P}$, which is equivalently stated $\langle S, s_1 \ldots s_n\rangle \in \mathbf{P}$, we can substitute S's right hand side for S wherever it appears.

Equation (6) 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 (7) 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), we 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 we 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', we can derive the original method call, retaining the old scope but using the new

store as in Equation (6) above. But, we must reconstruct the return value for ¿a; this is simply the value of y in the state following the derivation of B', which is $\varsigma' \circ \rho'$. We record v and $(\varsigma' \circ \rho')(y)$ for the trace in the result, using the shorthand defined above for $\Sigma_\bullet$.

Finally, Equation (8) and (9) define how to define multiple step derivation from the above rules. We need two rules to permit Equation (1) to be used.

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

B. An Example

An example is in order. Consider Grammar 3. We restate it formally here as the grammar G, where:

$$
\begin{aligned}
G &= \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle \\
\mathbf{NT} &= \{\textit{Start}, \textit{Base}, \textit{Tail}\} \\
\mathbf{\Sigma} &= \{\texttt{?begin}, \texttt{?setRollbackOnly}, \texttt{?commit}, \\
&\quad \texttt{?rollback}, \texttt{¿begin}, \texttt{¿setRollbackOnly}, \\
&\quad \texttt{¿commit}, \texttt{¿rollback}\} \\
\mathbf{Var} &= \{r, l\} \\
\mathbf{Dom} &= \mathbf{B} \cup \mathbb{Z} \\
\mathcal{Q} &= \left\langle\!\!\left\langle \begin{array}{l} \lambda\sigma.\sigma' = \sigma[l \mapsto \sigma(l) - 1]; \\ \qquad \sigma'[r \mapsto (\sigma'(r) \wedge \sigma'(l) \not\equiv 0)] \end{array} \right\rangle\!\!\right\rangle \\
\mathbf{SA} &= \{\langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle, \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle, \\
&\quad \mathcal{Q}, \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}] \rangle\!\rangle\} \\
\mathbf{SP} &= \{[\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\} \\
\mathbf{P} &= \{\langle \textit{Start}, \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle \ \textit{Base} \rangle, \\
&\quad \langle \textit{Base}, \texttt{?begin}()\ \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle \\
&\qquad \texttt{¿begin}()\ \textit{Base Tail}\ \mathcal{Q}\ \textit{Base} \rangle, \\
&\quad \langle \textit{Base}, \texttt{?setRollbackOnly}()\ \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{true}] \rangle\!\rangle \\
&\qquad \texttt{¿setRollbackOnly}()\ \textit{Base} \rangle, \\
&\quad \langle \textit{Base}, \epsilon \rangle, \\
&\quad \langle \textit{Tail},\ [\![\lambda\sigma.\sigma(r) \equiv \textbf{false}]\!]\ \texttt{?commit}()\ \texttt{¿commit}() \rangle, \\
&\quad \langle \textit{Tail},\ \texttt{?rollback}()\ \texttt{¿rollback}() \rangle\}
\end{aligned}
$$

Here, to simplify presentation we abbreviate one frequently used semantic action as $\mathcal{Q}$. As a notational convenience, when a method has no arguments rather than writing it as $?a(x)$ and then not using x, we write it as $?a()$ or $?a[]$. Similarly, when a method has a void return value, we write it as ¿$a()$ or ¿$a[]$. All the methods in this example take no arguments and have void returns. We use the syntax $\lambda x.y$ to denote the anonymous function taking one argument, x, and performing y.

In constructing this formal grammar, we have distinguished method calls and returns, which Grammar 3 conflated. We 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 we must track the return values from a method in the trace, and the only way to do that is to mark method returns in some fashion.

We assert that the trace

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

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, we begin a derivation from $\{\rho_0, \varsigma_0\}\, S$, as follows:

$$
\begin{aligned}
& \{\rho_0, \varsigma_0\}\, S \\
\Rightarrow\ & \text{by (5)} \\
& \{\rho_0, \varsigma_0\}\, \langle\!\langle \lambda\sigma.\sigma[r \mapsto \textbf{false}, l \mapsto 0] \rangle\!\rangle\ \textit{Base} \\
\Rightarrow\ & \text{by (2)} \\
& \{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ \textit{Base} \\
\Rightarrow\ & \text{by (5)} \\
& \{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ \texttt{?begin}() \\
& \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle\ \texttt{¿begin}()\ \textit{Base Tail} \\
& \mathcal{Q}\ \textit{Base}
\end{aligned}
$$

To proceed, we need to apply Equation (7); to do that we need to perform a sub-derivation as follows:

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

Now, we can apply Equation (7):

$$
\begin{aligned}
& \{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 0]\}\ \texttt{?begin}() \\
& \langle\!\langle \lambda\sigma.\sigma[l \mapsto \sigma(l)+1] \rangle\!\rangle\ \texttt{¿begin}()\ \textit{Base Tail}\ \mathcal{Q}\ \textit{Base} \\
\Rightarrow\ & \text{by (7)} \\
& \texttt{?begin}[\bot]\ \texttt{¿begin}[\bot] \\
& \{\rho_0, [\xi_0 \mapsto \textbf{false}, \xi_1 \mapsto 1, \xi_{2\ldots3} \mapsto \bot]\} \\
& \textit{Base Tail}\ \mathcal{Q}\ \textit{Base}
\end{aligned}
$$

Continuing in this same vein, we eventually derive that

$$
\begin{aligned}
\{\rho_0, \varsigma_0\}\, S \Rightarrow^* {} & \texttt{?begin}[\bot]\ \texttt{¿begin}[\bot]\ \texttt{?begin}[\bot] \\
& \texttt{¿begin}[\bot]\ \texttt{?commit}[\bot]\ \texttt{¿commit}[\bot] \\
& \texttt{?rollback}[\bot]\ \texttt{¿rollback}[\bot]
\end{aligned}
$$

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

III. Interface Grammar Language

Having defined the formalisms, we now proceed to define the language we use to write the grammars. In Fig. 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

$$main \rightarrow class* \quad (10)$$

$$class \rightarrow \texttt{class}\ \text{CLASSID}\ \{\ item^*\ \} \quad (11)$$

$$item \rightarrow semact\ \texttt{;} \quad (12)$$
$$\mid\ rule \quad (13)$$

$$rule \rightarrow \texttt{rule}\ \text{RULEID}\ block \quad (14)$$

$$block \rightarrow \{\ statement^*\} \quad (15)$$

$$statement \rightarrow block \quad (16)$$
$$\mid\ \texttt{apply}\ \text{RULEID}\ \texttt{;} \quad (17)$$
$$\mid\ semact\ \texttt{;} \quad (18)$$
$$\mid\ declaration\ \texttt{;} \quad (19)$$
$$\mid\ \texttt{choose}\ \{\ cbody^*\ \} \quad (20)$$
$$\mid\ \texttt{?}\ \text{MINVOCATION}\ (\ arguments\ )\ \texttt{;} \quad (21)$$
$$\mid\ \texttt{return}\ \text{MRETURN}\ semexpr?\ \texttt{;} \quad (22)$$

$$cbody \rightarrow \texttt{case}\ select?\ \texttt{:}\ \{\ statement^*\ \} \quad (23)$$

$$select \rightarrow \texttt{?}\ \text{MINVOCATION}\ (\ arguments\ )\ sempred? \quad (24)$$
$$\mid\ sempred \quad (25)$$

$$arguments \rightarrow (\text{TYPE ID}\ (\texttt{,}\ \text{TYPE ID})^*)? \quad (26)$$

$$sempred \rightarrow \langle\!\langle \text{EXPR} \rangle\!\rangle \quad (27)$$

$$semexpr \rightarrow \langle\!\langle \text{EXPR} \rangle\!\rangle \quad (28)$$

$$semact \rightarrow \langle\!\langle \text{STATEMENT} \rangle\!\rangle \quad (29)$$

$$declaration \rightarrow \text{TYPE ID} = \langle\!\langle \text{EXPR} \rangle\!\rangle \quad (30)$$

Fig. 3. Abstract syntax for the interface grammar language

```
class NestedTransaction {
  ⟪bool r; int i;⟫
  rule start {
    ⟪r = false; i = 0;⟫
    apply base;
  }
  rule base {
    choose {
      case ?begin(): {
        ⟪l++;⟫
        return begin;
        apply base;
        apply tail;
        ⟪l--; if (l == 0) r = false;⟫
        apply base;
      }
      case ?setRollbackOnly(): {
        ⟪r = true;⟫
        return setRollbackOnly;
        apply base;
      }
      case: { }
    }
  }
  rule tail {
    choose {
      case ?commit() ⟪r == false⟫: {
        return commit;
      }
      case ?rollback(): {
        return rollback;
      }
    }
  }
}
```

Fig. 4. Interface grammar for Grammar 3

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. In the grammar shown in Fig. 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. In Fig. 4, we have translated Grammar 3 into this syntax.

An interface grammar consists of a set of class interfaces—not to be confused with Java interfaces—(represented in Equation (10) in Fig. 3). The interface compiler generates one stub class for each class interface. These stub classes generated by the interface compiler pass the incoming calls to the table-driven parser generated by the interface compiler as lookahead symbols. The parser uses the parser stack, the lookahead, and the parse table to guide the behavior. We explain the behavior of the parser generated by the interface compiler in detail in Section IV.

Each class interface in an interface grammar consists of a set of semantic actions and a set of production rules that define the interface grammar for that class (Equations (11), (12) and (13)). A *semantic action* is simply a piece of Java code that is inserted to the stub class that is generated for the component (Equation (29)). A *rule* corresponds to a production rule in the interface grammar. Each rule has a name and a block (Equation (14)). A rule block consists of a sequence of statements (Equation (15)). 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 (Equations (16)–(22)). 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 (Equation (30)). A *choose block* is simply a switch statement (Equations (20) and (23)). 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 (Equations (24) and (25)). 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. Finally, 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 lookahead 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.

We have defined the semantics of interface grammars using the formalism introduced in Section II-A. To convert a grammar in the syntax specified in Fig. 3 to our formalism, we use

an attributed grammar. The attributed grammar does the following: given a parse tree from the grammar in Fig. 3, it computes a formal grammar suitable for the derivation rules in Fig. 2. This attributed grammar is presented in Fig. 5. In this attributed grammar, for every symbol s, $s.t \in \mathbf{Sym}_\circ^*$ is a sequence of nonterminals, terminals, semantic actions and predicates, and $s.p \subseteq \mathbf{Prod}$ is a set of productions. For *arguments*, *arguments*.v is a tuple of $\mathbf{Var}$. We use $\|$ to denote concatenation of sequences.

The syntax the compiler reads is a faithful rendering of Fig. 3's grammar into text. Fig. 6 shows the transaction and recursive transaction classes from the EJB interface specification. In addition to the `begin`, `commit`, `rollback` and `setRollbackOnly` methods we discussed in Section II, we also include two query methods: `isActive` and `getRollbackOnly`. The method `isActive` returns true if a transaction is active and the method `getRollbackOnly` returns the rollback-only state, i.e., it returns `true` if `setRollbackOnly` has been called and all the pending transactions that were active when `setRollbackOnly` was called have not been roll-backed yet.

The specification shown in Fig. 6(a) is the interface grammar for the `transaction` class and the specification shown in Fig. 6(b) is the interface grammar for the `recursive_transaction` class. The nonterminals used in the non-recursive transaction grammar are `start`, `inactive`, `active`, and `rollback_only`. By default `start` is the start symbol. Note that the nonterminals `start`, `inactive` and `active` in Fig. 6(a) are used similarly as the nonterminals *Start*, *Inactive*, and *Active* in Grammar 1 from Section II. The additional nonterminal `rollback_only` is used to keep track of the rollback-only state.

The nonterminals used in the recursive transaction grammar shown in Fig. 6(b) are `start`, `base` and `tail`. Again, these nonterminals are used similarly as the the nonterminals *Start*, *Base*, and *Tail* in Grammars 2 and 3 from Section II. Note that the variables `level` and `isRollbackOnly` used in the interface grammar shown in Fig. 6(b) correspond to the variables l and r in the Grammar 3 from Section II.

An interesting difference between the transaction and recursive transaction grammars in Fig. 6 is the way they handle rollback-only status. In the transaction grammar, rollback-only status is handled at the grammar level by using a nonterminal that corresponds the case when rollback-only is set. In the recursive transaction grammar, rollback-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 cause the interface specification to degenerate into a hand written Java stub.

IV. 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 an interface grammar;
3) Output a parser for this resulting interface grammar.

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 grammar specifications to interface grammars, generation of parsers for the resulting interface grammars, and the runtime system for the automatically generated parser/stubs.

Our method generates a interface 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 [14] 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 for interface grammars 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. Due to these concerns, our interface compiler generates table-driven parsers for interface grammars.

A. 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 [16] to parse the input interface

$$
\begin{array}{ll}
\textit{rule} \rightarrow \texttt{rule}\ \text{RULEID}\ \textit{block} & \textit{rule}.p := \{(\text{RULEID}, \textit{block}.t)\} \cup \textit{block}.p \\
\textit{block} \rightarrow \{\ \textit{statement}^*\ \} & \textit{block}.t := \uparrow \ \| \ \Big\|_i \textit{statement}_i.t \ \| \ \downarrow \\
 & \textit{block}.p := \bigcup_i \textit{statement}_i.p \\
\textit{statement} \rightarrow \textit{block} & \textit{statement}.t := \textit{block}.t \\
 & \textit{statement}.p := \textit{block}.p \\
\textit{statement} \rightarrow \texttt{apply}\ \text{RULEID}\ ; & \textit{statement}.t := \text{RULEID} \\
\textit{statement} \rightarrow \textit{semact} & \textit{statement}.t := \langle\!\langle \textit{semact.statement} \rangle\!\rangle \\
\textit{statement} \rightarrow \textit{declaration}\ ; & \textit{statement}.t := \Delta \textit{declaration.id} \ \| \\
 & \qquad \langle\!\langle \textit{declaration.id} = \textit{declaration.expr}\texttt{;} \rangle\!\rangle \\
\textit{statement} \rightarrow \texttt{choose}\ \{\ \textit{cbody}^*\ \} & \textit{statement}.t := \textit{statement.id} \\
 & \textit{statement}.p := \bigcup_i \{(\textit{statement.id}, \textit{cbody}_i.t)\} \cup \textit{cbody}_i.p \\
\textit{statement} \rightarrow ?\ \text{MINVOCATION}\ (\textit{arguments})\texttt{;} & \textit{statement}.t := ?\text{MINVOCATION}(\textit{arguments}.v) \\
\textit{statement} \rightarrow \texttt{return}\ \text{MRETURN}\ ; & \textit{statement}.t := ¿\text{MRETURN}(\texttt{result}) \\
\textit{statement} \rightarrow \texttt{return}\ \text{MRETURN}\ \textit{semexpr}\ ; & \textit{statement}.t := \langle\!\langle \texttt{result=}\textit{semexpr.expr} \rangle\!\rangle \ \| \\
 & \qquad ¿\text{MRETURN}(\texttt{result}) \\
\textit{cbody} \rightarrow \texttt{case}\ \textit{select}? : \{\ \textit{statement}^*\ \} & \textit{cbody}.t := \text{SELECT}.t \ \| \ \uparrow \ \| \ \Big\|_i \textit{statement}_i.t \ \| \ \downarrow \\
 & \textit{cbody}.p := \bigcup_i \textit{statement}_i.p \\
\textit{select} \rightarrow ?\ \text{MINVOCATION}\ (\textit{arguments})\ \textit{sempred} & \textit{select}.t := ?\text{MINVOCATION}(\textit{arguments}.v) \ \| \\
 & \qquad [\![\textit{sempred.expr}]\!] \\
\textit{select} \rightarrow ?\ \text{MINVOCATION}\ (\textit{arguments}) & \textit{select}.t := ?\text{MINVOCATION}(\textit{arguments}.v) \\
\textit{select} \rightarrow \textit{sempred} & \textit{select}.t := [\![\textit{sempred.expr}]\!] \\
\textit{arguments} \rightarrow \epsilon & \textit{arguments}.v := \langle\rangle \\
\textit{arguments} \rightarrow \text{TYPE ID}_a(\texttt{,}\ \text{TYPE ID}_i)^* & \textit{arguments}.v := \left\langle \text{ID}_a \| \Big\|_i \text{ID}_i \right\rangle
\end{array}
$$

Here, $\|$ 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.

Fig. 5. Translation from syntax tree to interface grammar

specification and construct an abstract syntax tree representing the input specification.

1) Generating the interface grammar: The second major step of computation in our interface compiler involves converting the abstract syntax tree generated by the parser into a interface grammar. In addition to nonterminal and terminal symbols, the resulting interface grammar also contains semantic predicates and actions. The terminal symbols of the resulting interface 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. To accomplish this, we use the attributed grammar defined previously in Fig. 5.

2) The parsing algorithm: Given an interface grammar, we can now begin to parse it. By embedding the grammar (or its equivalent parse table, as discussed later) in the runtime environment that completes the stub, we can parse it there. We can consider each class in isolation since they are independent and contain independent parsing tables.

The core parsing algorithm is given in Fig. 7. Each stub we generate contains code that corresponds to an implementation of this algorithm. The algorithm shown in Fig. 7 is based on the standard LL(1) table-driven parsing algorithm but adds semantic actions and semantic predicates and also allows ambiguity.

For the purpose of verification, we assume the presence of a model checker that has exposed its backtracking primitives in the following function: **choose**(S) nondeterministically chooses one element from the set S and returns it. For convenience, we denote throwing an exception or violating an assertion with **fail**.

```
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;
      }
    }
  }
}
```

(a)

```
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;
      }
    }
  }
}
```

(b)

Fig. 6. A portion of the EJB interface grammar that specifies the transactional interface constraints. (a) is the grammar for non-recursive transactions; (b) is the grammar for recursive transactions.

```
procedure WITNESS(t)
    while stack ≠ t‖X do
        o‖stack ← stack
        if o ∈ Σ then fail
        else if o ∈ SP then
            if ¬o.apply() then
                fail
        else if o ∈ SA then
            o.apply()
        else
            productions ← table(o, t)
            viable ← {prod : (p, prod) ∈ productions
                          ∧ p.apply()}
            chosen ← choose(viable)
            stack ← chosen‖stack
    stack ← X
```

Fig. 7. Parsing algorithm

WITNESS is called with one terminal of lookahead. The variable *stack* (here global, in practice a member variable of the parsing class) is a list of terminals, nonterminals, semantic predicates and semantic actions. If the first symbol on the stack is the lookahead token, then we are done; pop it off and return. If the first symbol on the stack is a terminal that is not the lookahead symbol, then we have a problem; in this case the parse fails. If it is not a terminal, then perhaps it is a semantic predicate; in that case we verify that it returns true, and fail otherwise. If it is neither, perhaps it is a semantic action; in this case apply the semantic action and resume looping.

Finally, it could be a nonterminal. In this case, we examine the parse table, looking for a production of o given a lookahead of t. This gives a set containing predicate, production pairs; the idea here is that if the predicate returns false, then that production is not available at this time. Accordingly we filter all unavailable productions from the list, and then choose one. The production is then prepended to the stack, as in standard table parsing.

This algorithm requires a modified LL(1) table; we describe how to compute this next.

3) Constructing the parse table and computing the first and follow sets: To perform the above parsing algorithm we need the LL(1) parser table. 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 $G = \langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S \rangle$ our parsing table is a function $t : \mathbf{NT} \times \mathbf{\Sigma} \to \mathcal{P}(\mathbf{SP} \times \mathbf{P})$.

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. To accommodate the nondeterministic choose operator, we permit multiple entries in each cell in the parsing table. Thus, 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 IV-A2.

To compute the parse table t, we need two auxiliary functions *first* and *follow* [14], which we compute using the algorithms shown in Fig.s 8 and 9. Because we are dealing with code that writes code, we introduce some conventions to make our presentation simpler:

- $\langle\!\langle \text{x} \rangle\!\rangle$ means "code that will output x".
- $[\![x]\!]$ is the predicate that, when evaluated, computes x.
- [] is the empty list.
- $\square$ is the end of input token.
- $\langle\!\langle \ldots \$\text{x} \ldots \rangle\!\rangle$ 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 [14], 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 IV-B.

These algorithms merit some brief discussion. Our approach for generating these sets is standard LL(1) save that we attach a predicate to each production, informing us whether that production is available. These productions are computed during *first* set computation. These predicates are by default $[\![\textbf{true}]\!]$; that is, all productions default to available all the time. When we encounter a semantic predicate, we compute the conjunction of that semantic predicate and our current running predicate; if the semantic predicate fails, that production is not, in fact, available. If we encounter a semantic action, however, we must stop these conjunctions; the environment for the semantic predicate will not necessarily be the same and so we cannot continue tracking these predicates.

After the productions have been computed, we merely have to respect them during *follow* set computation. However we have a new complication that arises during *follow* set computation; we permit ambiguity in our parse table. In practice this means that when the normal LL(1) table algorithm would conclude that two or more entries need to go in one table cell and signal an error, we put them all in a set, put that set in the table cell, and continue execution.

procedure FIRST(G, **out** *first*)
  $\langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S\rangle \leftarrow G$
  first $\leftarrow \{(n, \emptyset) : n \in \mathbf{NT}\} \cup \{([], \{([\![\mathbf{true}]\!], \epsilon)\})\}$
  repeat
    for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ **do**
      if $P \neq X \rightarrow \epsilon$ **then**
        $s \leftarrow$ *first*$(Y_1 Y_2 \ldots Y_n)$
        $p \leftarrow [\![\mathbf{true}]\!]$
        $c \leftarrow$ **true**
        for all Y_i **do**
          $d \leftarrow$ **false**
          if $Y_i \in \mathbf{SP}$ **then**
            if c **then**
              $p \leftarrow p \wedge Y_i$
          else if $Y_i \in \mathbf{SA}$ **then**
            $c \leftarrow$ **false**
          else if $Y_i \in \mathbf{\Sigma}_\circ$ **then**
            target $\leftarrow$ *first*(Y_i)
            if $\epsilon \in$ *target* **then**
              putative $\leftarrow \{(q,t) : (q,t) \in$ *target* $\wedge\, t \neq \epsilon\}$
              if $s \neq$ *putative* $\wedge\, s \neq$ *target* **then**
                first$(Y_1 Y_2 \ldots Y_n) \leftarrow$ *putative*
            else
              if $s \neq$ *target* **then**
                first$(Y_1 Y_2 \ldots Y_n) \leftarrow$ *target*
              $d \leftarrow$ **true**
          else
            if $Y_i \notin s$ **then**
              first$(Y_1 Y_2 \ldots Y_n) \leftarrow$ *first*$(Y_1 Y_2 \ldots Y_n) \cup \{(p, Y_i)\}$
            $d \leftarrow$ **true**
          if $\neg d$ **then**
            first$(Y_1 Y_2 \ldots Y_n) \leftarrow$ *first*$(Y_1 Y_2 \ldots Y_n) \cup \{(p, \epsilon)\}$
      for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ **do**
        first$(P) \leftarrow$ *first*$(P) \cup$ *first*$(Y_1 Y_2 \ldots Y_n)$
    until no element in *first* has changed

Fig. 8. Algorithm for computing *first* sets

4) 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 $\langle\!\langle \mathrm{x} \rangle\!\rangle$ 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.

This Java construct creates a new, anonymous subclass of `Closure` with a new method `apply` overriding the method in `Closure`; it then constructs a single object of this subclass. It achieves precisely our aims; from this we get a data structure with an `apply` method that executes the code we desire.

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.

procedure FOLLOW(G,*first*, **out** *follow*)
  $\langle \mathbf{NT}, \mathbf{\Sigma}_\circ, \mathbf{Q}, \mathbf{SA}, \mathbf{SP}, \mathbf{P}, S\rangle \leftarrow G$
  follow $\leftarrow \{(n, \emptyset) : n \in \mathbf{NT}\} \cup \{([], \{([\![\mathbf{true}]\!], \Box)\})\}$
  repeat
    for all productions $P = X \rightarrow Y_1 Y_2 \ldots Y_n$ **do**
      for $i \leftarrow 1 \ldots n$ **do**
        if $Y_i \in \mathbf{NT}$ **then**
          if $i = n$ **then**
            $s \leftarrow []$
            $f \leftarrow \{([\![\mathbf{true}]\!], \epsilon)\}$
          else
            $s \leftarrow Y_{i+1} \ldots Y_n$
            $f \leftarrow$ *first*(s)
            if $\exists p : (p, \epsilon) \in f$ **then**
              $f \leftarrow f \cup$ *follow*(X)
          follow$(Y_i) \leftarrow$ *follow*$(Y_i) \cup f$
  until no element in *follow* has changed

Fig. 9. Algorithm for computing *follow* sets

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 all *declaration* $\in$ *visibleSymbolsBefore* **do**
  ⟪$(*declaration.type*), $(*declaration.name*)
    `=symbols.get(`$(*declaration.id*)`)`⟫
⟪$*code*⟫
for all *declaration* $\in$ *visibleSymbolsAfter* **do**
  ⟪`symbols.put(`$(*declaration.id*)`,`
    $(*declaration.name*)`)`⟫

We must also have opening scopes, closing scopes, and binding in our interface grammar; fortunately we have already accommodated this need, and Fig. 5 includes the necessary mapping.

B. 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;
}
```

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 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.

V. 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.

A. 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, 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 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 language 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.

B. *Persistence* API *clients*

The normal life cycle of a Persistence API client is to use a global `EntityManagerFactory` to retrieve 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 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 relevant interface: `EntityManagerFactory`, `EntityManager` and `EntityTransaction`. Portions of these grammars are shown in Fig. 6. 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 back end is some 64,000 lines of Java code.

C. *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.

To increase legibility, we give the full test name here once, and refer to an abbreviation in future. We 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.testPersistNone-Generator` (**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.testMergeAndBidirOneTo-One` (**Merge**) verifies that the bidirectional association works even with detached objects.
- `CallbacksTest.testCallBackListeners-Heirarchy` (**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.testEntityNotFound-Exception` (**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.testAlways-TransactionalOperations` (**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 we can verify that our interface represents a valid implementation of the EJB Persistence API by running all the test cases as written, we can also use the test cases that incorrectly use the API to simulate an incorrect client. We do so by modifying the test cases to rethrow any exceptions they catch. In our results in Tables I through IV, we 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. Accordingly we 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.

These bounds represent constraints on the maximum number of repetitions, not the number of repetitions itself. That is, when we report the run time for, say, **Get** running with 3 objects and 5 repetitions, we are 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.

We 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, we have excerpted representative results into Tables I and II, and present several graphs showing selected results for "correct" execution in Fig. 10 through Fig. 12. We present the number of states JPF has visited as a more representative measure of memory consumption; standard memory allocation algorithms request only large blocks of memory from the operating system, and so these figures are too coarse grained for out purposes. We give timing and state data for "incorrect" execution in Tables III and IV and show some graphs displaying timing results for "incorrect" execution in Fig. 13; we have omitted the state graphs for "incorrect" execution because, as shown in Table IV, there is no increase in the number of states processed in order to detect an error in any of those tests.

Our 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 we halt verification the moment an error is detected it is difficult to predict the run time or memory usage.

The expectation we 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 we would see the run time follow a doubly quadratic curve $O(n^2m^2)$. Because JPF stores visited states indefinitely, we would also expect the memory usage to increase by this same parameter.

We find that this expectation is upheld in all our 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 we cannot control for; for example variances in start up time, cool caches or disk access.

Our results also demonstrate the efficiency of our 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. An 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 we have reduced the execution time required to perform API operations to approximately the same time required for object allocation.

VI. Discussion and Related Work

This paper extends the results reported in [17]. In particular, in this paper we provide a formal semantics for our interface grammar specification language (discussed in Section II-A). Furthermore, we provide a significantly larger set of experiments in Section V, where each test case is parametrized with respect to the number of operations performed and the number of objects created.

Below, we first discuss the limitations of our approach and possible extensions to overcome these limitations. Following that, we discuss related work in grammar-based testing and interface specification and environment generation.

A. Extensions to Interface Grammars

One limitation of our current interface language is that it does not handle call-backs, i.e., we do not provide support for

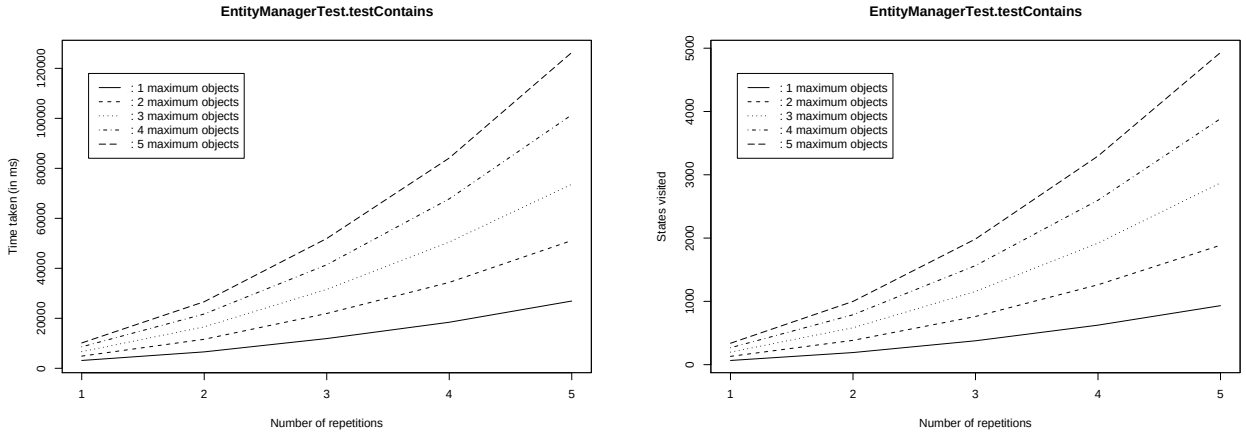

Fig. 10. Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **Cont**

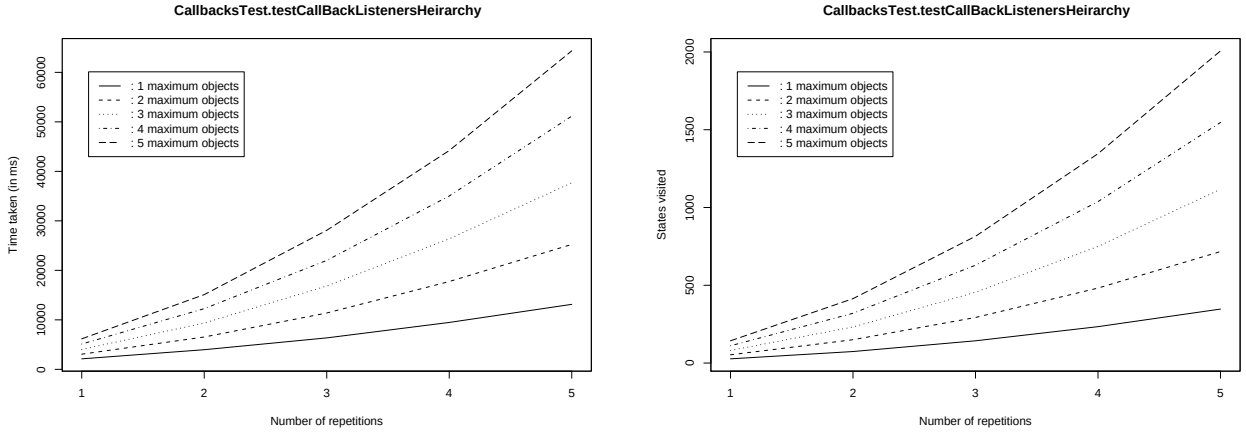

Fig. 11. Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **CBack**

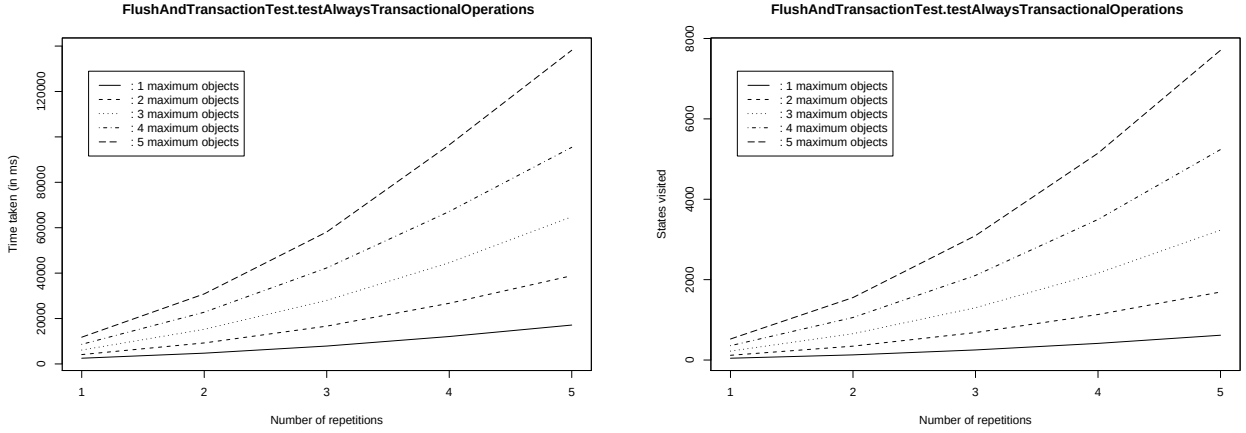

Fig. 12. Run time and state count vs. maximum number of repetitions and maximum number of objects for "correct" execution of **Trans**

TABLE I
"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

TABLE II
"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 III
"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 IV
"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

tracking call sequences in both directions. Note that allowing call-backs can result in situations where one or more method calls from the main program to the component we are modeling are pending while another method call from the main program to the component is initiated. Although this is an important limitation, it has not prevented us from analyzing the EJB Persistence API, nor would it prevent us from analyzing a SAX- or DOM-based XML library, a SOAP library, and many

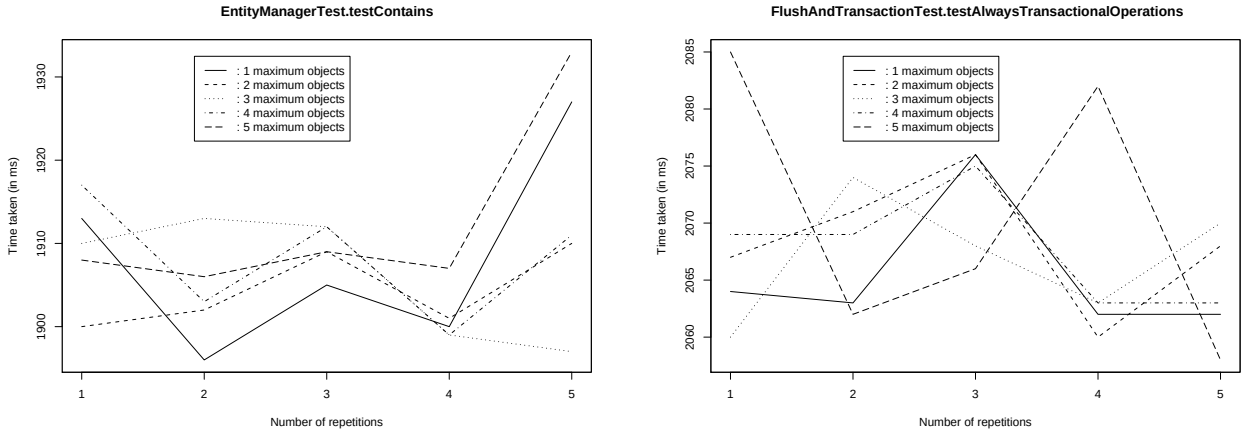

Fig. 13. Run time for "incorrect" execution for **Cont** and **Trans**

other applications. However, this restriction would prevent us from analyzing Swing listeners for example. We are currently working on a version of the interface grammar language and compiler that will accommodate interfaces with call-backs, and will thus be able to model this larger set of interfaces.

In this paper, we have discussed client-side verification; that is, we can check the correctness of the program assuming a correct implementation of the component. It would be desirable to be able to check correctness in both directions, that is to check the correctness of the program assuming a correct implementation of the component, and to check the correctness of the component assuming a correct implementation of the program. There are two major approaches to this goal:

1) Using our current approach, write two interface grammars, one for each direction, and use two separate verification steps to check each direction modularly. This can be done with the interface grammar and compiler we have presented in this paper; however, one must be careful to ensure that each interface is consistent with the other.
2) Extend our interface specification language and compiler to allow bidirectional interface grammars, and then write a single bidirectional interface grammar. We are currently working on extending our interface specification language in this direction as well.

In this paper we have not addressed generating or verifying the data associated with each method call, concentrating instead on control flow. Generating or verifying recursive data presents many of the same difficulties as analyzing recursive control flow; accordingly, it would be desirable to bring the power of interface grammars to bear in this task as well. We have extended our interface grammar language and compiler [18] by adding parameters to non-terminals, which permits specification of recursive data structures using grammar rules in precisely this fashion.

B. 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 [19] presented a fast algorithm for generating the minimal set of test cases required to achieve production coverage of a grammar, targeting parsers specifically. Because our grammars are interactive, this algorithm cannot be directly applied, but in future work we intend to adapt it.

Lämmel and Schulte [20] describe several techniques for limiting the combinatorial explosion of grammar based testing. Their technique uses a grammar to generate test cases; our 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 our stubs; one cannot achieve even full production coverage (which they call rule coverage) if the host program does not cooperate.

Maurer [21], [22] 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; our 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 lexical scoping of his variables, as we must with ours.

Offutt, Ammann and Liu [23] describe how mutation testing can be regarded as a type of grammar based testing, and give several useful coverage algorithms. We do not consider mutation at this time, and in any case their technique concentrates on test cases whereas ours is interactive.

Bauer and Finger [24] generate test cases using a regular grammar, which is strictly less powerful than ours and cannot accommodate recursion. As well, their technique generates non-interactive test cases.

Duncan and Hutchison [25] use attributed grammars to generate test cases. There are similarities in their attributed grammars and our interface grammars; for example, we both permit run-time guards, and in many cases their inherited and synthesized attributes can be made equivalent to our rule parameters. However their technique remains focused on test case generation and ours on interactive stubs.

Sirer and Bershad [26] 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 we can mimic their certificates using our semantic predicates; nonetheless in future work we intend to include some form of test certificate in our tool.

C. Interface Specification and Environment Generation

The use of finite state machines for specification, verification and extraction of interfaces have been studied extensively [27]–[32]. 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 Whaley et al. [29] or Alur et al. [30] 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 [6], [33]. 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 [31], [32], [34], 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.

Environment generation is a critical problem for achieving modularity in software model checking and has been studied before. Godefroid et al. [13] 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 [35] investigate 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.

Tkachuk et al.'s [36] Bandera environment generator 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.

Finally, it would be worthwhile to investigate restricted classes of interface grammars for efficient verification. For example, some closure properties that are undecidable for context free languages are decidable for *visibly pushdown languages* [37]. These results can be useful for some interface analysis problems if the interface grammar can be characterized as a visibly pushdown grammar. Also, it would be interesting to investigate applicability of the results on verification of pushdown systems (e.g., [38], [39]) to verification with interface grammars.

VII. 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.

References

[1] E. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts: The MIT Press, 1999.

[2] G. J. Holzmann, *Design and validation of computer protocols*. New Jersey: Prentice Hall, 1991.

[3] K. L. McMillan, *Symbolic model checking*. Massachusetts: Kluwer Academic Publishers, 1993.

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

[5] W. Visser, K. Havelund, G. Brat, and S. Park, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[6] T. Ball and S. K. Rajamani, "Automatically validating temporal safety properties of interfaces," in *Proceedings of the SPIN Workshop*, 2001, pp. 103–122.

[7] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of c code." in *NDSS*. The Internet Society, 2004.

[8] 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.

[9] 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*, December 2002. [Online]. Available: citeseer.ist.psu.edu/musuvathi02cmc.html

[10] 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)*. IEEE Computer Society, 2000, p. 3.

[11] 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.

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

[13] 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)*, 1998, pp. 345–357.

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

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

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

[17] 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)*, 2007, pp. 39–49. [Online]. Available: x-yojimbo-item://CC3D1214-37E5-4BFA-9D7F-D22F43161CEC

[18] ——, "Extended interface grammars for automated stub generation," in *Proceedings of the Automated Formal Methods Workshop (AFM 2007)*, 2007.

[19] P. Purdom, "A sentence generator for testing parsers," *BIT*, vol. 12, no. 3, pp. 366–375, 1972.

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

[21] P. M. Maurer, "Generating test data with enhanced context-free grammars," *IEEE Software*, vol. 7, no. 4, pp. 50–55, 1990.

[22] ——, "The design and implementation of a grammar-based data generator," *Software Practice and Experience*, vol. 22, no. 3, pp. 223–244, March 1992.

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

[24] J. A. Bauer and A. B. Finger, "Test plan generation using formal grammars," in *Proceedings of the 4th International Conference on Software Engineering*, Munich, Germany, September 1979, pp. 425–432.

[25] 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*, New York, NY, USA, March 1981, pp. 170–178.

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

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

[28] 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)*, 2002, pp. 428–441.

[29] 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.

[30] 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.

[31] 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)*, 2004, pp. 248–257.

[32] 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)*, 2005, pp. 750–759.

[33] T. Ball and S. K. Rajamani, "SLIC: A Specification Language for Interface Checking," Microsoft Research, Tech. Rep. MSR-TR-2001-21, January 2002.

[34] 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.

[35] 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)*, 2003, pp. 188–197.

[36] 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)*, 2003, pp. 116–129.

[37] Alur and Madhusudan, "Visibly pushdown languages," in *STOC: ACM Symposium on Theory of Computing (STOC)*, 2004.

[38] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: Application to model-checking," in *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR'97)*, 1997, pp. 135–150.

[39] R. Alur, K. Etessami, and P. Madhusudan, "A temporal logic of nested calls and returns," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004.*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 467–481. [Online]. Available: http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=2988&spage=467