Extended Interface Grammars for Automated Stub Generation*

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

ABSTRACT

An important challenge in software verification is the ability to verify different software components in isolation. Achieving modularity in software verification requires development of innovative interface specification languages. In this paper we focus on the idea of using grammars for specification of component interfaces. In our earlier work, we investigated characterizing method call sequences using context free grammars. Here, we extend this approach by adding support for specification of complex data structures. An interface grammar for a component specifies the sequences of method invocations that are allowed by that component. Our current extension provides support for specification of valid input arguments and return values in such sequences. Given an interface grammar for a component, our interface compiler automatically generates a stub for that component that 1) checks the ordering of the method calls to that component, 2) checks that the input arguments are valid, and 3) generates appropriate return values based on the interface grammar specification. These automatically generated stubs can be used for modular verification and/or testing. We demonstrate the feasibility of this approach by experimenting with the Java Path Finder (JPF) using the stubs generated by our interface compiler.

1. INTRODUCTION

Modularity is key for scalability of almost all verification and testing techniques. In order to achieve modularity, one has to isolate different components of a program during verification or testing. This requires replacement of different components in a program with stubs that represent their behavior. Our work on interface grammars originates from the following observation: If we can develop sufficiently rich interface specification languages, it should be possible to automatically generate stubs from these rich interfaces, enabling modular verification and testing.

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

In a recent paper [11] we proposed interface grammars as an interface specification language. 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. We built an interface compiler that takes the interface grammar for a component as input and generates a stub for that component. The resulting stub is a table-driven parser generated from the input interface grammar. Invocation of a method within the component becomes the lookahead symbol for the stub/parser. The stub/parser uses a parser stack, the lookahead, and a parse table to guide the parsing. The interface grammar language proposed in [11] also supports specification of semantic predicates and actions, which are Java code segments that can be used to express additional interface constraints. 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.

Although the interface grammar language proposed in [11] provides support for specification of allowable call sequences for a component, it does not directly support constraints on the input and output objects that are passed to the component methods as arguments or returned by the component methods as return values. In this paper we investigate the idea of using grammar production rules for expressing constraints on object validation and creation. Our approach builds on shape types [10], a formalism based on graph grammars, which can be used for specification of complex data structures. We show that grammar productions used in shape types can be easily integrated with grammar productions in interface grammars. In order to achieve this integration we allow nonterminals in interface grammars to have arguments that correspond to objects. The resulting interface specification language is capable of expressing constraints on call sequences, as well as constraints on input and output data that is received and generated by the component.

Our work is significantly different from earlier work on interface specification. Most of the earlier work on interfaces focuses on interface specification formalisms based on finite state machines [5, 4, 20, 2, 3]. More expressive interface specification approaches such as the ones based on design by contract [12, 8] are less amenable to automation. Moreover, it is not easy to express control flow related constraints,

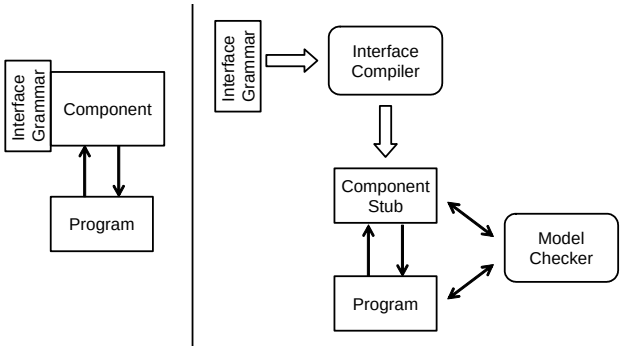


Figure 1: Modular verification with interface grammars.

such as the ones relating to call sequencing, as pre and post-conditions. We believe that the extended interface grammar specification language presented in this paper provides a unique balance between automation and expressiveness, and also enables specification of control flow and data structure constraints in a uniform manner.

There has been earlier work in grammar based testing such as [6, 13, 14, 15]. The common goal in these papers is to automatically generate test inputs using a grammar that characterizes the set of possible inputs. In contrast, in our work we use grammars as interface specifications where terminals correspond to method calls.

The rest of the paper is organized as follows. Section 2 gives an overview of interface grammars and shape types using a tree component as a running example. Section 3 includes a more detailed discussion on shape types. Section 4 discusses integration of concepts from shape types to interface grammars and how they can be used in object generation. Section 5 discusses and contrasts object validation versus object generation. Section 6 presents experiments demonstrating the use of interface grammars for modular verification of EJB clients using the Java PathFinder (JPF) [16] model checker. Section 7 concludes the paper.

2. EXTENDING INTERFACE GRAMMARS

The modular verification approach based on interface grammars is shown in Figure 1. Interface grammars provide a language for specification of component interfaces. The core of an interface grammar is a set of production rules that define a Context Free Grammar (CFG). This CFG specifies all acceptable method call sequences for the given component. Given an interface specification for a component, our interface compiler generates a stub for that component. This stub is a table-driven top-down parser [1] that parses the sequence of incoming method calls (i.e., the method invocations) based on the CFG defined by the interface specification.

As an example, consider a general tree with first-child and right-sibling pointers; for example, an XML Domain Object Model tree. If we add redundant left-sibling and parent pointers to the tree, we can then write a tree cursor that can traverse the tree, with the methods `moveup`, `movedown`, `moveleft` and `moveright`. We may wish to examine traversal algorithms independent of any particular tree representation, and thus want a component for this cursor. We can represent the cursor navigation operations using the following simplified interface grammar:

$$
\begin{array}{lcl}
\mathit{TreeCall} & \rightarrow & \texttt{movedown}\ \mathit{TreeCall} \\
& | & \texttt{movedown}\ \mathit{TreeCall}\ \texttt{moveup}\ \mathit{TreeCall} \\
& | & \texttt{moveright}\ \mathit{TreeCall} \\
& | & \texttt{moveright}\ \mathit{TreeCall}\ \texttt{moveleft}\ \mathit{TreeCall} \\
& | & \epsilon
\end{array}
$$

This is a context free grammar with the nonterminal symbol *TreeCall* (which is also the start symbol) and terminal symbols `moveup`, `movedown`, `moveright`, and `moveleft`. Each terminal symbol corresponds to a method call, and the above grammar describes the allowable call sequences that are supported by the component. We have restricted the permissible call sequences as follows: it is always an error to have more `moveup` symbols than `movedown` (corresponding to trying to take the parent of the root which may result in dereferencing a null pointer) and it is always an error to have more `moveleft` symbols than `moveright` at any given height (corresponding to trying to take the left sibling of the first child which may again result in dereferencing a null pointer). In our framework, this language corresponds to the set of acceptable incoming call sequences for a component, i.e., the interface of the component. Note that the set of acceptable incoming call sequences for the above example cannot be recognized by a finite state machine since the matching of `movedown` and `moveup` symbols, and `moveleft` and `moveright` symbols cannot be done using a finite state machine. The expressive power of a context free grammar is necessary to specify such interfaces. One could also investigate using extended finite state machines to specify such interfaces. However, we believe that grammars provide a suitable and intuitive mechanism for writing interface specifications.

Given the above grammar we can construct a parser which can serve as a stub for the Tree 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. In [11] we described such an interface grammar compiler that, given an interface grammar for a component, automatically constructs a stub/parser for that component.

Now, assume that, for the above tree example, we would also like to specify a `getTree` method that returns the tree that is being traversed. This is a query method and it can be called at any point during execution, i.e., there is no restriction on the execution of the `getTree` method as far as the control flow is concerned. However, the return value of the `getTree` method is a specific data structure. It would be helpful to provide support for specification of such data structures at the interface level. Our goal in this paper is to extend our interface grammar specification language to provide support for specification of such constraints. Such constraints can be used to specify the structure of the objects that are passed to a component or returned back from that component.

Consider the interface grammar below which is augmented

by a set of recursive rules that specify the structure of the tree that `getTree` method returns:

```
1   TreeCall      →  movedown TreeCall
2                 |  movedown TreeCall moveup TreeCall
3                 |  moveright TreeCall
4                 |  moveright TreeCall moveleft TreeCall
5                 |  getTree TreeGen x TreeCall
6                 |  ϵ
7   TreeGen x     →  N x null
8   N x y         →  leftc x z, parent x y, N z x, L x y
9                 |  leftc x null, parent x y, L x y
10                |  leftc x null, parent x null
11  L x y         →  rights x z, N z y, L z y
12                |  rights x null
```

The productions 1-4 and 6 are the same productions we used in the earlier interface grammar. The production 5 represents the fact that the modified interface grammar also accepts calls to the `getTree` method. The nonterminal *TreeGen* is used to define the shape of the tree that is returned by the `getTree` method. Productions 7-12 define a *shape type* based on the approach proposed by Fradet and le Métayer [10]. Shape types are based on graph grammars and are used for defining shapes of data structures using recursive rules similar to CFGs.

Before we discuss the shape types in more detail in Section 3, we would like to briefly explain the above example and the data structure it defines. The nonterminals *TreeGen*, *N* and *L* used in the production rules 7-12 have arguments that are denoted as *x, y, z*. Arguments *x, y, z* represent the node objects in the data structure. In this example, the data structure is a left-child, right-sibling (LCRS) tree. In this data structure, each node has a link to its leftmost child, its immediate right sibling, and its parent if they exist, otherwise these fields are set to null. The terminal symbols **leftc**, **rights** and **parent** denote the fields that correspond to the left-child, right-sibling, and parent of a node object, respectively. Each production rule expresses some constraints among its arguments in its right hand side, and recursively applies other production rules to express further constraints. For example **parent** *x y* means that the **parent** field of node *x* should point to node *y*. Similarly, **rights** *x* null means that the **rights** field of node *x* should be null.

In Figure 2 we show an example LCRS tree. Let us investigate how this tree can be created based on the production rules 7-12 shown above. Production 7 states that a LCRS tree can be created using one of the production rules for the nonterminal *N* and by substituting the node corresponding to the root of the tree (i.e., node 1 in Figure 2) for the first argument and null for the second argument. Let us pick production 8 for nonterminal *N* and substitute node 1 for *x*, null for *y* and node 2 for *z*. Based on this assignment, the constraints listed in the right hand side of production 8 state that: **leftc** field of node 1 should point to node 2; **parent** field of node 1 should be null; nodes 2 and 1 should satisfy the constraints generated by a production rule for nonterminal *N* where the first argument is set to node 2 and the second argument is set to node 1; and, node 1 and null should satisfy the constraints generated by a production rule for nonterminal *L* where the first argument is set to node 1 and the second argument is set to null. Note that, the first two constraints are satisfied by the tree shown in Figure 2. The last constraint is satisfied by picking the production rule 12, which states that the **rights** field of node 1 should be null, which is again satisfied by the tree shown in Figure 2. Finally, the third constraint recursively triggers another application of the production 8 where we substitute node 2 for *x*, node 1 for *y*, and null for *z*. By recursively applying the productions rules 7-12 this way, one can show that the tree shown in Figure 2 is a valid LCRS tree based on the above shape type specification.

The above example demonstrates that we can use shape types for object validation. Object validation using shape types corresponds to parsing the input object graphs based on the grammar rules in the shape type specification. Note that we can use shape types for object generation in addition to object validation. In order to create object graphs that correspond to a particular shape type we can randomly pick productions and apply them until we eliminate all nonterminals. Resulting object graph will be a valid instance of the corresponding shape type. In fact, in the above example, our motivation was to use the shape type formalism to specify the valid LCRS trees that are returned by the `getTree` method.

We would like to emphasize that, although we will use data-structures such as LCRS tree as running examples in this paper, our goal is not verification of data structure implementations, or clients of data structure libraries. Rather, our goal is to develop a framework that will allow verification of arbitrary software components in isolation. This requires an interface specification mechanism that is capable of specifying the shapes of the objects that are exchanged between components as method arguments or return values. Our claim is that extended interface grammars and our interface compiler provide a mechanism for isolating components which enables modular verification.

In the following sections we will discuss shape types in more detail and discuss how to integrate them to our interface grammar specification language. We will also demonstrate examples of both object validation and generation with our extended interface grammar specification language based on shape types.

We note one weakness of the above interface specification example. According to the above interface grammar specification, the tree that is returned by the `getTree` method may not be consistent with the previous calls to the `moveup`, `movedown`, `moveleft` and `moveright` methods that have been observed. For example, if a client calls the `movedown` method twice followed by two calls to the `moveup` method, then if the next call is `getTree`, the `getTree` method should return a tree of height greater than or equal to two to be consistent with the observed call history. However, the above specification does not enforce such a constraint. The `getTree` method can return any arbitrary LCRS tree at any time. Our interface specification language is capable of specifying this type of constraints (i.e., making sure that the the tree returned by the `getTree` method is consistent with the past call history to the tree component) using semantic predicates and actions.

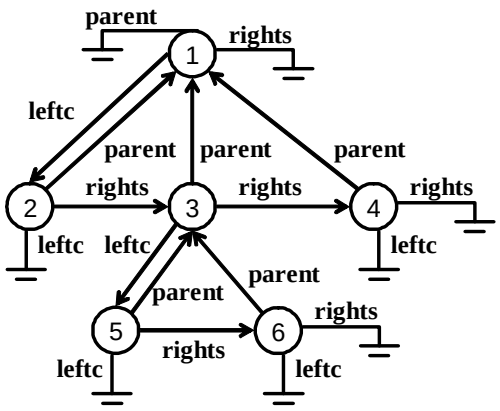


Figure 2: An example left-child, right-sibling (LCRS) tree.

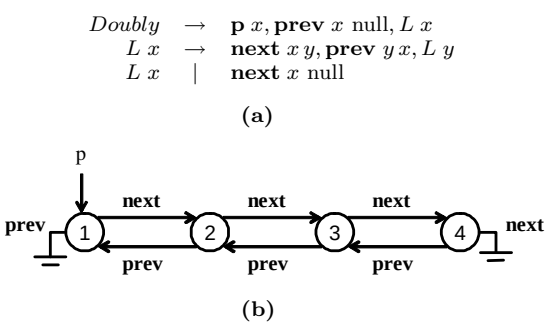


Figure 3: (a) Shape type for a doubly linked list, and (b) an example linked list of that type

3. SHAPE TYPES

A weakness of the interface specification language defined in our previous work [11] is that it does not provide direct support for describing the data associated with the method calls and returns of a component, i.e., the arguments and return values for the component methods. However, the interface specification language presented in [11] allows specification of semantic predicates and actions. This enables the users to insert arbitrary Java code to interface specifications. These semantic predicates and actions can be treated as nonterminals with epsilon-productions and the Java code in them are executed when the corresponding nonterminal appears at the top of the parser stack. The user can do object validation and generation using such semantic predicates and semantic actions. However, this approach is unsatisfactory for the same reason that hand writing a component stub in Java directly is unsatisfactory; it is frequently brittle and difficult to understand. Accordingly, we would like to extend our interface grammars to support generating and validating data, and to do so in a way that preserves the advantages of grammars.

The shape types of Fradet and le Métayer [10] define an attractive formalism based on graph grammars that can be used to express recursive data structures. We have been inspired by their formalism, but to accommodate the differences between their goal and ours our implementation becomes substantially different. Nonetheless, it is worthwhile explaining Fradet and le Métayer's shape types and then explaining how our approach differs syntactically before explaining our implementation.

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

Consider the language of strings $(\textbf{name}\ x\ y)^*$, where **name** is some string and x and y are integers. If we regard x and y as vertices, then we can obtain a labeled directed graph from any such string by regarding the string $\textbf{name}\ x\ y$ as defining an edge from the vertex labeled x to the vertex labeled y, itself labeled **name**. If we further regard the vertices in this graph as representing objects and the edges as representing fields, we can obtain an object graph. Note that this mapping is not 1-1: if the strings are reordered the same graph is obtained.

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

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

Shape types provide a powerful formalism for specification of object graphs. In Figure 3(a) we show the shape type for a doubly linked list and in Figure 3(b) we show an example doubly linked list of that type. In Figure 4(a) we show the shape type for a binary tree and in Figure 4(b) an example binary tree of that type.

4. OBJECT GENERATION WITH INTERFACE GRAMMARS

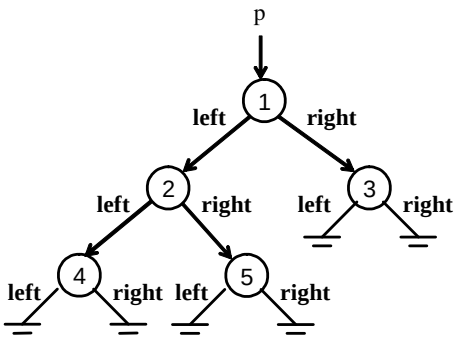
In this section we will discuss how we integrate shape types to our interface grammar specification language. First, we start with a brief discussion on alternative ways of generating arbitrary object graphs in a running Java program. Next, we give an overview of our extended interface grammar language and discuss how this extended language supports shape types. We conclude this section by presenting an example interface grammar for the left-child, right-sibling (LCRS) tree example discussed in Section 2.

4.1 Creating Object Graphs

There are three major techniques for object graph creation: with JVM support, serialization, and method construction. The first technique uses support from the JVM to create

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

(a)



(b)

Figure 4: (a) Shape type for a binary tree, and (b) an example binary tree of that type

objects arbitrarily and in any form desired. Visser et al. [18] use this technique, extending the Java PathFinder model checker appropriately. While this is very powerful, it is necessarily coupled to a specific JVM and can be easy to inadvertently create object structures that cannot be recreated by a normal Java program. We reject this approach because we do not want to be overly coupled to a specific JVM.

The second technique uses the Java serialization technologies used by Remote Method Invocation (RMI) [19]. This is almost as powerful as the first technique and has the advantage of being more portable. Since the serialization format is standardized, it is relatively easy to create normal serialization streams by fiat. There are two major issues with this approach. First, it requires that all the objects that one might want to generate be serializable, which requires changing the source code in many cases. Second, it is possible for an object to arbitrarily redefine its serialization format or to add arbitrarily large amounts of extra data to the object stream. This is common in the Java system libraries. Accordingly we have rejected this approach as well.

The third approach, and the one we settled upon, is to generate object graphs through the object's normal methods. The main advantages this has is that it works with any object, it is as portable as the original program, and it is impossible to get an object graph that the program could not itself generate. The main disadvantage is that this approach cannot be fully automated without a specification of the object graph shapes that are valid. Since we do semiautomated analysis, we combine approach with the shape types of the previous section and ask the user to tell us what sort of shapes they desire.

4.2 Extended Interface Grammar Language

In addition to providing support for context free grammar rules, our interface specification language also supports spec-

```
(1)  main         →  class *

(2)  class        →  class CLASSID { item * }

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

(5)  rule         →  rule RULEID ( declaration * )
                       block

(6)  block        →  { statement * }

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

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

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

(18) sempred      →  ⟪ EXPR ⟫

(19) semexpr      →  ⟪ EXPR ⟫

(20) semact       →  ⟪ STATEMENT ⟫

(21) declaration  →  TYPE ID
```

Figure 5: Abstract syntax for the extended interface grammar language

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

In Figure 5 we show a (simplified) grammar defining the abstract syntax of our interface grammar language. We denote *nonterminal* and `terminal` symbols and Java CODE and IDENTIFIERS with different fonts. The symbols ⟪ and ⟫ are used to enclose Java statements and expressions. Incoming method calls to the component (i.e., method invocations) are shown with adding the symbol ? to the method name as a prefix. Outgoing method calls (i.e., method calls by the component) are shown with adding the symbol ! to the method name as a prefix. In the grammar shown in Figure 5, we use "*" to denote zero or more repetitions of the preceding symbol, and "?" to denote that the preceding symbol can appear zero or one times.

An interface grammar consists of a set of class interfaces (not to be confused with Java interfaces) (represented in rule (1) in Figure 5). The interface compiler generates one stub class

for each class interface. Each class interface consists of a set of semantic actions and a set of production rules that define the CFG for that class (rules (2), (3) and (4)). A *semantic action* is simply a piece of Java code that is inserted to the stub class that is generated for the component (rule (20)). A *rule* corresponds to a production rule in the interface grammar. Each rule has a name, a list of declarations, and a block (rule (5)). The use of declarations will be explained in Section 4. A rule block consists of a sequence of statements (rule (6)). Each statement can be a rule application, a semantic action, a declaration, a choose block, a method invocation, a method return or a method call (rules (7)-(14)). A semantic action corresponds to a piece of Java code that is executed when the parser sees the nonterminal that corresponds to that semantic action at the top of the parse stack. A *rule application* corresponds to the case where a nonterminal appears on the right hand side of a production rule. A *declaration* corresponds to a Java code block where a variable is declared and is assigned a value (rule (21)). A *choose block* is simply a switch statement (rules (11) and (15)). A selector for a switch case can either be a method invocation (i.e., an incoming method call), a semantic predicate or the combination of both (rules (16) and (17)). A switch case is selected if the semantic predicate is true and if the lookahead token matches to the method invocation for that switch case. A *method return* simply corresponds to a return statement in Java. When the component stub receives a method invocation from the program, it first calls the interface parser with the incoming method invocation, which is the lookahaed token for the interface parser. When the parser returns, the component stub calls the interface parser again, this time with the token which corresponds to the method return. Finally, a *method call* is simply a call to another method by the stub.

4.3 Support for Shape Types

We can obtain all the power required to embed the shape types of Section 3 into our interface grammars with the following addition: we permit rules to have parameters. Because we need to be able to pass objects to the rules as well as retrieve them, we have chosen to use call-by-value-return semantics for our parameters rather like the "in out" parameters of the Ada language. These parameters are reflected in the declaration list of line 5 of Figure 5, and in the identifier list of line 8 of that same figure. Because we have chosen uniform call-by-value-return semantics, only variable names may be supplied to `apply`.

Because our previous work required lexical scoping, the runtime needed only to be changed as follows: when encoding an `apply`, store the current contents of all its variables in a special location—we currently assign parameter n to variable $-(n+1)$, as all our variables have a nonnegative associated integer used in scoping—push the nonterminal onto the stack as normal, and afterward overwrite each variable with the result, again stored in the special location. That is, if $\langle\!\langle x \rangle\!\rangle$ is the closure performing x and $a_0, \ldots a_n$ is the list of arguments, then the series of grammar tokens corresponding to `apply` *rule* $(a_0, \ldots a_n)$ is

for $i = 0$ **to** n **do**
  $\langle\!\langle$`symbols.put ($(-(i+1)),`
    `symbols.get ($(`a_i`.id)));`$\rangle\!\rangle$
od
rule
for $i = 0$ **to** n **do**
  $\langle\!\langle$`symbols.put ($(`a_i`.id),`
    `symbols.get ($(-(i+1))));`$\rangle\!\rangle$
od

Similarly, for every production for a `rule`, the compiler must, at the start of the production, bind all its parameters from the special location; and at the end it must store the current values of each of its parameters to the appropriate place in the special location. For example, given a production $A\ a_0 \ldots a_n \rightarrow x_0 \ldots x_m$, the amended production would be as follows:

$A \rightarrow$ ($\langle\!\langle$`symbols.push ();`$\rangle\!\rangle$,
  for $i = 0$ **to** n **do**
    $\langle\!\langle$`symbols.bind ($(`a_i`.id));`$\rangle\!\rangle$,
    $\langle\!\langle$`symbols.put ($(`a_i`.id),`
      `symbols.get ($(-(i+1))));`$\rangle\!\rangle$
  od,
  $x_0, \ldots x_m$,
  for $i = 0$ **to** n **do**
    $\langle\!\langle$`symbols.put ($(-(i+1)),`
      `symbols.get ($(`a_i`.id)));`$\rangle\!\rangle$
  od,
  $\langle\!\langle$`symbols.pop ();`$\rangle\!\rangle$)

5. OBJECT GENERATION VS. OBJECT VALIDATION

Using the extended interface grammar specification language presented in Section 4 it is possible to specify both generation and validation of data structures, and to do so in a manner that is reminiscent of the shape types of Section 3. Object validation is used to check that the arguments passed to a component by its clients satisfy the constraints specified by the component interface. Object generation, on the other hand, is used to create the objects that are returned by the component methods based on the constraints specified in the component interface.

Figure 6, shows object generation and validation for doubly linked list and binary tree examples. Figure 6 contains three specifications for each of the two examples. At the top of the figure we repeat the shape type specifications for doubly linked list and binary tree examples from Section 3 for convenience. The middle of the figure contains the interface grammar rules for generation of these data structures. Note the close similarity between the shape type productions and the productions in the interface grammar specification. The bottom of the figure shows the interface grammar rules for validation of these data structures.

Object generation and validation tasks are broadly symmetric, and their specification as interface grammar rules reflects this symmetry as seen in Figure 6. While in object generation semantic actions are used to set the fields of objects to appropriate values dictated by the shape type specification, in object validation, these constraints are checked using semantic predicates specified as guards. Note that the set of nonterminals and productions used for object generation and validation are the same.

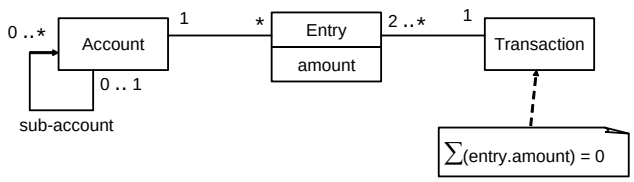


Figure 7: UML diagram of the Account pattern

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

Detecting aliasing among objects is necessary during object validation. Note that, since shape type formalism assumes that no aliasing should occur unless it is explicitly specified, during object validation we need to make sure that there is no unspecified aliasing. Instead of trying to enforce a fixed policy on aliasing, we leave the specification of the aliasing policy during object validation to the user. The typical way to check aliasing would be by using a hash-set as demonstrated by the two object validation examples shown in Figure 6. Note that, the interface grammar rules for object validation propagate the set of nodes that have been observed and make sure that there is no unspecified aliasing among them.

6. VERIFICATION WITH INTERFACE GRAMMARS

In this Section, we report some experiments on modular verification of Java programs using stubs automatically generated by our interface compiler. We use the model checker Java PathFinder (JPF) [17] as our verification tool. JPF is an explicit and finite state model checker that works directly on Java bytecode. It enables the verification of arbitrary pure Java implementations without any restrictions on data types. JPF supports property specifications via assertions that are embedded into the source code. It exhaustively traverses all possible execution paths for assertion violations. If JPF finds an assertion violation during verification, it produces a counter-example which is a program trace leading to that violation.

To analyze the performance of stubs automatically generated by our interface compiler, we have written several small clients for an Enterprise Java Beans [7] (EJB) persistence layer. We used a similar technique in our prior work [11]; here we handle some types of queries and perform relational integrity checks upon the resulting database.

We have chosen to base our clients around the Account pattern from Fowler [9]. Strictly speaking this is a pattern for an object schema; accordingly we have implemented it for these tests with the SQL mapping in the EJB framework. The Account pattern is useful for us because it represents structured data and also has a hierarchical element (accounts can have sub-accounts).

A brief description of the Account pattern and how we interpreted it is in order. A UML diagram illustrating all this can be seen in Figure 7. An *account* contains entries and can be a parent to other accounts; the account instances make up a forest. An *entry* is associated with exactly one account and exactly one monetary transaction, and has a field representing an amount of money. A *monetary transaction* is associated with at least two entries, and the sum of all entries in every monetary transaction must be zero at the end of a database transaction—this is often stated as "money is neither created nor destroyed." Since unfortunately the term 'transaction' here refers to two distinct concepts both of which are important to us, we must be explicit: in the absence qualification, 'transaction' always refers to a monetary transaction.

This structure possesses a number of natural invariants. We have already mentioned the key transaction invariant. Accounts and their children must possess the tree property; that is an account can not have two parents. The sum of all entries in all accounts in the system should also be zero; if it is not, we may have forgotten to store an account, an entry, or a transaction. Because we permit more than only two entries per transaction, our transactions are called multi-legged; it is usually considered undesirable or an outright error for one transaction to have more than one "leg" in any one account.

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

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

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

To exercise this stub, we have written four EJB Persistence

Shape Type Specification

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

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

Object Generation with Interface Grammars

```
rule genDoubly (Node x) {
  ⟪ x = new Node (); ⟫
  ⟪ x.setPrev (null); ⟫
  apply genL (x);
}
rule genL (Node x) {
  choose {
  case:
    Node y = ⟪ new Node () ⟫;
    ⟪ x.setNext (y); ⟫
    ⟪ y.setPrev (x); ⟫
    apply genL (y);
  case:
    ⟪ x.setNext (null); ⟫
  }
}
```

```
rule genBintree (Node x) {
  ⟪ x = new Node (); ⟫
  apply genB (x);
}
rule genB (Node x) {
  choose {
  case:
    Node y = ⟪ new Node (); ⟫;
    Node z = ⟪ new Node (); ⟫;
    ⟪ x.setLeft (y); ⟫
    ⟪ x.setRight (z); ⟫
    apply genB (y);
    apply genB (z);
  case:
    ⟪ x.setLeft (null); ⟫
    ⟪ x.setRight (null); ⟫
  }
}
```

Object Validation with Interface Grammars

```
rule matchDoubly (Node x) {
  Set nodesSeen = ⟪ new HashSet () ⟫;
  guard ⟪ x instanceof Node
    && !nodesSeen.contains (x) ⟫;
  ⟪ nodesSeen.insert (x); ⟫
  guard ⟪ x.getPrev () == null ⟫;
  apply matchL (x, nodesSeen);
}
rule matchL (Node x, Set nodesSeen) {
  choose {
  case ⟪ x.getNext () == null ⟫:
  case ⟪ x.getNext () != null ⟫:
    Node y = ⟪ x.getNext () ⟫;
    guard ⟪ y instanceof Node
      && !nodesSeen.contains (y) ⟫;
    ⟪ nodesSeen.insert (y); ⟫
    guard ⟪ x.getNext () == y ⟫;
    guard ⟪ y.getPrev () == x ⟫;
    apply matchL (y, nodesSeen);
  }
}
```

```
rule matchBintree (Node x) {
  Set nodesSeen = ⟪ new HashSet () ⟫;
  guard ⟪ x instanceof Node
    && !nodesSeen.contains (x) ⟫;
  ⟪ nodesSeen.insert (x); ⟫
  apply matchB (x, nodesSeen);
}
rule matchB (Node x, Set nodesSeen) {
  choose {
  case ⟪ x.getLeft () == null ⟫:
    guard ⟪ x.getRight () == null ⟫;
  case ⟪ x.getLeft () != null ⟫:
    Node y = ⟪ x.getLeft () ⟫;
    guard ⟪ y instanceof Node
      && !nodesSeen.contains (y) ⟫;
    ⟪ nodesSeen.insert (y); ⟫
    Node z = ⟪ x.getRight () ⟫;
    guard ⟪ z instanceof Node
      && !nodesSeen.contains (z) ⟫;
    ⟪ nodesSeen.insert (z); ⟫
    guard ⟪ x.getLeft () == y ⟫;
    guard ⟪ x.getRight () == z ⟫;
    apply matchB (y, nodesSeen);
    apply matchB (z, nodesSeen);
  }
}
```

Figure 6: Interface grammars for doubly linked list and binary tree generation and matching

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

Table 1: Run time and memory usage vs. number of entries

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

Table 2: Run time and memory usage vs. number of accounts

API clients, and have run the clients with varying parameters in the JPF model checker. Two clients are correct in their use of the database and we expect that JPF will report this. Two are incorrect; one triggers a fault almost immediately, and the other is only invalid some of the time. Our clients are as follows:

1. *deparent* takes an account and removes it from its parent.

2. *voider* selects a transaction and 'voids' it, by creating a new transaction negating the original transaction. This introduces new objects into the system.

3. *reparent* takes two entries in the system and trades their transactions.

4. *increaser* increases the monetary value of entries in the system.

Our results are presented in Tables 1 and 2. When there is an interface violation, JPF halts at the first assertion violation and reports an error. In the experiments reported in Table 1 we restricted the state space to a single account and we observed the change in the verification results with respect to increasing number of entries. In the experiments reported in Table 2 we restricted the number of entries and increased the number of accounts.

The *deparent* client removes parent of an account. Changing the parent of an account can cause cycles if done naïvely, but removing the parent is always safe. Since the *deparent* client does not violate any interface properties, JPF does not report any assertion violations for *deparent*. As the number of accounts and entries increases verifying this operation takes an exponentially increasing amount of time to complete due to exponential increase in the state space.

The *voider* client does not violate any interface properties and, hence, JPF does not report any assertions violations for *voider*. Since *voider* introduces new objects to the system it creates a larger state space and its verification takes a longer time than *deparent*.

The *reparent* swaps the transactions of two entries. If the entries encode the same monetary value this can be safe, but in the general case this operation will break the transaction invariant. An additional complication is that if there are less than four entries in the system, *reparent* cannot fail; there is only one transaction available. The time it takes for model checker to reach an assertion violation for *reparent* depends on the order the model checker explores the states.

However the proportion of the state space where *reparent* is valid decreases precipitously as the number of entries in the system increases. Accordingly we expect that the running time will eventually come to some equilibrium if we increase the number of entries, but will consume an exponentially increasing amount of time if we hold the number of entries constant and increase the number of accounts (as observed in Table 2.

Since the *increaser* client always increases the monetary values of the entries, it always violates the transaction invariant, with even two entries in the system. So it takes model checker approximately the same amount of time to report an assertion violation for the *increaser* client regardless of the size of the state space.

7. CONCLUSION

We presented an extension to interface grammars that supports object validation and object creation. The presented extension enables specification of complex data structures such as trees and linked lists using recursive grammar rules. The extended interface grammar specification language provides a uniform approach for specification of allowable call sequences and allowable input and output data for a component. Given the interface grammar for a component, our interface compiler automatically creates a stub for that component which can be used for modular verification or testing. We demonstrated the use of interface grammars for modular verification by conducting experiments with JPF using stubs automatically generated by our interface compiler.

8. REFERENCES

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

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

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

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

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

[6] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 170–178, 1981.

[7] Enterprise java beans 3.0 specification. Technical report, Sun Java Community Process, May 2006. JSR-000220.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245, 2002.

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

[10] P. Fradet and D. le Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM Press.

[11] 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. To appear.

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

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

[14] P. M. Maurer. The design and implementation of a grammar-based data generator. *Softw., Pract. Exper.*, 22(3):223–244, 1992.

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

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

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

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

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

[20] 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.