Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 110 (2004) 33–54

www.elsevier.com/locate/entcs

Using Off-the-Shelf Formal Methods to Verify Attribute Grammar Properties

Shirley Goldrei Anthony Sloane

Department of Computing
Macquarie University
Sydney, NSW 2109, Australia

Abstract

Attribute Grammars are the specification language of many tools that automatically generate programming language implementations. We consider the problem of verifying properties of attribute grammar specifications, particularly properties that are not well supported by existing tools. Rather than propose methods for extending existing tool implementation techniques, we propose the use of off-the-shelf formal methods tools as the basis for attribute grammar verification. Off-the-shelf tools can provide significant expressive power at a much lower cost than extending an existing evaluator generator. As a specific example, we describe how to use the Alloy model finding and checking tool to verify properties of remote attribution constructs in the LIDO attribute grammar specification language. A naive application of this approach has significant performance overheads; we discuss techniques for limiting the scope of the problems that are solved to make the approach tractable.

Keywords: Compiler generation, attribute grammars, formal methods, software verification.

1 Introduction

Attribute Grammars are a well-established formal notation for specifying the semantics of programming languages and structured text [1,13,14,16]. Many methods exist to generate efficient evaluators from attribute grammars [2,10].

Attribute evaluator generators typically focus on the core problem of producing an efficient evaluator from a user-supplied attribute grammar. Peripheral functionality that is potentially useful during development but not required to produce an evaluator is often omitted. Examples of this kind of functionality include browsers to show the attribute grammar from different

doi:10.1016/j.entcs.2004.06.011

perspectives or analyses for discovering or checking formal properties of grammars. In other current work we are addressing the former kind of functionality by developing integrated development environments for attribute grammars including the application of program slicing methods to facilitate viewing [18].

The present paper focuses on the second class of functionality: formal analysis of attribute grammar properties [3]. Of course, evaluator generators have to analyse various formal properties of the input grammar to verify that an evaluator can be produced or to produce a correct evaluator (e.g., attribute dependences, circularity). We are interested in providing analyses that augment the feedback provided by generators.

Our techniques can be applied to many different analyses. To keep our presentation concrete, we use as a running example the diagnosis of problems with remote attribution constructs in the LIDO specification language from the LIGA attribute grammar system [9]. The aim is to provide proper feedback when an INCLUDING attribution construct reaches up the abstract syntax tree (AST) to obtain an attribute value from an ancestor node, but where the ancestor node will not necessarily be present in the tree. In this situation we want to present the user with an example of the problem rather than just a notification that the error exists (LIGA's current behaviour). Section 2 provides more detail about LIDO and the INCLUDING problem.

Rather than modify existing attribute grammar systems to add extra feedback, we have chosen an approach based on an off-the-shelf formal methods tool. Formal methods can provide an expressive platform for describing software properties and assertions about them. The properties and assertions are independent of particular programming notations, so front-ends can be written to enable the analysis tool to be used with different languages. For many problems a formal methods tool can be used to achieve an effective implementation with much less effort than modifying an existing generator. This strategy also maintains the modularity of the overall system because the generator can focus on its task and the auxiliary tools can provide support for peripheral problems.

Our method is based on formal relational models as supported by the Alloy model finding and checking tool [6]. Section 3 summarises the relevant capabilities of Alloy and discusses its suitability for the class of problems we are addressing. A crucial aspect of the tool is its capability to provide *counterexamples* to illustrate violations of assertions.

Section 4 provides the details of our method consisting of the following main steps:

(i) The attribute grammar to be analysed is automatically translated into a *model specification* that includes both properties of grammars in general and properties of the specific grammar under analysis (Sections 4.1 and 4.2). In our example, the general properties include the fact that all grammars have a single root non-terminal and that non-terminals are related by parent and child relationships which are inverses of each other. Grammar-specific properties include the identity of the root non-terminal and the parent relationship between non-terminals as implied by the productions of the grammar.

(ii) The properties that we wish to check are expressed as *assertions* about a model that conforms to the model specification (Section 4.3). The assertions required are automatically derived from the attribute grammar. In our example, for each remote attribution construct, we assert that the ancestor node named in the construct must be reachable in all possible ASTs.

(iii) The model specification and the assertions are then passed to Alloy for automatic analysis. Alloy proceeds to generate models that conform to the specification and check that the assertions are true for these models. A model that doesn't satisfy the assertions constitutes a counterexample. In our example, a counterexample will consist of a model that describes a particular AST and a path through it that does not contain the ancestor node. We post-process the counterexample model from Alloy notation back into a path in user-level notation for presentation to the user in an error message.

Scalability is an important issue for formal models of software. An approach based on formal methods is no use if it cannot handle realistic inputs. It often turns out that the key to controlling scalability is to make sure that the model specification is not overly specific. For our example, this means that we must be careful not to try to model complete ASTs. Rather, we model paths through ASTs, which allows models to omit non-terminals that do not occur on the path being modelled. Section 5 evaluates the performance of our analysis on an attribute grammar that specifies the semantic analysis for a subset of Pascal and presents further modifications that significantly improve performance.

2 LIDO and the INCLUDING problem

LIDO is the specification language of the LIGA attribute grammar system [9]; in turn, LIGA is the major semantic analysis component of the Eli language

```
 1 ATTR value: int;
 2 RULE: dc ::= expr ’=’ COMPUTE
 3    dc.value = expr.value;
 4 END;
 5 RULE: expr ::= expr ’+’ term COMPUTE
 6    expr[1].value = ADD (expr[2].value, term.value);
 7 END;
 8 RULE: expr ::= term COMPUTE
 9    expr.value = term.value;
10 END;
11 RULE: term ::= digit COMPUTE
12    term.value = digit;
13 END;
14 SYMBOL dc COMPUTE
15    printf ("%d\n", THIS.value);
16 END;
```

Figure 1. Desk Calculator for Addition

processor generation system [4]. LIDO[1] is descended from the ALADIN language that was the specification language in the GAG system, a predecessor of LIGA [8]. LIDO and ALADIN have many features in common with the main difference being that LIDO only provides a general prefix-style for expressing computations whereas ALADIN was a full-featured functional programming language. Most notably, both of these languages have the INCLUDING construct which is at the core of the main example problem addressed in this paper.

2.1 Desk Calculator

To give a flavour of LIDO notation, Figure 1 shows a very simple attribute grammar for a desk calculator with only an addition operator. The grammar calculates the value of an input expression and prints it.

LIDO can introduce attributes independently of symbols using the ATTR keyword, so line 1 of Figure 1 specifies that the *value* attribute has integer type. Computations that define how to determine attribute values are given in association with either non-terminal symbols or rules. In Figure 1 computations of the *value* attribute of the *dc*, *expr* and *term* symbols are given in four rules with the productions that define their structure (lines 2–13). The use of the terminal *digit* in line 12 refers to the intrinsic value of the terminal as computed by the lexical analyser. The computation that prints the value of the *dc* symbol is associated with the symbol rather than with the rule defining *dc* because it does not need to refer to attributes of the descendants of the *dc* symbol (lines 14–16). Computations use a prefix notation to call functions defined outside the grammar notation. In this case, a library macro *ADD* is

[1] For full details of LIDO and its use in the Eli system please consult the Eli documentation available in the Eli distribution from http://eli-project.sourceforge.net.

```
PEN UP          PEN UP
  DRAW 3          DRAW 1                   *
NEP             NEP                       ***
PEN DOWN        PEN DOWN                 *****
  DRAW 1          DRAW 5                *******
  NEWLINE         NEWLINE
NEP               DRAW 7
PEN UP            NEWLINE
  DRAW 2        NEP
NEP
PEN DOWN
  DRAW 3
  NEWLINE
NEP
```

Figure 2. Simple Plotter Language Sample Program and Output

used to perform integer addition and the C *printf* function is used to print the value of the expression.

2.2 *Simple Plotter Language*

In the desk calculator example, all attribute values of symbols on the left-hand side of productions are derived from the values of attributes on the right-hand side; that is, from immediate child nodes in the tree. This is a very simple pattern of attribution. In more complex examples it is often useful to be able to refer to attribute values from nodes which are further removed: either above the current symbol in the tree or in the sub-tree rooted at the current symbol.

The left-hand two columns of Figure 2 show a program written in a simple language that requires this sort of processing. The language allows control of a plotter pen. The pen may be set to the UP state or the DOWN state. In either state the pen can be moved to the right with the DRAW command. The DRAW command takes an integer argument which determines how many spaces to the right the pen will move. When the pen is DOWN, it will draw a character for each space that it moves right. When the pen is UP, the pen will also move to the right, but will not draw anything. There is also a NEWLINE command that will place the pen on the first space of the next line. The right-most column of the figure shows the output that is produced by this program.

The complete Simple Plotter Language attribute grammar is given in Figure 3. The grammar uses Eli's Pattern-based Text Generator (PTG) library to accumulate the output in the chain attribute *pentrace* (initialised on line 5 and output on line 6). Each draw command has a *pattern* attribute holding the output to be produced by that command (lines 24 and 28). The pattern for a draw command is added to the *pentrace* chain in line 32.

For our purposes, the most relevant part of this grammar is the use of the INCLUDING remote attribution construct in line 25. The language semantics

```
 1  ATTR pattern: PTGNode;
 2  CHAIN pentrace: PTGNode;
 3
 4  RULE: program ::= pen_blocks COMPUTE
 5     CHAINSTART HEAD.pentrace = PTGNULL;
 6     PTGOut (TAIL.pentrace);
 7  END;
 8  RULE: pen_blocks LISTOF pen_block END;
 9
10  RULE: pen_block ::= 'PEN' pen_modifier commands 'NEP' COMPUTE
11     pen_block.pattern = pen_modifier.pattern;
12  END;
13  RULE: pen_modifier ::= 'DOWN' COMPUTE
14     pen_modifier.pattern = PTGAsIs ("*");
15  END;
16  RULE: pen_modifier ::= 'UP' COMPUTE
17     pen_modifier.pattern = PTGAsIs (" ");
18  END;
19  RULE: commands LISTOF command END;
20  RULE: command ::= draw END;
21  RULE: command ::= pen_block END;
22
23  RULE: draw ::= 'DRAW' length COMPUTE
24     draw.pattern = repeatPTG (length,
25                               INCLUDING pen_block.pattern);
26  END;
27  RULE: draw ::= 'NEWLINE' COMPUTE
28     draw.pattern = PTGAsIs ("\n");
29  END;
30
31  SYMBOL draw COMPUTE
32     THIS.pentrace = PTGSeq (THIS.pentrace, THIS.pattern);
33  END;
```

Figure 3. Simple Plotter Language Attribute Grammar

are that the state of the pen for a draw command is to be obtained from the block that immediately encloses the draw command. Thus the computation obtains the pattern to be applied when drawing from the *draw* symbol's *pen_block* ancestor. On line 11 *pen_block* obtains the actual pattern from its *pen_modifier* descendant (lines 14 and 17).

2.3 The problem with INCLUDING

The Simple Plotter Language grammar is correct but it illustrates a typical situation where a mistake is easy to make. A likely initial attempt at specifying the INCLUDING is:

```
RULE: draw ::= 'DRAW' length COMPUTE
   draw.pattern = repeatPTG (length,
                             INCLUDING pen_modifier.pattern);
END;
```

In other words, we are attempting to directly access the *pen_modifier* pattern from the *draw* context. Unfortunately, this usage is statically illegal because *pen_modifier* will never be an ancestor of *draw*. When processed by the Eli system, this version produces the following error message.

```
"spl.lido", line 25,30: ERROR: in some contexts none
                        of the INCLUDING symbols is found
```

¿From this error message it can be difficult to diagnose the problem. In general it requires careful analysis of the abstract syntax to determine the relationship between *draw* and *pen_modifier*.

Instead of just notifying the user about the error, the system could provide a counterexample that illustrates the problem. In this case a counterexample will be a path in some tree, from the context of the INCLUDING (*draw*) to the root of the grammar, that does not include the required symbol (*pen_modifier*). The method we describe in the rest of this paper produces the following messages in addition to the one above.

```
"spl.lido", line 25,30: eg, can get to root via path:
                        draw -> command -> commands ->
                        pen_block -> pen_blocks -> program
"spl.lido", line 25,30: pen_modifier symbol(s) will not be
                        reachable by INCLUDING in some trees
```

These messages clearly identify the problematic symbol and a way in which the static requirements of INCLUDING are violated. With this information the cause of the problem is almost always immediately apparent.

The INCLUDING problem is actually more complicated than presented here. LIDO also allows computations to be associated with symbols instead of rules. Moreover, it is possible to have *class symbols* that do not appear in a production, contrasted with *tree symbols* that occur in productions and hence in the AST. Class symbols can be *inherited* onto tree symbols thereby allowing their attribution to be reused. This capability lies at the heart of LIGA's support for reusable attribution modules [11]. Since class symbols can have attribution and hence INCLUDING constructs, any system for diagnosing problems with INCLUDING must also take into account symbol inheritance.

2.4 Impact

It is easy to conceive of other forms of analysis that would be useful. For example, once the user has received the extended message above, they will probably want to find a common ancestor of *pen_modifier* and *draw*. Once an ancestor is determined, *pen_block* in this case, the pattern can be transported there and it will then be accessible from *draw* using an INCLUDING. So, support for finding common ancestors is another potentially useful analysis.

LIDO supports other remote attribution constructs. `CONSTITUENT(S)` is similar to `INCLUDING` except that it creates a dependency between a symbol and one or more other remote symbol below the current symbol in the tree. The `CHAIN` construct creates a chain of dependent values which are threaded throughout all the symbols in a depth-first left-to-right order through the tree. These constructs have similar static checks that could be analysed in a similar way to the INCLUDING problem illustrated in this paper.

The real benefit of automatic analysis of the kind described in this section is apparent when bigger grammars are considered. If the attribute grammar contains dozens of non-terminals, defined by 50–100 rules, with dozens of attributes then manually determining the relationships between non-terminals can mean wading through a large amount of irrelevant detail. This problem gets even worse when the grammar is comprised of independent modules possibly with references to library modules or when the grammar was developed by somebody else. In these cases the user may have to search through multiple grammar files some or all of which were not written by them. In contrast, an automatic approach integrates nicely with the normal reporting mechanisms of the system and requires no extra work on the part of the user.

3 Formal Methods

Formal methods have been used for decades to aid in the development of software, with the goal of ensuring that it meets its stated requirements. Formal methods analysis tools, which are used to automate to a greater or lesser extent the checking of the correctness of a system, can be divided into two general types: *proof assistants* and *model checkers*.

Proof assistants (also known as theorem provers), guide users through a series of logical deductions relating to their software specification. Systems comprising complex data types can be described quite elegantly, using the mathematically-based languages of proof assistants. In particular, infinite structures such as trees pose no particular impediment to these techniques.

The level of automation available with different proof assistants varies, but ultimately either an assertion is proved or a point is reached where no further deductions can be made. If the proof is concluded successfully then a great deal of information about the problem is established and one may have much confidence in the result. However, if a proof cannot be completed, then the technique provides very little additional information about the problem that may help the developer progress with the development of the software. The cause of the problem may be in the software under development, but it could also lie with the proof tactics themselves. In general, successful use of a proof assistant requires significant intervention from the developer.

Like proof assistants, model checkers in general can verify the correctness of a system however in contrast to proof assistants, model checkers do not provide much additional information about a correct software description. However they can usually be used without developer intervention and can identify *counterexamples* to illustrate the falsity of assertions. Counterexamples can provide significant assistance during the software development process because they point to specific circumstances in which desired properties of the software fail to hold. For our purposes, counterexamples are crucial because they form the basis of providing better feedback to an attribute grammar developer.

The most widely-known class of model checking tools operate on finite-state models. The symbolic model verifier SMV is a well-known tool in this class [15]. SMV takes a description of a finite-state machine and a specification expressed as a statement in the Computational Tree Logic (CTL) which is a form of temporal logic. The verification goal is achieved by a search of the state space defined by the machine description. The search yields a counterexample whenever it finds a state in which the CTL specification is not satisfied. The counterexample consists of the sequence of states leading up to that point. From the states in the counterexample the developer can determine circumstances in which the machine would violate its specification.

Finite-state models are a powerful specification method and tools like SMV are able to search very large state spaces. This method is most suitable for verification of complex concurrent systems such as communication protocols or hardware architectures where the notion of state is prevalent. We have applied this approach to modelling attribute grammars but found the semantic gap between the attribute grammar concepts we want to model and the notion of finite-state machines to be too great. In attribute grammar analysis it is not natural to think in terms of states and state variables and SMV doesn't support mechanisms for abstracting away from the underlying state-based model. We were able to produce counterexamples using SMV but the machines and specifications were unintuitive and unwieldy.

Instead, we have based our work on the Alloy tool [2] which provides some of the advantages of both of the two types of formal methods described above [6]. The Alloy specification language uses a first-order relational logic, is declarative and is based on the formal specification language Z [17]. Unlike Z, Alloy specifications can be analysed automatically in much the same way as a finite-state machine specification can be analysed by a model checker. Just like model checkers based on finite-state models, Alloy provides counterexamples

[2] Alloy is available at http://alloy.mit.edu.

to illustrate non-satisfiable assertions.

Alloy's method is governed by the *scope* of the model which limits the maximum size of the atom sets underlying the modelled relations. Thus the approach is fundamentally incomplete in the sense that an unduly small scope may prevent a counterexample from being found. We do not believe this is a serious problem for our approach since we are not relying on the model checkers ability to detect errors but rather to illustrate errors that are already known to exist. We discuss the issue of scope selection in Section 5.

Unlike finite-state model checkers, Alloy's specification language is expressive enough to make it easy to reason about systems with states made of complex data structures. Alloy's expressive relational language is therefore well suited to describing the tree or graph-like structures inherent in attribute grammars. Further, since a grammar is not a dynamic system, but rather a static structure, Alloy's relational logic provides a more intuitive language for describing assertions regarding a grammar than a temporal logic such as CTL.

3.1 Alloy by example

This section briefly introduces the Alloy specification language by way of a simple model specification of families. This example is loosely based on sample code supplied with the Alloy distribution. We will introduce Alloy notations informally since in most cases they correspond closely with well-known concepts of logic and mathematical relations. The Alloy manual contains a full description of the specification language syntax and semantics [6].

The family model specification begins by specifying a class of atoms representing people.

```
sig Person {
  spouse: option Person,
  parents: set Person
}
```

The above statement declares a class of people by declaring the "signature" of relations in which people participate. A person p is related to their spouse (who is also a person) and is related to a set of people who are the parents of p. The spouse relation is optional (i.e., possibly empty) since not all people have a spouse.

We partition the class of people into sub-classes of men and women using

```
part sig Man, Woman extends Person {}
```

Facts are statements regarding the membership of sets, and relations between atoms of sets that are true of all arrangements of atoms. For example, a biological fact that no person can be their own ancestor could be written as:

```
fact Biology {
  no p:Person | p in p.^parents
}
```

In more mathematical words this fact states that there is no member p of the class of people such that p is in the set formed by taking the transitive closure on the parents relationship starting at p.

Facts regarding social norms of the spouse relationship could include:

- The spouse relationship is symmetric (the tilde symbol ˜ represents the transpose of the given relationship).
  `all p:Person | p.spouse = p.~spouse`
- Nobody is his or her own spouse.
  `no p:Person | p.spouse = p`
- A man's spouse is a woman and vice versa.
  `Man.spouse in Woman && Woman.spouse in Man`

We might like to see whether our model prohibits intermarriage. We define this to mean that no person's spouse is either their sibling or their parent or other direct ancestor. None of the facts above explicitly restricts the model in this way, but we can check whether they implicitly define this restriction by specifying the requirement as an assertion and searching for a counterexample to the assertion. Here is one possible assertion:

```
assert NoIntermarriage {
  no p:Person | some p.spouse &&
                ((some p.spouse.parents & p.parents) ||
                 (p.spouse in p.^parents))
}
```

In other words, the assertion says that there is no person p who has a spouse where the spouse has a parent in common with p or where the spouse is an ancestor of p.

When presented with the model specification and this assertion, Alloy generates models that satisfy the specification and checks that the assertion is true in each of these models. It is important to appreciate the difference between facts and assertions. Facts specify constraints that models must obey to conform to the specification. Alloy will not consider models that do not satisfy the facts. Assertions specify properties that we expect to hold but may not actually hold for some models.

Since typical specifications describe potentially infinite structures, it is necessary to restrict the scope of the classes used to generate models. In our example, we must restrict the size of the Person class; in other words, we must specify the maximum number of people in a model that is to be checked. For this specification the assertion fails on some models with a Person class having scope of just three. The counterexample generated by Alloy is the obvious one: Person 0 is the parent of both Person 1 and Person 2, and Person 1 and Person 2 are spouses of each other. Armed with the knowledge that the existing constraints are not sufficient, the developer can add additional constraints to the specification to ensure that this situation is not allowed.

Clearly the question of the scope of a model is important for practical verification with Alloy. The scope must be restricted so that the method is tractable. Alloy is implemented by translating the model specification into a Boolean expression that is passed to an off-the-shelf SAT solver [5]. SAT solving technology is currently quite powerful but still has limits on the size of the expressions that can be practically analysed. By restricting the scope of the underlying sets in an Alloy model we can limit the size of the expressions that must be solved. Of course, the limit means that verification is not complete. Even if Alloy fails to find a counterexample, there may be larger models that violate assertions. In practice, experience shows that tractable model sizes suffice to find problems with realistic model specifications [7,12].

4 Modelling the INCLUDING Problem

Now we turn to the task of modelling the INCLUDING problem outlined in Section 2 using the Alloy specification language. First, we consider how to capture the properties that all attribute grammars have which are needed to solve the INCLUDING problem. Then we consider extending the specification with details of a particular attribute grammar using the Simple Plotter Language grammar from Section 2 as an example. Once the specification is complete, we show how to express assertions about the static requirements of INCLUDING constructs. We then describe how to extend the model to deal with LIDO's symbol attribution inheritance construct.

The techniques described in this section are independent of the particular attribute grammar notation. They have been embodied in an Eli-specific tool that translates LIDO attribute grammars into Alloy specifications. Thus the analysis process is completely automatic. We evaluate the approach in the Eli setting in the next section.

4.1 Properties of attribute grammars in general

For our purposes, the relevant properties of an attribute grammar relate to paths in the ASTs conforming to the underlying context-free grammar. To this

```
sig GGraph[s] {
   root : s,
   parent : s -> s,
   child : s -> s
} {
   // child is the transpose of parent
   child = ~parent
   // Don't allow cycles
   no n : s | n in n.^child
   // No disconnected components
   s in root.*child
}
```

Figure 4. Alloy specification of the relationship between non-terminal symbols in ASTs.

end, we model the relationships between non-terminal symbols in the AST. There is a distinguished *root* non-terminal and a *parent* relationship between non-terminals. A non-terminal X is a parent of a non-terminal Y if X occurs on the left-hand side of a production and Y occurs on the right-hand side of the same production. All non-terminals except the root will have at least one parent. Terminal symbols are ignored because they cannot have associated attribution in LIDO and hence don't figure in the INCLUDING problem.

Figure 4 shows an Alloy specification GGraph of the parent-child relationship. The specification is generic in *s* which is the class of atoms representing non-terminal symbols. The first part of the specification declares the relevant relations: a unary relation representing the root, then the binary parent relation. Some properties are easier to state with a child relation than with a parent relation so we declare that too.

After the signature are facts (or constraints) that must always hold for every instance of GGraph. First, the child relation is just the transpose of the parent relation so that only one of them needs to be defined explicitly. Second, we don't allow cycles in the child relation; this enables us to use a finite model to describe potentially infinite ASTs arising from a recursive grammar. [3] Finally, we make sure that all components of the model are connected by requiring that all symbols are accessible from the root.

GGraph defines all of the relationships between non-terminals. For the INCLUDING problem we need to be able to talk about particular paths in this graph. The following specification builds on the GGraph signature, adds no relations, but adds a fact that restricts the path models to those where each symbol has at most one child. For any single GGraph instance there will be potentially many GPath instances representing all of the different paths from symbols to the root.

[3] Note that this decision is justified by the nature of the INCLUDING problem. We don't need to consider what happens to recursive occurrences of a non-terminal X because they will be legal if and only if the non-recursive occurrences of X are legal.

```
sig GPath[s] extends GGraph[s] {
} {
   all n : s | sole n.child
}
```

Finally, we declare a class of symbols. This class serves as a placeholder and will be extended by the actual symbols from a grammar in the next section.

```
sig Symbol {}
```

4.2 Properties of specific attribute grammars

Models of specific attribute grammars conform to the specifications in the previous section but also have to provide details to fill in the gaps in those specifications: the identity of the symbols (including which one is the root) and the nature of the parent or child relation.

The non-terminal symbols of the grammar are declared by extending the general Symbol class.[4] We require that the class for a particular symbol be disjoint from that of the other symbols (indicated by the `disj` keyword); otherwise, Alloy would permit models which equate different symbols.

```
disj sig program, pen_blocks, pen_block, pen_modifier,
   commands, command, draw extends Symbol {}
```

Since we do not need to consider recursive constructs, we can limit each of these symbol classes to contain at most one atom. This constraint significantly reduces the size of the search space.

```
fact atMostOne {
   sole program
   sole pen_blocks
   ...
}
```

The identity of the root symbol and the nature of the parent or child relation are specified by a fact about all instances g of the GGraph specification instantiated with the symbol class s being `Symbol`. We choose to define the child relation explicitly because it is more intuitive than the parent relation; the parent is implicitly defined by the transpose constraint given in the previous section.

```
fact {
```

[4] In the following we assume that LIGA identifiers can be safely used directly in an Alloy specification. In practice, some syntactic changes are required to ensure that these uses are legal.

```
   all g : GGraph[Symbol] |
        g.root in program
     &&
        g.child in (program -> pen_blocks +
                    pen_blocks -> pen_block +
                    pen_block -> (pen_modifier + commands) +
                    commands -> command +
                    command -> draw +
                    command -> pen_block)
}
```

The first part of the fact body says that it concerns all GGraph[Symbol] graphs *g*. Then we specify that *program* is the root of *g*. Then the child relation is defined by enumerating its constituent pairs as given by the context-free grammar; for example, the pair *(program,pen_blocks)* is in the child relation because of the production `program ::= pen_blocks`.

4.3 Assertions about remote attribution

Finally we need to express assertions that describe the static requirements of the INCLUDING constructs in the attribution.[5] In the erroneous Simple Plotter Language example there is only one INCLUDING:

```
RULE: draw ::= 'DRAW' length COMPUTE
   draw.pattern = repeatPTG (length,
                             INCLUDING pen_modifier.pattern);
END;
```

The requirement implied by this INCLUDING is that from a *draw* context it will always be possible to go up the AST and find a *pen_modifier* node. We can express an assertion that this requirement holds in Alloy as follows:

```
assert drawIncludespen_modifier {
   all p : GPath[Symbol] |
      all x : draw |
         some y : pen_modifier |
            y in x.^(p.parent)
}
```

[5] The general form of INCLUDING allows more than one attribute to be specified, in which case the construct is legal if any one of the specified symbols is present as an ancestor in any AST. Our method easily extends to dealing with multiple symbols in INCLUDINGs but we omit the detail for space reasons.

In other words, for all paths in the graph, for each instance of a *draw* symbol there must be a *pen_modifier* that is in the transitive closure of the path's parent relation starting at the *draw*.

When presented with the model specification from the previous two sections and this assertion, we can ask Alloy to search for models that conform to the specification but violate the assertion. We use the following Alloy check statement to constrain the scope of the models to include no more than five symbols since there can be no more than five symbols in a path from any symbol to the root of this grammar. Also, we only use one GGraph instance because it can be shared between all paths.

```
check drawIncludespen_modifier for 5 but 1 GGraph[Symbol]
```

One model that violates the assertion contains the following child relation:

```
p.child = (program -> pen_blocks +
           pen_blocks -> pen_block +
           pen_block -> commands +
           commands -> command +
           command -> draw)
```

which conforms to the following erroneous path:

```
draw -> command -> commands -> pen_block ->
   pen_blocks -> program
```

4.4 Modelling symbol inheritance

Up to this point our model specification only deals with tree symbols. We must add modelling of inheritance relationships in order to be able to deal properly with INCLUDING constructs that occur in class symbol attribution or that refer to a class symbol as the ancestor symbol that must be present. Figure 5 shows a simple example of the former case. The tree symbols are A, B, C and D (lines 2–7). The sole class symbol is Q (line 9) which is inherited onto D (line 8). The computation of *Q.value* makes use of an INCLUDING to obtain the value from a B ancestor (line 10). If the D associated with the Q is derived from B then the INCLUDING is legal, but D can also be derived from C and then from A in which case no B is present.

To model symbol inheritance we extend the basic graph model with the following two relations which are transposes of each other:

```
sig GGraph[s] {
   ...
   inherits : s -> s,
ATTR value: int;
RULE: A ::= B COMPUTE
  B.value = 1;
END;
RULE: A ::= C END;
RULE: B ::= D END;
RULE: C ::= D END;
TREE SYMBOL D INHERITS Q END;
CLASS SYMBOL Q COMPUTE
   INH.value = ADD (INCLUDING B.value, 1);
END;
```

Figure 5. A simple attribute grammar using symbol inheritance.

```
    inheritedby : s -> s
} {
    ...
    inherits = ~inheritedby
    s in root.*child + (root.*child).*inherits
}
```

We have also extended the reachability fact to include the possibility of reaching a symbol via the inheritance relationship. This makes sure that class symbols are connected to the rest of the model. The definitions of GPath and Symbol remain as before.

To distinguish between tree and class symbols we add an extra level of discrimination to the symbol class and ensure that there are no other symbols.

```
disj sig TreeSymbol, ClassSymbol extends Symbol {}
fact { Symbol = TreeSymbol + ClassSymbol }
```

The actual symbols in the grammar then extend either *TreeSymbol* or *ClassSymbol* as appropriate.

```
disj sig A, B, C, D extends TreeSymbol {}
disj sig Q extends ClassSymbol {}
```

The root of the grammar and the child relationship are specified as before but are augmented by a specification of the inheritance relationship between *D* and *Q*.

```
fact {
   all g : GGraph[Symbol] |
        g.root in A
     &&
        g.child in (A -> (B + C) +
                    B -> D +
                    C -> D)
     &&
        g.inherits in (D -> Q)
}
```

A slight complication arises due to the fact that Alloy may choose to omit any of the symbols from the model of a particular path. Only the symbols that are actually on the path need to be in the model. However, if one of those symbols inherits one or more class symbols, then we must make sure that these class symbols are also in the model. Otherwise, we are not able to properly model the situation where an INCLUDING refers to a class symbol that is inherited onto an ancestor symbol. For the example, we need the following extra constraints which specify: 1) that if there is a D in the model then there must be a Q too, and 2) that if there is a D in the model then every path should have the inheritance relationship between D and Q.

```
fact {
   some D => some Q
   some D => all p : GPath[Symbol] | (D -> Q) in p.inherits
}
```

To express the requirement of the INCLUDING in line 10 of Figure 5 we use the following assertion.

```
assert QIncludesB {
   all p : GPath[Symbol] |
      all x : Q |
         some y : B |
            y in (x.*(p.inheritedby).^(p.parent).*(p.inherits))
}
```

which has the same form as before except it traverses the inheritance relation as well as the parent one. The requirement on the last line ensures that we follow the inheritance relation down from Q to any tree symbols that inherit Q, then go up the AST via the parent relation from those tree symbols, and then include any class symbols inherited to the ancestor tree symbols that we find. With this assertion we can check the INCLUDING with the command

```
check QIncludesB for 3 but 1 GGraph[Symbol]
```

and Alloy will find the following counterexample:

```
Q -> D -> C -> A
```

5 Evaluation

Our early attempts to develop models required us to model the entire grammar at once. This turns out to be a poor strategy because it results in models that are so large as to be intractable to generate and check. On non-trivial grammars we always gave up waiting for Alloy to produce a counterexample.

The version we present in this paper remedies this deficiency by only modelling paths, which in general contain many fewer symbols than the whole grammar. Put another way, the scope required for the symbol class need only be big enough to deal with the longest possible non-recursive path. For realistic grammars, we can expect this to be in the order of 10 to 20 symbols. The relations representing symbols that do not occur on the path can be empty so they don't contribute to the complexity of the solution process.

Our main evaluation has been performed on a grammar that performs semantic analysis for a subset of Pascal [19]. The grammar has 52 productions and uses 68 tree symbols and 25 class symbols; the class symbols come mostly from library modules. There are 36 INCLUDING constructs in the grammar. We ran our experiments on an Apple Powerbook G4 with 1 GHz processor and 1GB RAM running Mac OS X 10.2.6.

We developed an Eli package that can be invoked by the user when LIGA reports an INCLUDING problem. We implemented a translator that takes LIDO attribute grammars and translates them into Alloy specifications as described in Section 4.[6] (For the Pascal subset grammar the Alloy specification is generated in a few seconds.) The Eli package translates the user's LIDO specifications, runs Alloy, translates the counterexamples (if any) in the Alloy output back into LIDO terminology, and presents the counterexample in an error message as shown in Section 2.3. If Alloy fails to produce a counterexample then the user will just see the original LIDO messages.

We timed the analysis of the correct Pascal subset grammar in order to determine worst case times.[7] Our current measurements show that it is possible to analyse all 36 of the INCLUDINGs in about 20 minutes of user time with a scope of 10, or 34 minutes with a scope of 15. (The two different scopes produce the same results.) These times benefit significantly from the decision described in Section 4.2 to limit each symbol class to hold at most one atom. Without this restriction a full analysis with scope 10 takes over 20 hours.

We expect our approach to be most useful once a problem with a specific INCLUDING has been diagnosed by some other method (e.g., via a message from LIGA). Thus, a more realistic use case is to analyse just one INCLUDING at a time. We examined the individual INCLUDINGs in our test case and

[6] Our current translator does not implement one aspect of LIGA semantics: If symbol X inherits from symbol Y and Y has a definition of an attribute, then X can override that definition. If Y's version involves an INCLUDING our current tool will incorporate checking for that INCLUDING even though it has been overridden and will not be used.

[7] We have not observed any memory consumption problems so we don't report these measurements here.

determined the one that takes the biggest proportion of the analysis time. It requires about 24 seconds with a scope of 10 or one minute with a scope of 15. These times do not quite yield interactive performance but are quite bearable for those rare situations where a diagnosis is needed.

In ongoing work, we expect to see performance improvements due to:

(i) Only modelling erroneous INCLUDING constructs. Our current model specifications incorporate the whole grammar and assertions for every INCLUDING. A better strategy is to target the models towards solving only those INCLUDINGs that are triggering LIGA errors. We expect to see diagnosis times for a problem with a single INCLUDING in the order of ten seconds for grammars of reasonable size.

(ii) More accurate setting of the scope of the model. At present we use a fixed scope for all grammars. A better approach would be to calculate the maximum scope needed for a specific grammar so that we can guarantee that no counterexample is missed but limit the redundant work that must be done. In the INCLUDING case, this would be a measure of the longest possible non-recursive path.

(iii) Improvements in the SAT solving technology underlying the Alloy tool. Since we are using Alloy as a black box, we can automatically benefit as better off-the-shelf SAT solvers come along.

6 Conclusion and Future Work

We have described a practical method for analysing attribute grammar specifications using an off-the-shelf formal methods tool. We have demonstrated the effectiveness of the method by showing how a typical problem involving remote attribution can be diagnosed automatically. Our analysis tool is portable to other attribute grammar notations. Performance is almost acceptable for interactive use and we have outlined modifications that we expect to improve it even further.

Our work demonstrates that using an off-the-shelf tool approach is a viable alternative to modifying existing tools. The tradeoff was whether the effort to develop the LIDO to Alloy translator was likely to be more than the effort required to modify LIGA. We believe this was clearly the case for us, being non-LIGA experts, but we haven't conducted a formal comparison. Also, a large part of our translator could be used to build a translator for other attribute grammar systems.

We plan to extend our tool to diagnose problems with other attribute grammar constructs starting with other forms of remote attribution. We will also investigate whether a formal methods approach can be used to diagnose semantic errors in attribute grammars particularly problems with the use of standard library modules.

Acknowledgments

Bill Waite consulted on the general problem of grammar analysis and the issue of scope in typical grammars. Ilya Shlyakhter and Alan Fekete provided insight into the appropriate use of the Alloy model specification language. Dom Verity consulted on various aspects of logic.

References

[1] Henk Ablas. Introduction to attribute grammars. In *International Summer School on Attribute Grammars, Applications and Systems*, Lecture Notes in Computer Science, vol. 545, Berlin, Germany, 1991. Springer-Verlag.

[2] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography.* Lecture Notes in Computer Science, vol. 323. Springer-Verlag, Berlin, Germany, 1988.

[3] Shirley Goldrei. Formal verification of attribute grammar specifications. Honours thesis, Macquarie University, 2003.

[4] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[5] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 130–139. ACM Press, 2000.

[6] Daniel Jackson. Micromodels of software: Lightweight modelling and analysis with Alloy. Download from http://alloy.mit.edu/reference-manual.pdf, 2002.

[7] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–25. ACM Press, 2000.

[8] Uwe Kastens. *GAG: A Practical Compiler Generator.* Springer Verlag, 1983.

[9] Uwe Kastens. LIGA: A language independent generator for attribute grammars. Technical Report Bericht Nr. 63, University of Paderborn, Germany, 1989.

[10] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In *International Summer School on Attribute Grammars, Applications and Systems*, LNCS, vol. 545, Berlin, Germany, 1991. Springer-Verlag.

[11] Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[12] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 13. IEEE Computer Society, 2000.

[13] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[14] Donald E. Knuth. Semantics of context-free languages: Corrections. *Mathematical Systems Theory*, 5(1):95–96, 1971.

[15] K.L. McMillan. The SMV system for SMV version 2.5.4. Download from http://www-2.cs.cmu.edu/~modelcheck/smv/smvmanual.ps, 2000.

[16] Jukka Paaki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, June 1995.

[17] J.M. Spivey. *The Z Notation; A reference Manual.* Prentice Hall, 1987.

[18] Lincoln David Stone. Slicing compiler specifications. Honours thesis, Department of Computer Science, James Cook University, 1997.

[19] W. M. Waite. A complete specification of a simple compiler. Download from http://www.cs.colorado.edu/~eliuser/pascal_html/pascal-.ps, 1997.