MECHANIZING LOGIC IN AN ASPECT ORIENTED ATTRIBUTE GRAMMAR SYSTEM

v c

A. Azurat, I. S. W. B. Prasetya, and S. D. Swierstra

Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
e-mail: {ade,wishnu,doaitse}@cs.uu.nl

ABSTRACT

This paper reports a preliminary work on using an aspect oriented attribute grammar system called UU_AG to develop computer aided verification tools. UU_AG provides an abstract and modular way to develop such a tool and later on incrementally upgrade them. This paper shows an example of a toy programming logic implemented in UU_AG. We will show the implementation of the verification condition generator (VCG). We extend the implementation with a new feature such as run-time-trace generator to validate the computation of the implemented inference engine.

Keywords: Verification tools, Attribute grammar, Verification condition generator

Manuscript received [10 Maret 2003]. Revised [12 Agustus 2003].

1. INTRODUCTION

Program verification uses mathematical models to characterize the entire set of the possible behaviour of a program and with a suitable logic we can prove properties about it without having to actually execute the program. Although it is superior to testing, because of its complete coverage, in practice program verification is often considered as expensive (requiring people who are highly skilled in mathematics). In certain areas however (safety or mission critical systems), we have seen that people are beginning to consider verification as a serious option.

Crucial in making the approach more affordable is the use of tools. Many components of verifications tools carry out non-trivial transformations on programs or formulas. When developing such a component we often start with a version that only offers a set of basic capabilities, and in some later stages incrementally extend it with more capabilities, such as error messaging, reporting, and run-time validation capabilities. However, these kind of extensions often cannot be composed hierarchically. Instead, they have to be tightly woven together. From experience we know that this is very error prone! So we want to have a better way to implement a verification tool –a tool for producing trustworthy programs should in itself be very trustworthy.

Attribute grammar [8] is a programming formalism that allows us to declaratively describe (recursive) tree walking programs (the programmer does not need to specify the computation flow). Systems implementing attribute grammars, such as the UU_AG system[5], are able to read attribute grammar specifications and generate their implementations. Such a system is suitable for specifying and generating various components for verification tools, e.g. as shown in [7]. Another very attractive feature of UU_AG is that it supports an *Aspect Oriented* (AO) style of programming[4]. Unlike conventional programming, AO programming allows aspects (capabilities) of the same program to be described separately (modularly).

To demonstrate the virtue of UU_AG we will first show how a toy programming logic can be implemented using the system and then we will show how a new aspect can be added to it. In this example, the new aspect generates, for each individual run of the logic's inference engine, a Hoare triple styled proof to certify the run. In general, the same technique can be used to modularly add a run-time validation capability to a program.

Section 2 gives a brief introduction of UU_AG –more information about the system can be found in [5]. Section 3 shortly describes the logic that we use as the running example (called VSPL). Section 4 shows how VSPL is described in UU_AG. Section 5 shows how to add a new aspect to VSPL's inference engine. Section 6 concludes. We will assume the reader's familiarity with functional programming and data type representation of grammars.

Code 2.1 :

```
DATA Bintree
  | Leaf          Int
  | Node          left:Bintree right:Bintree

ATTR Bintree [ | | minLeaf : Int]
SEM Bintree
  | Leaf lhs.minLeaf = @int
  | Node lhs.minLeaf = @left.minLeaf 'min' @right.minLeaf
```

2. UU_AG THE SYSTEM

Code 2.1 shows a specification of a Haskell-like or ML-like data type called Bintree for representing binary trees and a function to compute the smallest leaf in a given binary tree.

The DATA section specifies the data type. Note that unlike in Haskell or ML, in we can specify the name of each field of a given data constructor. For example, the constructor Node has two fields. One is called left and it contains the left sub-tree under the node. The other is called right, containing the right sub-tree.

UU_AG is an attribute grammar system. It means that it is specifically intended for the programming of (recursive) tree walking programs, such as a program that computes the smallest leaf in a given tree. In the attribute grammar formalism, such a program is usually called a *semantic function* as it can be thought to 'read' a tree and interpret it to something else. Semantic functions are specified in a SEM section. Note that the attribute grammar style of specification is more abstract than, for example, an ordinary Haskell program: we only need to describe what the function does on each node (and leaf), without having to explicitly code the recursion itself –see the SEM section in Example 2.1.

During the recursion we usually need to pass information from parent to child and from child to parent. In the attribute grammar formalism, *attributes* are used for this purpose. We can think that each node in a tree is extended with these attributes. Attributes which are used to hold data passed from parent to child are called *inherited* attributes. Those that are used to pass information in the other direction is called *synthesized* attributes.

Attributes are specified in the ATTR section. For the computing the smallest leaf, we only need one synthesized attribute called minLeaf containing an integer which is the minimum value of the leaves under the node to which the attribute is attached.

Given the specification as in Code 2.1 the UU_AG system will generate a Haskell implementation thereof.

Quite often a semantic function has to walk a tree several times and in a certain order to produce its final result. Each walk can be thought to compute a set of attributes which are needed as the input for the next walk. For example, replacing the leaves in a given tree T with the value of the smallest leaf in T (so-called *repmin* problem [2]) will require the tree to be walked twice. For a complicated semantic function, e.g. a type inference function, specifying the right order with which the tree should be walked is not trivial. relies on the lazy evaluation of its target language (Haskell) to automatically discover the right order in which the attributes are computed. So, this complication is abstracted away: the user does not need to specify nor to be aware of the order of with which the attributes are to be computed.

Another benefit of UU_AG is its aspect oriented style. We will explain this as we go later.

3. RUNNING EXAMPLE: VSPL

Consider a very simple programming logic – abbreviated VSPL – whose languages are described in Figure 1. A program is either a skip, an assignment, a nested conditional, or a catenation of two statements. VSPL has only two types of values: Boolean and Integer. Programs are specified using Hoare triples, for example:

Example 3.1 : Swapping value

$\{x = A \wedge y = B\}$

```
 x := x + y;
 y := x - y;
 x := x - y
```

$\{x = B \wedge y = A\}$

The specification can be verified by first computing the weakest pre-condition of the program, and then show that the pre-condition implies this weakest pre-condition:

Theorem 3.2 : Spec-reduction Rule

$$\frac{[P \Rightarrow wp\, P\, Q]}{\{P\}\, S\, \{Q\}}$$

The weakest pre-condition wp can be computed in the standard way:

Definition 3.3 : WP

```
wp skip Q              =   Q

wp (v := E) Q          =   Q[E/v]

wp (if g then          =   (g ⇒ wp S Q) ∧
S else T) Q                (¬g ⇒ wp T Q)
wp (S ; T) Q           =   wp T (wp S Q)
```

4. IMPLEMENTATION

To implement VSPL in UU_AG we first need to define a set of Haskell-like data types to represent the grammar of VSPL. Values of these data types have a tree-like structure and represent VSPL programs and specifications, complete with their syntactic structures. This approach for representing grammars and sentences is quite standard.

The grammar of VSPL has three (major) syntactic categories: Stmt, Expr, and Spec representing statements, expressions, and specifications. Code 4.1 shows (a fraction of) the data types representing these categories.

Code 4.1: VSPL datatype

```
DATA Stmt
| Skip
| Assign      x: Variable  e: Expr
| Catenation  s: Stmt      t: Stmt
| IfThenElse  g: Expr
                 s: Stmt
                 t: Stmt

DATA Expr
| Negate      p: Expr
| Equal       x: Expr   y: Expr
| Imp         p: Expr   q: Expr
  ...
```

Once the data types are specified, we can define semantic functions to process values of those data types. For example, the code below shows how the reduction of a VSPL specification as specified by the inference rule in Theorem 3.2 can be coded in the UU_AG:

Code 4.2:

```
ATTR Spec [ | | vc : Expr ]
SEM Spec
 | Hoa lhs.vc = Imp @p.itself @s.wp
```

The ATTR section specifies one synthesized attribute called vc. We intend it to hold the result of the reduction. The SEM section above constructs the representation of the expression above the horizontal line (often called *verification condition*) in Theorem 3.2 and put it in the attribute vc. The itself keyword represents the entire subtree that hangs under a particular field of a given node (data constructor). Given the above code, UU_AG will produce the corresponding Haskell implementation (executable).

Once a VSPL specification is reduced by the above function, we still need to prove the resulting formula. We cannot use UU_AG for this purpose. There are however enough theorem provers, such as HOL [3], which are well suited for finishing the task.

Note that the function specified above requires the information in the attribute called wp. We have not specified this attribute yet, but it is intended to hold the weakest pre-condition of the given statement relative to the given post-condition. Definition 3.3 abstractly specifies how this can be computed. The code below shows its UU_AG implementation:

Code 4.3 : wp computation

```
ATTR Stmt [ q: Expr | | wp: Expr ]
SEM Stmt
 | Skip       lhs.wp  = @lhs.q
 | Assign     lhs.wp  = subst
                                          @x.itself
                                          @e.itself
                                          @lhs.q
 | Catenation lhs.wp  = @s.wp
              s.q     = @t.wp
              t.q     = @lhs.q
     ...
```

The function specified above will recursively walk a tree of type Stmt, which is the representation of an actual VSPL statement. It also receives the post-condition, which is passed in the inherited attribute q. The result is stored in the synthesized attribute wp.

Remember that a synthesized attribute A to a node N is used to pass information by the processing in N to the processing of N 's parent. In the current implementation of UU_AG, A is accessible from N 's parent, but not from A itself. The way to get around this is to keep a local copy of A . This has to be added by hand, but in the future version of UU_AG we hope that this will be automated. For the extension which we will discuss in the next section it is useful for each node in an Stmt tree to have access to its wp attribute. The code below shows a fraction of the new UU_AG code for the wp computation. Notice that there are now two wp attributes: a synthesized wp, and a local wp which is simply a copy of the other.

Code 4.4 : wp computation

```
SEM Stmt
 | Skip    loc.wp  = @lhs.q
           lhs.wp  = @wp

 | Assign  loc.wp  = subst
                                   @x.itself
                                   @e.itself
                                   @lhs.q
           lhs.wp  = @wp
 ...
```

5. MULTIPLE ASPECTS

The function to compute wp from the previous section, which is used to reduce a given VSPL specification, is very simple, so it is reasonable to say that it can be implemented quite reliably. In a more general we may have a quite complicated reduction algorithm. In an attempt to increase its reliability we may want to add a capability to produce a trace of validation code for each run of the program P implementing the algorithm. A validator can be constructed to read such a code and is used to validate a given run of P . Typically, such an extension has to be built by weaving it into P 's recursions, which is of course very error prone. The UU_AG system allows us to program each capability (aspect) separately.

Let us assume that in this case the validation code we try to generate is simply a Hoare triple styled proof confirming that the resulting wp will at least be strong enough to guarantee the post-condition of a given VSPL program. To implement this, we first define some datatypes to represent VSPL proofs –see Code 5.1. So, a tree of type Proof represents a proof. Given such a proof □ , the field goal specifies the VSPL specification which is proven by □ . The field ruleName specifies the name of the VSPL inference rule that will prove the goal. Application of this rule may generate premises which in turn need to be proven. The premises are specified in the premises field. Given such a proof, a suitable pretty printer can be written to render it to, for example, something like Figure 2. We will also need a validator to validate such a proof, which we will not show since this is not the main issue here.

Code 5.1 : Proof datatype

```
DATA Proof
  | Rule goal     : Spec
         premises : ProofList
         ruleName : String
```

Code 5.2 shows a (Haskell) implementation of Theorem 3.2 (for (automatically) reducing a given specification) that one would likely come up with when implementing it directly (without a generator such as).

Code 5.2 :

```
vcg_top p s t = vc where (vc,_) = vcg p
s t

vcg :: Expr -> Stmt -> Expr ->
(Expr,Expr)

vcg p (Assign x e) q = (vc,wp)
    where
           vc = Imp p wp
           wp = subst x e q

vcg p (Catenation s t) q = (vc,wp)
    where
    dummy = TT
    (_,wp_t) = vcg dummy t q
    (vc,wp) = vcg p s wp_t
...
```

The function vcg_top does the reduction. It calls vcg, which computes the reduction, and along with it the wp whose value is needed in computing the reduction.

Code 5.3 shows the same functions vcg_top and vcg, but now they are extended with the ability to co-produce a validation trace. Notice that to construct such a trace we have to collect some run-time information of vcg. So that is why its construction has to be weaved into the recursion of vcg (because it requires run-time information. As can be seen, the code becomes cluttered. To make things worse, we are not only required to weave a new piece of code into the existing one, but we also have to change the old code.

$$\dfrac{x = y = B \wedge y = A \Rightarrow x - y = B \wedge y = A}{\{x - y = B \wedge y = A\}\, x := x = y\, \{x = B \wedge y = A\}}\ \textit{Substituti on}$$

M

θ

$$\dfrac{\dfrac{x - (x - y) = B \wedge x - y = A \Rightarrow x - (x - y) = B \wedge x - y = A}{\{x - (x - y) = B \wedge x = y = A\}\ y := x - y\ \{x - y = B \wedge y = A\}}\ \textit{substitution} \qquad \theta}{\{x - (x - y) = B \wedge x - y = A\}\, y := x - y\ ;\ x := x - y\ \{x = B \wedge y = A\}}\ \textit{catenation}$$

M

ψ

$$\dfrac{\dfrac{x = A \wedge y = B \Rightarrow x + y - (x + y - y) = B \wedge x + y - y = A}{\{x = A \wedge y = B\}\ x := x + y\ \{x - (x - y) = B \wedge x - y = A\}}\ \textit{Substitution} \qquad \psi}{\{x = A \wedge y = B\}\ x := x + y;\ y := x - y;\ x := x - y\ \{x = B \wedge y = A\}}\ \textit{catenation}$$

Figure 2: The Hoare Logic rule traces of Example 3.1

Code 5.3 :

```
vcg_top p s t = (vc,vtrace) where (vc, _, vtrace) = vcg p s t

vcg :: Expr -> Stmt -> Expr -> (Expr,Expr,Proof)

vcg p (stmt@(Assign x e)) q = (vc,wp,vtrace)
  where
  vc = Imp p wp
  wp = subst x e q
  vtrace = Rule (HOA p stmt q) premises ruleName
  ruleName = "Assign rule"
  premises = [Rule (Valid (Imp p wp)) [] "Pred. logic"]

vcg p (stmt@(Catenation s t)) q = (vc,wp,vtrace)
  where
  dummy = TT
  (_, wp_t, _, _) = vcg dummy t q
  (_, _, rulet,vtrace_t) = vcg wp_t t q
  (vc,wp,rule_s, vtrace_s) = vcg p s wp_t
  vtrace = Rule (Hoa p stmt q) premises ruleName
  ruleName = "Catenation rule"
  premises = [vtrace_s, vtrace_t]
...
```

Notice how lines 13-14 in the old code have to be changed to lines 16-17 in the new code. So, extending a function in this way is really error prone.

Now, let us see how we do that in UU_AG. Code 5.4 shows the attributes and the top level function needed to implement the generation of the proof. Note that we simply add another set of ATTR and SEM declarations. The system will take care that all SEM declarations are properly merged in generated implementation.

The ATTR sections below state that we have a new attribute called proof. It will contain the validation trace that we want to co-produce.

Code 5.4 :

```
ATTR Spec [          ||proof:Proof]
ATTR Stmt [preC:Expr ||proof:Proof]
```

The SEM sections below tells us how this attribute proof is computed.

Notice that the SEM sections are really separate sections! They do not clutter other sections –for example, the SEM section defining how wp is computed. Neither do we have to patch the code of the existing sections.

Note also that the attributes have to be computed in a certain order. For example the premises cannot be computed before the wp attribute is computed. For example, in Code 5.3 this order is specified in lines 13-15[1].

However, notice that in UU_AG we do not need to specify this order. The system relies on lazy evaluation to automatically discover the right order, which is often very helpful since finding the right order can be non-trivial.

6. CONCLUDING REMARKS

The idea of using attribute grammar formalism to develop a verification tool is not new. For example [7] shows a proof editor as a component of language editor specified in an attribute grammar style. However the combination of the attribute grammar and aspect oriented approach seems to be new. The example shown in this paper is very simple, but we believe that the approach is quite potential. We have been working on the re-implementation of a medium sized verification tool for distributed systems called xMech[1]. Compared to the conventional way used to implement the first version of xMech, we really appreciate the modularity provided by the UU_AG system.

The idea of generating validation code to validate a program's run (Section 5) is closely related to the ideas of *online proof*, *proof carrying code*, and *certifying model checkers* [6]. We are however are not aware of their implementation in the attribute grammar style nor in the aspect oriented style.

Code 5.5 : Constructing Validation Code

```
SEM Spec
  | Hoa  lhs .proof = @s.proof
       s.preC      = @p.itself
SEM Stmt
  | Assign
       local.ruleName = "Assign rule"
       local.premises = [Rule (Valid (Imp @lhs.preC @wp))
                              []
                              "Pred logic"]
       lhs.proof      = Rule (Hoa @lhs.preC @lhs.itself @lhs.q)
                              @premises
                              @rulename

  | Catenation
       local.ruleName = "Sequential rule"
       local.premises = [@s.proof, @t.proof ]
       s.preC         = @lhs.preC
       t.preC         = @t.wp
       lhs.proof      = Rule (Hoa @lhs.preC @lhs.itself @lhs.q)
                              @premises
                              @rulename
    ...
```

[1]For the sake of example, let us pretend that in Code 5.3 Haskell is not lazy.

REFERENCE

[1] A. Azurat and I. S. W. B. Prasetya. A preliminary report on xmech. Technical Report UU-CS-2002-008, Institute of Information and Computing Sciences Utrecht University, P. O. Box 80.089 3508 TB Utrecht The Netherlands, January 2002.

[2] Oege de Moor, Kevin Backhouse, and S. Doaitse Swierstra. First-class attribute grammars. *Informatica (Slovenia)*, 24(3), 2000.

[3] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[4] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland,* volume 1241 of *Lecture Notes for Computer Science,* pages 220–242. Springer-Verlag, June 1997.

[5] Andres Löh, Arthur Baars, and Doaitse Swierstra. *Using the AG* System. downloadable: http://www.cs.uu.nl/people/arthurb/data/AG/AGman.pdf.

[6] G. C. Necula. Proof-carrying code. In POPL 97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 106–119, 1997.

[7] T. Reps and T. Teitelbaum. The Synthesizer Generator: A System for Constructing Language-Based Editors. Texts and Monographs in Computer Science. Springer-Verlag, New York, 1988.

[8] S. D. Swierstra and H. H. Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 256–296. Springer, 1991.