Attribute Grammar Applications in Prototyping LOTOS Tools

Peter van Eijk
University of Twente
PO Box 217, 7500 AE Enschede
The Netherlands
mcvax!utrcu1!pve
(phone) +31 53 893678
(fax) +31 53 333815

Abstract

What is the practical applicability of attribute grammars? As we show in this paper, attribute grammars are at least good enough for the prototyping of fully functional interactive tools. Going from a definition of a language and the functionality of its tools to an attribute grammar is a discipline in need of a systematic approach, for which we give some initial material. As is inevitable when a system is extensively used (in our case the Cornell Synthesizer Generator), this paper also proposes extensions to the attribute grammar formalism and its supporting systems.

1 Introduction

This paper represents, in some way, a view from the trenches. How we prototyped tools contributing to a specification environment for LOTOS is the main topic here. Attribute grammars were chosen because they promised to be a good prototyping approach to language based software development, and the close relation between attribute grammars and the description of tool functions helps ensure the correctness of the tool prototypes. Needless to say, doing some large developments based on attribute grammars has given us some insights in what can be done with them and what their limitations are. Knowing some techniques for structuring attribute grammars is very important. Yet, the methodology of programming with attribute grammars is not fully developed. One of the aims of this paper is to contribute to that methodology. Usage of attribute grammars in our tools is also discussed in another paper [vE89b], which has more emphasis on the functionality of the tools produced.

The attribute system selected for our work is the Cornell Synthesizer Generator (SG) [TR89a,TR89b]. Its important features are that it is a fully functional generator of structure editors employing incremental attribute evaluation. Moreover, the fact that these generated editors work on a window system greatly contributes to their appeal.

The structure of this paper is as follows. We assume some familiarity with attribute grammars and the concept of structure editors. In section 2 we discuss the kind of tool functionality that we want to prototype for LOTOS tools, and show that the concept of structure editing has wide applicability for that. Section 3 is devoted to attribute grammar programming techniques and, by necessity, also comments on limitations and possible extensions. Creating interactive user interfaces is the subject of section 4.

2 Functionality of LOTOS Tools

Most interactive tools can be seen as editors. In an editor the user issues commands to navigate through, manipulate or view the edit-object. For example, the edit-object of a conventional text editor is a sequence of lines of characters. In a database system the edit-object is an entire database and can be accessed through a database query language. Tools that support a specification language operate on expressions in this language and on objects that denote the semantics of these expressions (as in a programming language: the program, its execution state and assertions). The SG generates language specific editors from a high level language specification. Editor specifications are written in a formalism (SSL) that is based on attribute grammars.

LOTOS (Language Of Temporal Ordering Specification) is a language developed for the formal specification of communication protocols and services. It was developed by ISO for the work on Open Systems Interconnection and is now international standard IS8807. LOTOS is based on Milner's CCS (Calculus of Communicating Systems) [Mil80]. Its data type structure is based on the abstract data type language ACT ONE [EM85]. An overview of LOTOS, including specifications and theory, can be found in [vEVD89]. A tutorial on LOTOS can also be found in [BB87].

In process algebraic approaches, such as LOTOS and CCS, the emphasis is on describing systems as behaviours. A behaviour is a sequence of events and event-offers. An event is the result of a *synchronous* interaction of behaviours. A behaviour is described by a behaviour expression. A behaviour expression can be composed by combining atomic event-offers and by combining other behaviour expressions with operators such as choice, parallel, enable, disable etc. Behaviour expressions can also be named and parameterised.

Semantically, a behaviour expression denotes a *transition system* of states and transitions (corresponding to events) between states. At each state of the behaviour a number of events are possible. The set of these events is called the *menu* of that state. If no events are possible, the state is usually called a *deadlock* state. All possible sequences of events form the tree of behaviour, also called the *communication tree*. Each node in the tree is a state, and each edge corresponds to an event, which is possibly parameterised with values. The communication tree is an important concept that is the basis of a number of

tools because it captures the dynamic behaviour of a specification.

Different specifications can be equivalent in a certain sense, e.g. they have the same externally observable behaviour. The typical example in the communication field is a service as opposed to the protocol that should deliver the service. The service only describes, for example, that a certain event on one side is followed by an event on the other side. The protocol also describes *how* this is done. One aspect of *verification* is proving the equivalence of two specifications. The strong semantical foundations of LOTOS allows the development of a theory that describes when two specifications are equivalent in a certain sense. A number of these equivalence relations are discussed in [Bri88a,Bri88b,Mil80]. These equivalence relations do not only have a use in the proof of correctness of a specification, as in the previous protocol example. They can also serve as the basis for correctness preserving transformations for restructuring specifications.

The purpose of tools is to support a design process, based on LOTOS, from the initial specification to the final implementation. After making a formal specification of a system in LOTOS, subsequent steps of this process are aimed at verifying it, refining it and adding implementation decisions. LOTOS based design methods are themselves under development, [VSvS88]. so it is important to facilitate the development of design support tools by rapid prototyping in order to assess these design methods.

Functions of tools can be grouped according to two broad phases in a design process. The first group of functions supports the construction and analysis of specifications. Here we find functions that support the editing of specifications and the checking of properties such as syntax and static semantics, functions for the static analysis (e.g. data type and process structure browsing), and functions for the dynamic analysis of behaviour (e.g. walking through and unfolding of the communication tree of the behaviour, also called simulation). The second group of functions supports the implementation, refinement and correctness proof of specifications. Characteristic of these functions is that they relate specifications to each other. Examples include comparing a service to a protocol specification, or transforming a specification into a differently structured one. This involves test generation and checking, symbolic manipulation, transformation by equivalence laws, and proof assistance.

Existing tools for full LOTOS include the simulator Hippo [vE89a] and the Ottawa simulators. [LOBF88]. LOLA [QPF89] is a transformation tool for most of full LOTOS.

The prime attribute grammar based tool that we worked on is a LOTOS editor. As stated, most interactive tools look like an editor. Nevertheless the functionality of the LOTOS editor extends greatly beyond what is normally perceived to be the functionality of an editor. Besides being a full structure editor and a pretty printer for LOTOS, it also checks the static semantics of the edit-object. It gives facilities for investigating the types of value expressions (LOTOS has overloaded operators) and for browsing through a specification and its error messages in a structural way. Furthermore it partly supports an analysis of the dynamic semantics of a specification (e.g. what does *this* behaviour expression do). The LOTOS editor is generated from approximately 20.000 lines of attribute grammar. It is noteworthy that implementing tool functions for a language feeds back on its definition. For example, the first attempts at describing the static semantics of

LOTOS in an attribute grammar have resulted in fairly extensive changes (clarifications) in the language definition.

The other tools that we worked on served more as prototypes into tool functionality for transforming a specification into an equivalent one. The editors help in conducting a proof, and serve as proof *assistants*. They are research vehicles and help us assess the viability of certain tool functions. The aforementioned tools are described in more detail in [vE89b]. In this paper we concentrate on the attribute grammar formalism.

3 Attribute Grammar Programming Techniques

In this section we discuss how an attribute grammar describing certain tool functionality can be developed. It is structured along the lines of an 'idealised' development of such a grammar. In sequence we discuss concrete syntax, abstract syntax, attributes and attribute equations, transformations, performance and some miscellanea. In the process we discuss features found useful and facilities found missing. So, the section shows attribute grammar programming techniques while at the same time being a critique of the system we used: the Cornell Synthesizer Generator. It should be noted that [TR89a] is also a good introduction into the use of attribute grammars. Furthermore, [KRS82] also discusses a design discipline for attribute grammars, and [vE89b] gives more detail about the attribute grammars for the tools discussed here.

3.1 Concrete Syntax

There are two aspects of concrete syntax for the SG. One is the syntax that is used for parsing a text so that it can be read into an editor. The other is the description of the *unparsing* of edit-objects.

We do not discuss the creation of a concrete syntax here. What we do discuss is the relation with abstract syntax. In the SG, an abstract syntax tree is created by attribution of a concrete syntax tree. This scheme has advantages and disadvantages. The advantage is that the definition of the abstract syntax (on which the semantical tool functions will be defined) is not cluttered up with details that only serve for parsing. The full attribute formalism is available for the translation between concrete and abstract syntax, which gives considerable freedom. Related to that is the fact that the unparsing (or pretty printing) can be described independently from the concrete parsing syntax, and can include information that is computed by attribute rules on that abstract syntax. The main disadvantage is that one needs three descriptions of the syntax (concrete parsing syntax, abstract syntax and unparsing), which can look very similar, and a mapping between these. In the initial development phase this problem can be slightly reduced if the description of the parse syntax is postponed (after all, a structure editor not necessarily has to have a way of reading text).

The treatment of comments in this scheme is also not completely trivial. If comments

are to be retained in the edit-object, they have to be part of the abstract syntax.

The pretty printing description language specifies the placement of unparsed objects relative to one another. An important feature of this language is, paradoxically, that it cannot describe every conceivable layout. This leads to simple and efficiently implementable unparsing schemes. In practice it turns out that a reasonable layout of LOTOS specifications can be described.

3.2 Abstract Syntax

Abstract syntax is the backbone of an attribute grammar based structure editor. It is important that the abstract syntax can be described in such a way that it fulfills the following requirements. The relation between abstract and concrete syntax should be easy to describe, both for parsing and unparsing. The abstract syntax also determines the ease with which structural editing operations can be done. Early structure editor generators only had an implicit concept of resting place (which are the places the selection of the editor can be): every nonterminal of the abstract syntax was a resting place. In the SG, resting places are a separate concept, enabling better compromises between the various requirements on an abstract syntax. The most important requirement is that the abstract syntax should allow a concise description of the attribute rules, as the attribute rules typically form the bulk of an editor specification.

The SG uses term algebras to describe both abstract syntax and attribute values. Simply stated, term algebras define abstract syntax trees and operators to construct them. The wide applicability of term algebras has a number of advantages. For example, attributes can be used as abstract syntax objects, which is useful in presenting derived information in the unparsing, and in computing the result of a transformation (discussed further on). Conversely, an abstract syntax object might be the canonical representation of an attribute value, e.g. an identifier is itself the best representation of an identifier. Finally, the description of 'incomplete', e.g. partially edited, objects is well integrated in the term algebra formalism of the SG. This implies that is is possible to describe attributions that can compute partial checks of partial edit-objects.

How does one derive an abstract syntax from a concrete syntax? In the abstract syntax all terminal symbols that serve solely for parsing purposes, such as keywords and parentheses, are omitted. A concrete syntax can also contain so-called chain productions, that serve to describe operator precedences. These should also be eliminated. It can sometimes be convenient to replicate a nonterminal structure, in order to ease the description of attribute rules. An example in LOTOS is formed by identifier lists. They can denote gate identifiers or value identifiers, which have to be attributed differently.

3.3 Attributes

Attributes and attribute equations are used, for example, for checking static semantics, for computing dynamic semantics, and for computing the layout information and infor-

matives. In the SG, the facility for defining functions over terms is effectively a side-effect free applicative language. One feature of this language is particularly interesting: *attribution expressions*, which allows any term, e.g. an attribute value, to be attributed. This is particularly convenient in the description of functions based on the dynamic semantics. In this way one can put a syntactic object, e.g. a behaviour expression, in an attribute, copy it to a different place, and attribute it there, using an attribution expression.

When is it possible to describe semantics with attribute grammars? It is necessary that a semantical definition can be made constructive, as opposed to a constraint oriented definition (as in: 'the requirement is fulfilled if a solution to these equations exists.').

How does one create a semantical definition with attribute grammars? There are some standard techniques, a large number of which are presented in [TR89a], e.g. how to do redundant parenthesis elimination, and three solutions for checking identifier binding, but do these contribute to a method? Our first approach to a method is the following. A language definition contains a large number of 'objects', some of them syntactical ones, but others introduced to express requirements and meaning in the semantics. Examples of these are environments of identifiers, types of expressions, and conditions. All these objects have to be identified, and then systematically mapped to concepts in an attribute grammar, e.g. abstract syntax nonterminal, attribute, function over terms. For example, for a semantical requirement one expects some attributes to hold and disseminate relevant information, some function to compute the requirement, an error attribute to hold the error message, and a suitable unparsing of the error message.

Of course there are trade-offs possible in the design of an attribute grammar. An important trade-off is between having a function compute some information or distributing the computation over the abstract syntax using attributes. For example, in Pascal a variable declaration can contain the following fragment: `a, b, c: int`. To compute the contribution of this fragment to the environment of defined identifiers one can either apply a function on the identifier list on the left and the type name, or one can thread an attribute representing the type through the list. Another trade-off has to do with incremental performance, an issue that will be discussed in more detail further on. Our name for a technique discussed there is *attribute splitting*, and it involves splitting an attribute that represents a certain composite value in two attributes each representing a part of that value. This is done in the hope that one of the attributes changes less often than the other one so that the cost of incremental reevaluation is reduced.

3.4 Transformations

Transformations in the SG are manually selected and applied. Their effect is to replace the current selection by a new abstract syntax tree object, which is presumably computed from the selection. In that computation, attributes and functions can be used. Transformations have two main applications. They are used to define structural editing operations, such as template invocation ('make this thing into a parallel composition of things'), and, more interestingly, to define semantical transformations. From the (dynamic) semantics we derive *equivalence laws*, which can be implemented as such transformations. The idea

is that we can prove the equivalence of one specification to another if we can transform the first into the second using only equivalence laws. Tools that support this are *proof assistants*. An example of such a transformation is an 'unfold' rule that replaces a process invocation by its defining body. One step beyond this is to use such tools to transform a specification into one that reflects implementation decision. From our experiences in working on such tools we see two possible improvements to the SG facility for defining transformations. One, the SG enables a transformation when the selection of the editor matches a certain syntactical pattern. It would be interesting to also allow 'semantical conditions' on transformations, which can for instance be on the value of a certain attribute. In our experiences, a large number of transformations are always syntactically possible but not semantically, making it hard for the user to find his or her way through a proof. Two, certain transformations *add* information. For example, consider a transformation to change the name of a certain variable in an expression. How is the new name entered? A facility for allowing the user to give a parameter to the transformation would solve that problem.

3.5 Performance

There are two aspects of performance: the time it takes to fully attribute a given abstract syntax tree, and the time it takes to respond to an edit operation (incremental performance). SG generated editors employ incremental reevaluation of attributes, meaning that only those attributes are reevaluated that have changed. Incremental reevaluation is an essential prerequisite for the concept of consistently attributed edit-objects. For instance, in our LOTOS tools, it keeps response times in the 0.5-5 sec range (most of the time), where a full attribution can take minutes for a realistically sized specification.

Incremental performance can vary between functionally equivalent editors. In an earlier section we mentioned *attribute splitting*, a technique for improving incremental performance. In the LOTOS editor, an important example is formed by the environment of process definitions. This environment has two parts: the process headings, and the process bodies, and the environment attribute is split accordingly. The headings are used in checking parameter list correspondence, and the bodies are used in certain dynamic semantics functions. Now if they are represented in a single attribute, each change in the body of one process definition changes the value of the entire environment, which causes all parameter list correspondences (static semantics) to be rechecked. After attribute splitting, a change in a body avoids this superfluous rechecking. Transformations like these can have a major impact on incremental performance, but unfortunately do not appear to be an active research area.

Summing up the performance of the LOTOS editor, there is good news and bad news. The good news is that people are willing to use the system on realistic specifications (approximately 2000 lines). The bad news is that this is the limit given the current version of the SG (3.2) and vintage 1988 hardware (Sun 3/60, 3 Mips, 8 Mbyte memory). Measurements show that the run time is dominated by environment table construction and lookup, even after considerable tuning.

3.6 Miscellanea

SG generated editors can manipulate multiple edit-objects, or buffers as they are also called. We have two uses for that. First, a LOTOS specification can refer to a *standard library*. In our editor, that library is a separate edit-object. Second, we place derived information, such as an analysis of dynamic behaviour (the menu), in separate edit-objects. The facilities of the SG to relate these objects are very limited. In fact, some fiddling in C code is necessary to get the previous functions to work. What one would want, is a facility to describe multiple buffers for edit-objects, and their dependencies, in such a way there is propagation of change between edit-objects. For example, the standard library, or a suitable representation of it, could be an inherited attribute of a specification buffer. Note that the dependencies between buffers are themselves an edit-object. In fact, making buffers first class citizens in the editor specification language may lead to a solution for this. Some work on coupling various *invocations* of editors is reported in [KKM87].

Editors can be interfaced to plain C code, although the current version of the SG makes this only possible, rather than convenient. One use of that is to perform computations on objects that are awkward to express in attribute grammars or whose attribute grammar implementation is inefficient. Examples are large state space exploration and abstract data type simulation. It is important that objects can be shuttled back and forth between the attribute grammar and the C code easily. For this reason these objects should have a simple and implementation independent description. Again, term algebras promise to be a good technique.

4 Interactive User Interfaces

A system like the SG provides a considerable number of interactive facilities almost for free. Having an editor maintain consistently attributed edit-objects is a powerful concept. The attributes can compute interesting information that is available on demand, e.g. select a value expression and request its type. For more global information, there is the concept of alternate *views*, which can be used to provide summary information. Examples of usage are a view to show all error messages, and a view to show all headings of definitions.

What can be improved in the SG is the ability to manipulate multiple buffers related by attribute dependencies, as discussed in the previous section. If this idea is extended to its potential, it might enable a more object oriented style to user interface design. A user interface then allows the manipulation of multiple objects that are related and respond to changes in them, where user invoked operations can create new objects that are placed in their own buffers. A tool that might benefit is a proof assistant. It could manipulate a hierarchy of objects representing either intermediate objects or steps in a proof. The idea would also be of benefit to the integration of external tools, as their inputs and outputs can be seen as objects in the user interface.

5 Evaluation

In this section we summarise our experiences with and comments on attribute grammars in general and the SG in particular.

Basing tool functionality strongly on syntactical structure is a good idea. Attribute grammars are a powerful formalism, and have enabled us to make usable tools. Attribute grammars are a slightly more restricted expression formalism than e.g. Prolog, but they are in general also faster in their implementation. Probably every expression formalism is a different compromise between expressive power and efficiency of implementation. Presumable good facilities for external interfaces can enable a 'best of both worlds' approach, optimised both for ease of description and speed of execution. For example, in one of our experiments we have coupled a state space exploration tool to a proof assistant.

The distinguishing strong point of the SG is its concept of generating editors that maintain consistently attributed edit-objects. A nice property of the description language of the SG, is that it makes no distinction between the syntactical domain, e.g. the edit-object, and the semantical domain, used for the attribute types. This enables, for example, the facility of attribution expressions. Transformations are a good idea for manually applying equivalence rules. Suggestions to extend them with parameters and conditions have been discussed. The interface to external tools is there, but needs integration with methods to build those other tools. The most promising potential for extension is the introduction of a more object oriented style, improving the handling of multiple related edit buffers and the interface to external tools.

6 Acknowledgements

The editors discussed were largely developed by Axel Belinfante, Erik Slotboom and Jan Tretmans. Tim Teitelbaum's group at Cornell University responded rapidly with fixes to our bug reports. Douglas Hofstadter provided inspiration for the composition of this paper.

References

[BB87] T. Bolognesi and H. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[Bri88a] H. Brinksma. *On the Design of Extended LOTOS.* PhD thesis, Twente University of Technology, Enschede Netherlands, 1988.

[Bri88b] H. Brinksma. A theory for the derivation of tests. In K. Sabnani and S. Aggarwal, editors, *Proceedings of the eighth international conference on protocol specification, testing and verification*, pages 63–74, North-Holland, Amsterdam, 1988.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[KKM87] G. E. Kaiser, S. M. Kaplan, and J. Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 20(4):58–67, nov 1987.

[KRS82] K. Koskimies, K.-J. Raiha, and M. Sarjakoski. Compiler construction using attribute grammars. *SIGPLAN Notices*, 17(6):153–159, June 1982.

[LOBF88] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri. An interpreter for LOTOS, a specification language for distributed systems. *Software - Practice and Experience*, 18(4):365–385, April 1988.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[QPF89] J. Quemada, S. Pavon, and A. Fernandez. Transforming LOTOS specifications with LOLA - the parameterised expansion. In K. J. Turner, editor, *Formal Description Techniques - Proceedings of the FORTE 88 Conference*, pages 45–54, North-Holland, Amsterdam, 1989.

[TR89a] T. Teitelbaum and T. W. Reps. *The Synthesizer Generator - A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.

[TR89b] T. Teitelbaum and T. W. Reps. *The Synthesizer Generator Reference Manual: Third Edition.* Springer-Verlag, New York, 1989.

[vE89a] Peter van Eijk. The design of a simulator tool. In P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 351–390, North-Holland, Amsterdam, 1989.

[vE89b] Peter van Eijk. LOTOS tools based on the cornell synthesizer generator. In H. Brinksma, G. Scollo, and C. A. Vissers, editors, *Proceedings of the ninth international symposium on protocol specification, testing and verification*, North-Holland, Amsterdam, 1989.

[vEVD89] P.H.J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS - Results of the ESPRIT/SEDOS Project.* North-Holland, Amsterdam, 1989.

[VSvS88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and specification style in formal descriptions of distributed systems. In K. Sabnani and S. Aggarwal, editors, *Proceedings of the eighth international conference on protocol specification, testing and verification*, pages 189–204, North-Holland, Amsterdam, 1988.