Specification of Interpreters and Debuggers using an extension of Attribute Grammars

Gail E. Kaiser

26 November 1985

CUCS-196-85

Department of Computer Science
Columbia University
New York, NY 10027

Table of Contents

List of Figures

Abstract

Recent research in programming environments has focused on the generation of software tools from specifications. Several specification mechanisms have been proposed, and the most successful of these has been attribute grammars. Attribute grammars have been successfully applied to compile-time tools such as type checkers and code generators, which perform static analysis of the program, but have hitherto seemed unsuited to the description of run-time tools such as interpreters and debuggers that involve dynamic interaction with the user of the programming environment. This report describes an extension to attribute grammars that solves this problem. The extended attribute grammars are suitable for the specification of all semantics processing performed by single-user programming environments.

1. Introduction

Integrated programming environments are rapidly replacing the traditional tools used by programmers to edit, compile and debug their programs. The key components of an integrated programming environment are a standard user interface and a common database. A large number of prototype and teaching programming environments are now built using structure editing technology, which supports both of these features [35, 8, 29, 15, 1, 5, 12, 31, 30, 3, 11, 13, 4]. Each of these environments consists of an integrated collection of tools that may be applied incrementally as the programmer writes and tests her programs. In some cases, the tools are automatically applied without the explicit intervention by the programmer. For example, type checking and symbol resolution are performed automatically as the program is created and modified; code generation and some code optimization may also be performed incrementally.

The early structure editor-based programming environments, such as the Cornell Program Synthesizer [34], were entirely hand-coded. Then Medina-Mora demonstrated in [28] that a structure editing environment can be generated from an *environment description*. A program called an *environment generator* combines an environment description with the common editor kernel to produce the desired programming environment. The person who writes the environment description is called the *implementor* of the programming environment while a person who uses the programming environment to write her programs is called a *user*.

An environment description has two components, the syntax description and the semantics description. The *syntax description* includes the abstract syntax (or structure) of the programming language and the user interface (or concrete syntax) for programs in the language. It is now well-established that a syntax description is written in a declarative notation similar in style to a context-free grammar or BNF. It is very easy to write a syntax description for a conventional programming language. It might take two days for an implementor to write the syntax description for Pascal and as much as two weeks for the more complex syntax of Ada. A syntax description alone can be used as an environment description if no semantics processing is required. An environment generator can combine the syntax description with the editor kernel to produce a pure syntax-directed editor that supports program editing and enforces correct syntax.

The *semantics description* specifies all the processing performed by the environment, *i.e.*, everything the environment does that is not among the standard facilities provided by the editor kernel. In other words, the syntax description describes the program database and the semantics description describes the tools that operate on the database. The semantics processing of a programming environment is performed by a collection of tools that are knowledgeable about the particular programming language. The tools are often divided into two categories: the tools that handle static (compile-time) semantics and the tools that handle dynamic (run-time) semantics. The implementor of a programming environment describes the semantics processing in terms of the static and dynamic properties of the programming language. *Static properties* can be determined by inspection of the program while *dynamic properties* reflect the interaction between the user and the programming environment. Compilation is described in terms of static properties, while the run-time support is characterized by dynamic properties.

The description of static and dynamic properties is very difficult. In contrast to the syntax description, there is no commonly accepted form for the semantics description of a structure editing environment, and the development of a semantics description for a relatively simple environment such as the Gnome teaching environment [18] can take many months. However, there are two methods of semantics description, action routines and attribute grammars, that have been more widely used than their competitors in the generation of integrated programming environments.

Action routines were proposed by Medina-Mora in [10]. The semantics processing is written as a set of routines in a conventional programming language. A particular routine is associated with each rule in the abstract syntax. The corresponding routine is automatically invoked by the editor kernel when an editing command is applied to a node. Action routines were adapted from the semantic routines associated with parser generators such as YACC [7].

Attribute grammars were originally proposed by Knuth [26], have become a standard technique for compiler generation [9, 14, 16, 27, 17] and were adapted to interactive programming environments by Reps in [6] and [32]. The semantics processing is written as a set of attribute definitions associated with each rule in the abstract syntax. An incremental evaluator automatically re-evaluates all attributes whose values may have changed as the result of a modification of the syntax tree.

Unfortunately, none of the previously proposed methods fulfill all the requirements of an implementor of a programming environment. The basic problem with action routines is that the design, implementation and debugging of the routines is tedious and error-prone compared to the ease with which a syntax description can be developed. The problem with attribute grammars is even more severe, as the capabilities of attribute grammars are generally limited to static properties and attribute grammars have not been successfully applied to the description of dynamic properties.

1.1. Two New Paradigms

One way of viewing action routines and attribute grammars is as the two extremes of a spectrum of approaches to solving the semantics problem. The action routines paradigm is at the informal, imperative end of the spectrum while the attribute grammar paradigm is at the formal, declarative end. Two newly developed paradigms fall at intermediate points in this spectrum.

The *tree-oriented action routines* paradigm described in [2] is an improvement on Medina-Mora's action routines. The extensions include an implementor model that replaces the framework provided by the original action routines paradigm and a tree-oriented imperative programming language that replaces the conventional programming languages previously adopted for writing action routines. Unfortunately, the improved paradigm still falls far short of matching the ease with which attribute grammars express the static semantic properties of conventional programming languages, to a large degree because the semantics are still expressed in a more-or-less general-purpose imperative programming language. This led to the desire for a declarative notation in a style similar to attribute grammars but that also supported the expression of dynamic properties.

Attribute grammars are unsuited to the description of dynamic properties because of the *derived* nature of the attributes. The value of each attribute is calculated from the source program and other attributes. By definition, it cannot depend in any way on the history of modifications to the source or of the execution of the source. This feature is exactly what is desired for static semantic checking and for code generation, but it is completely inappropriate for the dynamic properties of dynamic semantics. In the dynamic semantics case, the semantic state depends on the history of the program execution.

This problem has been solved by the new *action equations* paradigm, which is presented in detail in [25]. The primary contribution of this paradigm is that it supports the expression of *history* or dynamic properties in a style similar to attribute grammars. This is done by embedding *equations* similar to attribute definitions in an event-driven architecture. The *events* activate equations in the same sense that user commands trigger action routines. The editor kernel orders the evaluation of active equations according to the commands invoked by the user and the dependencies given by the equations. Equations that apply at all times are not attached to particular events and these correspond exactly to attribute grammar definitions.

This event-driven nature is one of the crucial differences between action equations and attribute grammars. Another way of stating this difference is that action equations support multiple events while attribute grammars support only one event, the ***change*** event. In the attribute grammar paradigm, the ***change*** event can be received from the user and propagated by attribute definitions to other definitions. In the action equations paradigm, the ***change*** event, standard events and implementor-defined events can all be received from the user and/or sent from action equations to other equations.

Another crucial distinction between the two paradigms is that attribute grammars are applicative while action equations support certain non-applicative mechanisms. Attribute

grammars require that attribute definitions are applicative. This means that an attribute definition is re-evaluated only when the program changes, and then the attribute definition is restricted to replacing the old value with an entirely new value. These applicative restrictions are removed in the new paradigm. An action equation may be re-evaluated due to the receipt of an event, and then the equation is permitted to directly modify the current value of the attribute. Together, these side-effects and the added dimension of multiple events make it possible for action equations to support the expression of dynamic properties in the same style in which attribute grammars support the expression of static properties.

It should be noted that it is theoretically possible but not usually desirable to describe dynamic properties using attribute grammars. First, the notion of 'source program' is redefined to include a script that represents all interactions between the user and the programming environment. Second, stacks and other relatively complex data structures are employed to simulate the necessary non-applicative mechanisms. Essentially, the implementor treats the attribute grammar notation as a general programming language and hacks out a solution. This is rarely done in practice.

2. Compile-Time Semantics

This section discusses the addition of static semantic processing to structure editor-based programming environments. The phrase *static semantics* refers to the context-sensitive properties of programming languages. The static semantics processing of conventional programming languages includes symbol resolution, type checking, flow analysis for anomaly detection and source-level optimizations, and code generation.

The *A*ttribute *C*onstraint *L*anguage (ACL) is one specific notation for writing action equations. This section introduces ACL with particular emphasis on how its functionality follows and improves on the functionalities of the attribute grammar notations used by two important proponents of attribute grammars. First, action equations are compared to attribute grammars as adopted by Reps, and then the paradigm is compared to the extended attribute grammars advanced by Johnson in [22] and [23]. This section demonstrates that (1) action equations are suitable for expressing the static semantic properties of programming languages and (2) in the context of static semantic checking, the differences between attribute grammars and action equations are relatively minor.

2.1. Comparison with Reps

There is a mechanical procedure to translate from Reps' attribute grammar notation into ACL. Reps' notation consists of attribute declarations, attribute definitions and the semantic functions that are invoked in attribute definitions. The translator copies the declarations and functions as is, since the syntax description languages and the expression languages are assumed to be equivalent. The only actual conversion occurs for the attribute definitions.

Two classes of definitions are considered. The first defines **error** attributes, and the second defines all other attributes. The **error** attribute of each node is a distinguished attribute whose

value is a string; if the string is non-null, then the string represents the error message for the node and this string is displayed to the user of the programming environment. In this case, the definition is translated to an action equation called an *assertion*; in the second case, the definition is translated into an ACL *constraint*.

There are two forms for an attribute definition that defines an error attribute. One form is the if-then-else (or a more general conditional), and the other form involves an encapsulated semantic function that returns a string. The first form is translated into an ACL assertion as shown in Figure 2-1; the translation of the second form is similar.

```
error := if <boolean expression>
         then ""
         else <non-empty string>

ASSERT <boolean expression>
ERROR <non-empty string>
```

Figure 2-1: Translation Into Assertion

The translation from attribute grammar into ACL notation is even simpler in the case of definitions that do not define an error attribute. There is only one form. *<expression>* is any expression, including the invocation of a semantic function. This is translated to an identical ACL constraint as shown in Figure 2-2.

<attribute name> := *<expression>*

<attribute name> := *<expression>*

Figure 2-2: Translation Into Constraint

The implementation of action equations depends on an incremental evaluator that is an adaptation of Reps' incremental attribute re-evaluation algorithm. When the adapted evaluator is applied to an environment description that is the result of this translation, Reps' asymptotic optimality result continues to hold. This is demonstrated in [25].

2.2. Comparison with Johnson

In an attributed syntax tree derived from a standard attribute grammar, a given node may participate in a rule instance in one of only two ways: it may be a child or it may be a parent. Johnson extends the attribute grammar formalism to include another kind of rule in which a node may participate. A *non-local rule* is an association of nodes in a syntax tree that may directly communicate attribute values among one another. A non-local rule permits nodes that are not adjacent in the tree to be grouped together, and attribute definitions are associated with such groupings as well as with standard rules. The implementation of non-local rules is discussed in [24].

This extension provides a significant advantage over the standard formalism followed by Reps. Certain kinds of changes propagate directly to the affected portions of the program through non-local links defined in this manner. In contrast, the standard attribute grammar

formalism requires the same kinds of changes to propagate only through local links. This results in more work for the incremental evaluator and more work for the implementor. Each piece of the abstract syntax that may be on the route through which such information might flow must be associated with copy rules or other devices in order to pass along the information.

However, Johnson's extension is only a partial solution to the problem of non-local dependencies. This is because non-local links are constructed as the two halves of each link become available in the syntax tree. This means a link can be used for change propagation only after both the left hand side and the right hand side of the non-local rule have been constructed. It is not possible to, for example, construct a link between one half of the rule and a stub for the other half.

The ability to manipulate stubs in this way would be very useful for linking identifier uses with their definition sites. When an identifier use is entered into the program tree, the equations would attempt to link it to a definition site. If a corresponding entity had not yet been defined, the identifier use would instead be linked to a stub. When the identifier definition is entered later, it would replace the stub; then the non-local link could be used immediately to, for example, change the status of each use from 'notdeclared' to 'declared'. The action equations paradigm provides three features -- sets, membership constraints and propagate equations -- that together support a complete solution to the non-local dependency problem.

The *set* is a new kind of node, on a par with non-terminal, terminal and sequence node. A set is a homogeneous structure composed of an arbitrary number of substructures (nodes). The syntactic definition of a set includes a *key* that names a component that is common to every substructure. A set may contain only one element with the same value for the key component. ACL provides a primitive "lookup" function that takes a set and a value for the key and returns the corresponding element, if there is one; if not, it returns a meta node. Sets with keys support a limited sort of relation, in a sense similar but not identical to the relations of relational databases. (In contrast, Horowitz extends attribute grammars with full relational capabilities and achieves the same result [21].)

A node becomes a member of a particular set in much the same way that a node is assigned a value. In the latter case, an *assignment constraint* describes the value of the node while in the former case a *membership constraint* describes the composition of the set. Assignment constraints are equivalent to the attribute definitions of Reps' notation while membership constraints are similar to Johnson's non-local rules and permit the implementor to describe the contents of uselists.

The ACL notation for a membership constraint is shown in Figure 2-3. *<database>* is an instance of any production defined as a set in the syntax description. *<key>* is any legal value for the key component. *<field name>* names any component of the tuple whose type is specified as set or sequence. *<node>* is an address expression that denotes a node. The nodes specified in a membership constraint equation are automatically maintained in the relation defined by the constraint.

```
RELATION <database>, <key>, <field name>, <node>
```

Figure 2-3: Membership Constraint

The *propagate equation* describes the flow of information along the local or non-local dependency defined by a membership constraint and maintained in a set. The simple case of a propagate equation has the form shown in Figure 2-4 (the general case permits multiple destinations). The equation means that the node denoted by *<destination>* depends on the node denoted by *<source>*. When the value of the source changes, the action equations for the destination must be re-evaluated. *<destination>* is a computed address, that is, an address expression that may denote any node in the attributed syntax tree.

```
<source> PROPAGATES TO <destination>
```

Figure 2-4: Propagate Equation

Relations described by membership constraints implement the stubs mechanism. The use sites and definition sites that share the same printname (type, *etc.*) are associated in a tuple. It is not necessary for a definition site to exist before use sites can take part in the relation. When a definition site is created, the non-local link represented by the relation can immediately be used to propagate this information.

3. Run-Time Semantics

This section discusses the addition of dynamic semantic processing to structure editor-based programming environments. The phrase *dynamic semantics* refers to the run-time behavior of programs. This area includes interpreters, run-time environments and symbolic debuggers. Code generation and code optimization are not included because the object code is determined from static inspection of the source program. The code generated for a program does not (in general) change during the execution of the program. The *dynamic properties* of a program are those properties that may change during the execution of the program; the *static properties* of the program must not change during its execution.

This section extends the ACL notation to express dynamic properties. It demonstrates how the new features support (1) interpretation; (2) run-time support; and (3) interactive execution of programs.

3.1. Implementation of Interpreters

Action equations support the implementation of conventional control constructs using events and propagate equations. An *event* usually corresponds to a user command and may be standard or implementor-defined. A collection of equations may be *attached to* an event as shown in Figure 3-1. This means that after *<event>* is triggered and before any new event is triggered, each equation attached to *<event>* is active; the equations are passive at all other times. Only active equations may be evaluated.

When a propagate equation is attached to an event, the other kinds of equations attached to the

<event> --->
    <equation$_1$>
    . . .
    <equation$_n$>

Figure 3-1: Attaching Equations To An Event

same event are evaluated first. When a propagate equation is not attached to any event, the other kinds of equations that are also not attached to any event are evaluated first. In either case, if the evaluation results in a change to the source attribute of the propagate equation, or if *<source>* is not given, then the actual propagation occurs.

A propagate equation may include an event, as illustrated in Figure 3-2. The result of the propagation is that *<event>* is sent to each *<destination>*. If no event is named, then the implicit ***change*** event is sent instead. Receipt of a named event has the effect of activating all equations attached to that event. Receipt of the implicit ***change*** event has the effect of activating all equations that are not attached to any particular event.

<source> PROPAGATES *<event>* TO *<sequence of destinations>*

Figure 3-2: Propagate Equation

Sequencing, selection, iteration and branching can be expressed with simple combinations of these two facilities. This is demonstrated in [25]. For example, the description of iteration shown in Figure 3-3 involves a circularity between the sources and destinations of the propagate equations that describe the flow of control through the loop. This is potentially a problem, because the action equation evaluator is an adaptation of Reps' incremental attribute evaluation algorithm [33], which does not guarantee termination in the presence of circularities in the dependency graphs.

The solution to this problem has two parts. First, explicit dependencies are excluded from the dependency graphs; instead, these dependencies are calculated dynamically as needed. The second part of the solution involves modifying Reps' algorithm to support lazy propagation in order to handle the explicit circular dependencies. The main idea of *lazy propagation* is that each equation attached to an event is evaluated exactly once for each receipt of the event and only on receipt of the event. These equations are never evaluated in response to the ***change*** event. Lazy propagation prevents unintentional infinite cycling due to dependencies among the equations attached to an event while supporting intentional infinite loops required by the semantics of the programming language.

3.2. Run-Time Support for Program Execution

An interpreter or a run-time environment for compiled code must implement the memory management required for subroutine invocation and data manipulation. This is supported by (1) extending the notation to permit any nodes, not just attributes, to take part in action equations and (2) augmenting ACL with the view definition notation described by Garlan in [20]. The first change permits action equations to modify the source program as well as the values of attributes, introducing a dangerous source of side-effects. However, this ability is used not to modify the

```
/* abstract syntax for loop statement */

loop => initialization: STATEMENT
        condition: EXPRESSION
        re-initialization: STATEMENT
        body: STATEMENT

/* interpretation of loop statement */

    run -->
        PROPAGATES run TO initialization

    /* the "run ON initialization, re-initialization"
       notation is simply a short-hand for writing each
       of the two events separately with the equations
       attached to those events */

    run ON initialization, re-initialization -->
        PROPAGATES run TO condition

    /* when the run event is received by the condition
       component, its result attribute is set by an
       action equation such as
       "result := <semantic function>"
       that is associated with the particular instance of
       the EXPRESSION class
       if the new value of result is false, then the
       propagation does not occur */

    run ON condition -->
        PROPAGATES run TO if condition.result then body
    run ON body ->
        PROPAGATES run TO re-initialization
```

Figure 3-3: Loop Statement Syntax and Semantics

source program constructed by the user but to modify alternative views of the source program that represent the internal state of the run-time environment.

A *static view* consists of a collection of rules that define an abstract syntax. The abstract syntax that defines the standard representation of the source program is only one static view of the programming language. For example, the source view might describe the standard abstract syntax for a procedure while the execution view might define an activation record. This is illustrated in Figure 3-4. During program execution, a copy of the execution view of the procedure (a new activation record) would be pushed onto the run-time stack at procedure invocation. The stack of activation records would be represented as a component of the execution view of the program; the heap would be another component of this view. [25] demonstrates that the entire internal state of a run-time environment can be described in this manner by a collection of static views.

```
VIEW "source"

procedure => name: identdef
             parameters: seq of vardef
             variables: seq of vardef
             body: seq of STATEMENT

vardef => id: identdef
          type: TYPE

VIEW "execution"

/* the default qualification is 'ReadWrite' */

procedure => name: identdef qualified 'ReadOnly'

             /* combines parameters and variables */

             locals: seq of vardef

             /* "ref to" indicates a symbolic reference
                to a node that exists elsewhere in the
                syntax tree
                the reference terminal node type can be
                used to extend a rigid syntax tree to an
                arbitrary graph structure */

             pc: ref to STATEMENT
             body: seq of STATEMENT qualified 'ReadOnly'

vardef => id: identdef qualified 'ReadOnly'
          value: block
```

Figure 3-4: Two Static Views of Procedure Syntax

3.3. Interactive Execution and Debugging

The only requirements of interactive execution and debugging that cannot be satisfied by the facilities previously described are those that involve direct interaction with the user of the programming environment. These requirements are met using the delay equation and Garlan's display views [19].

A *display view* describes how a static view is displayed on a terminal screen. It also supports editing commands such as **create** a node and **exit** the current node with the cursor. These editing commands are always applied to the underlying attributed syntax tree in terms of the displayed representation. Display views are sufficient to support both display and modification of the internal state of the programming environment. During debugging, the user can display the **source** and **execution** views of her program in different windows.

The same mechanism can be used for stream input/output, where the stream is represented as a component of the **io** view of the program. Implementation of the **write** statement is extremely

easy. The value to be written is concatenated to the end of the I/O stream, which is automatically displayed on the screen. The read statement is slightly more difficult and requires the addition of the delay equation. The function of the delay equation is to request a particular event and delay the other active equations until the user sends the event. The user does this by selecting the command that corresponds to the event. The delay equation implements the read statement by suspending program execution until the user has entered another line of input.

The ACL delay equation has the form illustrated in Figure 3-5, where *<node>* is an address expression and *<event>* is any standard or implementor-defined event that can be selected by the user as a command. The evaluation of a delay equation means that the currently active equations cannot be evaluated until *<event>* has been received by *<node>*. The active equations are suspended when the delay equation is evaluated and awakened when the event occurs at the node. This happens when the user performs some action that has the effect of sending the standard or implementor-defined event to the appropriate node.

```
DELAY UNTIL <event> AT <node>
```

Figure 3-5: Delay Equation

The delay equation is the means for implementing debugging facilities such as breakpoints and singlestepping. Program execution would be suspended at a marked statement or after every statement, respectively, until the user invokes the **continue** command.

4. Implementation

This section explains the implementation of the action equations paradigm. The implementation consists of two parts. The first part is the translation from a semantics description into an executable form. The translation process is entirely algorithmic and can be performed without human intervention. The second part is the run-time environment for the executable form. This run-time environment is provided as extensions to an editor kernel that supports the tree-oriented action routines paradigm.

4.1. The Translation to an Executable Form

The executable form consists of three parts: a collection of procedures, a collection of graphs and a collection of action routines. Together, these represent all the information provided by the implementor in a semantic description. Each procedure evaluates an individual action equation. Each graph specifies the dependencies among a set of action equations. Each action routine determines the subset of the graphs that represent the currently active action equations. One or more action routines are automatically invoked by the editor kernel in response to each activity performed by the user of the programming environment. The graphs selected by the action routine(s) are then used by the run-time environment to order the invocation of the procedures.

There is a one-to-one correspondence between procedures and action equations. Each procedure evaluates a particular action equation. The generated procedures are in the tree-oriented programming language understood by the editor kernel.

There is also a one-to-one correspondence between local dependency graphs and certain sets of action equations. The local dependency graphs are used by the run-time environment to drive the invocation of the procedures. The graphs determine which action equation procedures should be invoked and the order in which they should be invoked.

The third part of the executable form is a collection of tree-oriented action routines. There is zero or one action routine for every type of node defined in the syntax description. There is exactly one action routine for every type of node that is associated with one or more action equations in the semantics description. The role of the action routines is to select a subset of the graphs according to certain criteria. This subset represents the currently active action equations, while the graphs excluded from the subset represent the currently passive equations. Only the graphs in the active subset are considered by the run-time environment when ordering the procedures.

4.2. The Run-Time Environment

The run-time environment also has three parts: an incremental evaluator, a signal queue and a request set. Together with the editor kernel for the tree-oriented action routines, these provide complete run-time support for the executable form of a semantics description. The incremental evaluator is the central component of the run-time environment. It uses the local dependency graphs to order the invocation of action equation procedures. The signal queue and the delay set handle the implementation of two specific features of the action equations paradigm. The signal queue maintains the sequence of events triggered by the user and by propagate equations. The request set maintains the unordered set of events requested by delay equations.

The *incremental evaluator* is an adaptation of the incremental attribute evaluator described by Reps in [32]. Given a set of dependency graphs, the evaluator constructs the model. The *model* is another graph that represents all the active interdependent sets for the entire program, while a local dependency graph represents only a single interdependent set for one type of node. The incremental evaluator performs a topological sort of the vertices in the model to order the invocation of the corresponding procedures.

The *signal queue* drives the triggering of action routines, as described in [2]. Whenever the user of the programming environment selects a command, the command interpreter adds one or more signals corresponding to that command to the end of the signal queue. Similarly, whenever a propagate equation sends an explicit event, signals corresponding to the event are also added to the end of the signal queue. A *signal* is a record that includes all the parameters to an action routine. The action routine manager repeatedly removes the first signal from the queue, selects the appropriate action routine, and invokes the action routine with the parameters given in the signal. After the completion of each action routine execution, the action routine manager invokes the incremental evaluator with the local dependency graphs selected by the action routine. The incremental evaluator continues until quiescence, and then returns control to the action routine manager. The command interpreter does not accept another command from the user until the action routine manager indicates that the signal queue is empty.

The *request set* watches for commands corresponding to requested events. A *request* is a record that includes a representation of the desired event and the signals corresponding to the active action equations suspended by the delay equation that initiated the request. Whenever the signal queue empties, the action routine manager compares the most recent command to the requests in the set. If there is a match, then the action routine manager returns the suspended signals to the signal queue. These signals are then processed before the action routine manager activates the command interpreter.

5. Conclusions

The primary result of the research described in this paper is the development of a new paradigm within which implementors can easily develop the run-time components of structure editor-based programming environments. The action equations paradigm represents a significant improvement over previously proposed paradigms. The major improvement over action routines is the declarative style of notation. This raises the level of abstraction at which implementors can describe semantics processing and drastically eases the debugging and enhancement of their programming environments. The major improvement over attribute grammars is the simple means for expressing dynamic as well as static properties of the programming environment. This allows the implementor to specify both code generation and the run-time support for program execution.

Action equations can also be used for generation of compilers and run-time environment as separate tools, apart from programming environments, in the same manner that attribute grammars are currently used for compiler generation. The implementation in this case is much easier, since the action equations are evaluated once, until quiescence, with respect to the whole program rather than incrementally as the program is modified.

Acknowledgement

This research was conducted while the author was a member of the Gandalf project at Carnegie-Mellon University. Research on Gandalf is supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] Cyril N. Alberga, A.L. Brown, G.B. Leeman, Jr., Martin Mikelsons and Mark N. Wegman.
A Program Development Tool.
In *Conference record of the Eighth ACM Symposium on Principles of Programming Languages*. January 1981.

[2] Vincenzo Ambriola, Gail E. Kaiser and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, CMU Department of Computer Science, August 1984.

[3] Vincenzo Ambriola and Carlo Montangero.
Automatic Generation of Execution Tools in a Gandalf Environment.
The Journal of Systems and Software 5(2), May 1985.

[4] James Archer, Jr. and Richard Conway.
COPE: A Cooperative Programming Environment.
Technical Report TR 81-459, Cornell University Department of Computer Science, June 1981.

[5] Norman M. Delisle, David E. Menicosy and Mayer D. Schwartz.
Viewing a Programming Environment as a Single Tool.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April 1984.

[6] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages.* January 1981.

[7] T.A. Dolotta and R.C. Haight.
PWB/UNIX -- Overview and Synopsis of Facilities.
Technical Report, Bell Laboratories, June 1977.

[8] Veronique Donzeau-Gouge, Gilles Kahn, Bernard Lang and B. Melese.
Documents Structure and Modularity in Mentor.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April 1984.

[9] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October 1984.

[10] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5), September 1981.

[11] Ross S. Finlayson.
EDT A Syntax-Based Program Editor Reference Manual.
Technical Report 83-245, Stanford University Computer Systems Laboratory, July 1983.

[12] Charles N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, Daniel L. Stock.
The POE Language-Based Editor Project.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April 1984.

[13] Christopher W. Fraser and David R. Hanson.
High-Level Language Facilities for Low-Level Services.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. January 1985.

[14] Mahadevan Ganapathi and Charles N. Fischer.
Description-Driven Code Generation using Attribute Grammars.
In *Conference record of the Ninth ACM Symposium on Principles of Programming Languages*. January 1982.

[15] E.R. Gansner, J.R. Horgan, D.J. Moore, P.T. Surko, D.E. Swartwout, J.H. Reppy.
SYNED -- A Language-Based Editor for an Interactive Programming Environment.
In *Digest Of Papers Spring CompCon '83*. November 1982.

[16] Harald Ganzinger, Robert Giergerich, Ulrich Moncke and Reinhard Wilhelm.
A Truly Generative Semantics-Directed Compiler Generator.
In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*. June 1982.

[17] Harald Ganzinger and Robert Giegerich.
Attribute Coupled Grammars.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*. June 1984.

[18] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. April 1984.

[19] David B. Garlan.
Flexible Unparsing in a Structure Editing Environment.
Technical Report CMU-CS-85-129, CMU Department of Computer Science, April 1985.

[20] David B. Garlan.
Representational Transformations for Tools in Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, 198x.
In progress.

[21] Susan Horowitz and Tim Teitelbaum.
Relational Databases and Attribute Grammars: a Symbiotic Basis for Editing Environments.
In *Proceedings of the SIGPLAN '85 Symposium on Language Issues in Programming Environments*. June 1985.

[22] Gregory F. Johnson and Charles N. Fischer.
Non-syntactic Attribute Flow in Language Based Editors.
In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*. January 1982.

[23] Gregory F. Johnson.
An Approach To Incremental Semantics.
PhD thesis, University of Wisconsin at Madison, 1983.

[24] Gregory F. Johnson.
A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors.
In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. January 1985.

[25] Gail E. Kaiser.
Semantics of Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, May 1985.
Technical Report CMU-CS-85-131.

[26] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2), June 1968.

[27] K. Koskimies, K.-J. Raiha and M. Sarjakoski.
Compiler Construction Using Attribute Grammars.
In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction.* June 1982.

[28] Raul Medina-Mora.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March 1982.

[29] Martin Mikelsons.
Interactive Program Execution in Lispedit.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*. August 1983.

[30] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2), May 1985.

[31] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. April 1984.

[32] Thomas Reps.
Generating Language-Based Environments.
PhD thesis, Cornell University, August 1982.
Technical Report TR 82-514.

[33] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems (TOPLAS) 5(3), July 1983.

[34] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September 1981.

[35] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
IEEE Computer , April 1981.