GAEA Action Equations Paradigm

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Simon M. Kaplan,
University of Illinois
Department of Computer Science
Urbana, IL 61801

June 1988
(revised December 1988)

CUCS-352-88

Abstract

This technical report consists of two papers describing the GAEA action equations paradigm. *Incremental Dynamic Semantics for Language-based Programming Environments* explains why attribute grammars are not suitable for expressing dynamic semantics and presents *action equations*, an extension of attribute grammars suitable for specifying the static and the dynamic semantics of programming languages. It describes how action equations can be used to generate language-based programming environments that incrementally derive static and dynamic properties as the user modifies and debugs the program. *Rapid Prototyping of Concurrent Programming Languages* extends this technology to a concurrent framework. It describes an (unimplemented) system that generates a parallel interpreter for the language and provides run-time support for the synchronization primitives and other facilities in the language.

Prof. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award. Prof. Kaplan is supported in part by a grant from AT&T Corporation.

Incremental Dynamic Semantics for Language-based Programming Environments

Gail E. Kaiser
Carnegie Mellon University

Abstract

Attribute grammars are a formal notation for expressing the static semantics of programming languages — those properties that can be derived from inspection of the program text. Attribute grammars have become popular as a mechanism for generating language-based programming environments that incrementally perform symbol resolution, type checking, code generation and derivation of other static semantic properties as the program is modified. However, attribute grammars are not suitable for expressing dynamic semantics — those properties that reflect the *history* of program execution and/or user interactions with the programming environment. This article presents *action equations*, an extension of attribute grammars suitable for specifying the static and the dynamic semantics of programming languages. It describes how action equations can be used to generate language-based programming environments that incrementally derive static and dynamic properties as the user modifies and debugs the program.

Categories and Subject Descriptors: D.2.3 [**Software Engineering**]: Coding — *program editors*; D.2.5 [**Software Engineering**]: Testing and Debugging — *debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments; D.3.1 [**Programming Languages**]: Formal Definitions and Theory — *semantics*; D.3.4 [**Programming Languages**]: Processors — *interpreters, translator writing systems and compiler generators*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs — *specification techniques*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages — *operational semantics*
General Terms: Algorithms, Design, Experimentation, Languages
Additional Key Words and Phrases: Attribute grammars, dynamic semantics, generation of language-based environments, interpreters

This research was supported by a Fannie and John Hertz Foundation Fellowship, by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and by ZTI-SOF of Siemens Corporation, Munich, Germany.

Author's current address: Columbia University, Department of Computer Science, New York, NY 10027.

1. Introduction

This article addresses the processing performed by language-based environments (LBEs). This processing is performed automatically and incrementally (in the background) as the user writes and tests his programs. It requires an internal representation that consists of the program itself plus additional information maintained by the environment during program construction and execution. This information represents two kinds of semantic properties, static and dynamic. *Static properties* are those that can be determined by inspection of the program, while *dynamic properties* reflect the interaction between the user and the environment. The implementor of an LBE describes its processing as derivation and manipulation of these properties. For example, symbol resolution, type checking and code generation involve static properties, while interpretation, run-time support and symbolic debugging involve dynamic properties.

Recent research has focused on the generation of LBEs from descriptions. Several mechanisms have been proposed for specifying the processing performed by the environments, and the most successful of these have been action routines, attribute grammars and denotational semantics. Action routines are written as a collection of imperative subroutines. Consequently, it has proved difficult for an implementor of an environment to anticipate all possible interactions among these subroutines that may result in adverse behavior. Attribute grammars are written in a declarative style and the implementor need not be concerned with subtle interactions because all interactions among semantic equations can be determined automatically. Attribute grammars have been successfully applied only to the description of static semantics, and have hitherto seemed unsuited to the description of dynamic semantics. Denotational semantics is a formal mechanism that provides direct means for defining certain dynamic properties, notably interpretation. Denotational semantic specifications have not been extended to other dynamic processing such as interactive debugging nor to incremental detection and reporting of static semantic errors.

This article proposes an extension to attribute grammars that supports incremental processing of both static and dynamic semantics. The extended paradigm is called *action equations*. Action equations are written in a notation that retains the flavor of attribute grammars but adds an easy means to express dynamic properties as well as static properties. The extensions to attribute grammars include attaching particular semantic equations to *events* that represent user commands and supporting dependencies among events as well as among attribute values. The applicative nature of attribute grammars is relaxed, allowing attributes to be treated as variables and permitting modification in addition to replacement for changing the values of attributes. Together, these extensions are sufficient to support incremental processing of dynamic semantics.

2. Generation of Language-Based Environments

LBEs are an alternative to the traditional tools used by programmers to edit, compile and debug their programs. The key components of an LBE are a standard user interface and a common program representation. Many programming environments have been built using structure editing technology, which supports both of these features. The user interface consists of some mixture of template editing and text editing (supported by incremental parsing [23, 35, 63]); the program is represented as a parse tree or abstract syntax tree, where each node may be decorated with attributes. Some of the best known LBEs are Mentor [11], Interlisp [60], the Program Synthesizer [59], Gandalf [24], Pecan [50], and Rational [2]. Each of these environments consists of an integrated collection of tools that (1) can be viewed as a single tool [7] and (2) may be applied incrementally as the programmer writes and tests his programs. In some cases, the tools are automatically applied without explicit intervention by the programmer. For example, type checking and symbol resolution may be performed automatically as the program is created and modified; code generation and some code optimization may also be done incrementally.

The early LBEs were hand-coded. Then several environment generators were developed, including ALOE [45], Metal [12] and the Synthesizer Generator [53]. An *environment generator* is a program that combines an environment description with the editor kernel to produce the desired LBE. The *editor kernel* provides the facilities common to all environments, such as window management and language-independent tree manipulation commands, while the *environment description* includes all the information specific to the desired programming environment. The person who writes the environment description is called the *implementor* of the environment while a person who uses the environment to write his 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 of the programming language and the user interface (or concrete syntax) for the language. This information is normally provided as some form of context-free grammar. A syntax description alone is sufficient 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.

2.1. Semantics Description

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. Although an LBE is a single tool, the semantics processing of an LBE is performed by what is conceptually a collection of tools and tool fragments that are knowledgeable about the particular programming language. The collection can be subdivided into tools that handle static semantics and tools that handle dynamic semantics.

The *static semantics* of a program involve those properties that, by definition, cannot change during its execution. The static properties of a conventional, lexically scoped programming language include symbol resolution, type identification and the object code generated for the program. For example, the set of identifiers defined in a particular program, the mapping between identifier uses and identifier definitions, and the types assigned to particular identifiers and expressions are all in the realm of static semantics.

Consider the program fragment in figure 2-1. The program states that the variable **a** is declared to be of type **integer**, but the program also states that the variable **a** constitutes the conditional expression of the **if** statement. The static semantics of this programming language require that a variable has the same type over its lifetime and that the condition expression of every **if** statement is of type **boolean**. Thus there is a static semantic error in the program. A programming environment that included a type checking tool would warn the user of this error.

```
VAR a: integer;
...
IF a THEN
    ...
ELSE
    ...
```

Figure 2-1: Type Checking Example

If the tool were *incremental*, it would warn the user as soon as the error could be detected. If the user had first entered that the type of **a** is **integer**, and later used **a** as the conditional expression for the **if** statement, then the error would be detected and reported immediately after the user entered the conditional expression. If the user had instead added **a** to the list of variables, without indicating its type, then used **a** as the conditional expression, and finally returned to the variables list to state that **a** is an **integer**, then the error would be caught immediately after the user entered this type information.

The *dynamic semantics* of a program involves the derivation and manipulation of those properties that may change during the execution of the program. The dynamic properties of a conventional language include the assignment of values to particular storage locations and the maintenance of the current focus of execution behavior (*i.e.*, the program counter). The area of dynamic semantics includes run-time support and symbolic debuggers as well as interpreters.

The same programming environment sketched above might include an interpreter as well as the type checking tool. The job of the interpreter is to directly execute programming language statements. The interpreter does not need to perform type checking or other static semantics processing, since these functions are handled by other tools. The interpreter performs the activities that the program fragment is defined to do according to the dynamic semantics of the programming language.

Consider the corrected program fragment in figure 2-2. The interpreter would begin execution of the **if** statement by getting the current value of the **a** variable from the store (which binds variable locations to values). If **a** does not have a value, this would be reported to the user as an error (or the environment could ask the user to enter a value). If **a** does have a value, the interpreter would then check whether it is 'true' or 'false'. If true, the interpreter would execute the then statement; if false, it would execute the else statement.

```
VAR a: boolean;
...
IF a THEN
    ...
ELSE
    ...
```

Figure 2-2: Interpretation Example

This behavior does not depend on whether the interpreter tool is incremental or non-incremental. By analogy to the type checking tool, an 'incremental' interpreter might follow along behind the user, executing the program as it is typed, as in VisiProg [25]. Instead, we think of an 'incremental' interpreter as one that permits the user to select, for example, the then part of the **if**, the entire **if** statement, or an arbitrary program unit, and give a command to interpret that unit. In a non-incremental environment, the user would have no choice but to commence execution of the program at the beginning.

Specifying static and dynamic semantics is very complex. In contrast to the syntax description, there is no commonly accepted paradigm for the semantics description of a programming environment. There are two major schools that support different methods of specifying the semantics processing of an LBE: action routines and attribute grammars. Both methods support interactive semantics processing, *i.e.*, the integrated, incremental, non-sequential, structure-oriented computing style described by Notkin in his thesis [46]. Such interaction with the user is an essential requirement for modern programming environments. A third major school — denotational semantics — disagrees with this claim, and supports another method of specifying semantics processing for non-incremental, sequential programming environments. These three methods are briefly described here and are explained in detail in the references.

The first school uses *action routines*, which were proposed by Medina-Mora in his thesis [45] for use in LBEs [18, 24]. Action routines are based on the semantic routines used in compiler generation systems such as Yacc [31]. The semantics processing is written as a set of routines in either a conventional programming language or in a special purpose programming language designed for writing action routines [1]. A set of routines is associated with each production in the abstract syntax, one for each user command (such as Create, Delete, Enter, Exit, Execute, *etc.*) that can be applied to an instance of that production. The corresponding routine is automatically invoked by the editor kernel when an editing command is applied to a node in the syntax tree representing the program.

The second major group uses *attribute grammars*, which were introduced by Knuth [43] for specifying the context-sensitive properties of programming languages. Attribute grammars are an alternative to semantic routines in compiler-compilers [14, 20]. The generation of LBEs from attribute grammars [32, 53] was proposed by Demers, Reps and Teitelbaum [8]. The semantics of the programming language are written as (1) a set of attribute declarations associated with each symbol; and (2) a collection of semantic equations — each associated with a particular production — that define the values of the attributes of the symbols on the production's left and right hand sides. The values of the attributes are determined by *evaluating* all the semantic equations as a set of simultaneous equations. During program editing, an incremental algorithm [33, 52] automatically reevaluates those attributes whose values may have changed as the result of a subtree replacement (editing operation).

The third school uses *denotational semantics*, originally promoted by Scott and Strachey [55] for formal reasoning about programs. The semantics of the programming language are written as a set of formal definitions — associated with each production in the abstract syntax — that specify the denotation of each language construct in terms of the environment (which binds variable identifiers to locations), the store, and the denotations of other productions. Several research groups have applied denotational specifications to generation of compilers [6, 48, 49] and interpreters [5], but none of these systems are effective in an incremental programming environment. However, Johnson has recently developed an incremental interpreter/debugger for GL [34], an expressional language based on denotational semantics.

Other methods have been proposed (*e.g.*, [3, 10, 13, 51]), but none fulfill all the requirements of an LBE. The basic problems are:

- The design, implementation and debugging of action routines, or any other procedural mechanism, is tedious and error-prone compared to the ease with which a syntax description can be developed.
- The capabilities of attribute grammars, denotational specifications and the various other declarative methods are generally limited to a relatively small subset of the processing performed by modern programming environments.

This article describes a new method, *action equations*, that augments attribute grammars with mechanisms taken from action routines. The 'action' of action equations comes from association of user commands (or actions) with action routines, while the 'equations' comes from the semantic equations of attribute grammars. Action equations achieve a synthesis with most of the advantages of both paradigms but few of their disadvantages. Action equations were originally presented in the author's thesis [36], and additional details can be found there.

2.2. Overview of Action Equations

Attribute grammars are not suitable for the description of dynamic semantics because of the inherently static nature of their attributes. The value of each attribute is equated to a specified function of the program text and other attributes. It cannot depend in any way on the history of modifications to the program text or of the execution of the program. By definition, attribute grammars are inappropriate for expressing dynamic semantics.

The primary contribution of the action equations paradigm is that it supports the expression of *history* or dynamic properties in a style based on attribute grammars. This is done by embedding rules similar in form to semantic equations in an event-driven architecture. *Events* correspond to user commands and activate their attached equations in the same sense that, in the action routines paradigm, commands trigger the associated action routines. The editor kernel orders the evaluation of active equations according to the commands invoked by the user and the dependencies among attributes and events as defined by the equations. Equations that apply at all times are not attached to particular events and these correspond exactly to the semantic equations of attribute grammars.

Those action equations attached to events, however, should not be confused with semantic equations. Attribute grammars are applicative: an attribute is a variable in the sense of algebra's simultaneous equations but not in the sense of conventional programming languages. An attribute is reevaluated only when the program is modified, and then the semantic equation replaces the old value with an entirely new value.

These restrictions are relaxed for action equations, as follows. First, an equation may be reevaluated due to the selection of an event, so an attribute may be reassigned many times even though the program has not changed. Second, an equation is permitted to define the new value of the attribute as a modification of its previous value in the case of *aggregate* (or composite) values, such as the symbol table and the run-time stack. This second extension to pure attribute grammars has recently appeared in several 'attribute grammar' systems [26, 54]. Together, these side-effects and the added dimension of events make it possible for action equations to support the expression of dynamic semantics in a style similar to how attribute grammars support the expression of static semantics.

3. Description of Dynamic Semantics

3.1. Action Equations

```
goal symbol ::= component_1: type_1
                ...
                component_n: type_n

/* A production consists of a non-terminal goal symbol, followed by "::=",
followed by a list of components.  A component is defined by a name, followed
by ":", followed by its type. */
```

Figure 3-1: Production

Action equations are associated with particular productions in the syntax description in the same manner as the semantic equations of attribute grammars. The productions define the composition of the non-terminal nodes in the syntax tree representing the program. Figure 3-1 illustrates the context-free grammar notation adopted for action equations. This notation is based on the Interface Description Language [44, 58] (IDL) developed as part of the Ada implementation effort, and has been used previously in DOSE [39], an interpretive LBE generation system. Only the abstract syntax is shown; the concrete syntax, or 'syntactic sugar', is omitted throughout this article. This syntax description notation is not in any way integral to action equations, and any other context-free grammar notation could be substituted — the only difficulty might be a less readable semantics description.

A non-terminal goal symbol is associated with a list of components, where each component has a name and a type. The same symbol may appear as the goal of multiple productions, indicating several alternative derivations; for example, a STATEMENT may be an if statment, a while statement, a compound statement, *etc.* The type of a component may be a non-terminal symbol, a terminal symbol or a sequence. Terminal symbols correspond to the primitive types of conventional programming languages. The set of terminal symbols available is specific to the implementation of the environment generation system, but would typically include identifier, integer, real, boolean, string and text. The sequence constructor is in contrast to the tail recursive method of defining lists using non-terminal, terminal and empty symbols. In each case, the sequence definition includes the element type.

In addition to alternative sets of components, a group of attributes and events may be

```
goal symbol { attribute_1: type_1
              ...
              attribute_m: type_m
              event_1, ..., event_p }

/* The attributes and events associated with the goal symbol are declared
between braces "{}". */
```

Figure 3-2: Attribute and Event Declarations

associated with each goal symbol as depicted in figure 3-2. Each node defined by this symbol is decorated with this set of attributes, which represent the current values of its properties; the events are attached to action equations that manipulate these properties. Attributes are typed in the same manner as components, where the type is given as a non-terminal symbol, a terminal symbol or a sequence.

```
production

    equation_1
    ...
    equation_n
```

Figure 3-3: Action Equations

The action equations associated with a particular production describe the semantics processing for each node that is an instance of the production. A production and its action equations are depicted in figure 3-3. As in attribute grammars, the order equation$_1$, ..., equation$_n$ shown does not imply any sequencing among these equations, or that they should be evaluated in this or any other particular order.

```
location := function(attributes and terminals)
```

Figure 3-4: Assignment/Constraint Equation

There are five kinds of action equations: assignments, conditionals, constraints, delays and propagates. The assignment and constraint equations both have the form shown in figure 3-4, and the distinction is due to whether or not the equation is attached to an event. *Assignments* are, by definition, attached to events while *constraints*, by definition, are not; this is explained in the next section. For both, the right hand side denotes some function of attribute values and terminal node values; these values are called the arguments or *inputs* of the equation. The value computed by this function, called the result or *output*, is placed in the location given on the left hand side of the equation. The location is typically the name of an attribute, in which case the equation is identical in form to the semantic equation of attribute grammars.

The location may also be given as an address expression applied to an attribute name. This permits the modification of a previously calculated attribute value. This divergence from the attribute grammar paradigm has a significant implication: Attribute modification, together with events, make it possible for an attribute to reflect the history of program modification and/or execution. Otherwise, each attribute would of necessity be derived solely from the program text as explained previously.

The third alternative is for the location to be an address expression applied to a node in the syntax tree. Thus, the equation directly modifies the program as seen by the user, which is not possible in the pure attribute grammar paradigm. It might be argued that modification of the program by the environment should never be possible, on the grounds of the 'principle of least astonishment'. This argument assumes the programmer does not expect the programming environment to change his program, but exactly the opposite is true in transformational programming environments, formal [4, 19, 47, 57] or informal [62]. There is no reason the programmer should expect less from an LBE; in particular, manipulation of the program text by action equations is one mechanism for implementing transformations.

```
If expression
Then equation(s)
Else equation(s)
```

Figure 3-5: Conditional Equation

The *conditional* equation consists of an expression and two sets of equations, as depicted in figure 3-5. The conditional equation specifies that when the expression is true, the first set of equations must hold; when the expression is false, the (optional) second set is applicable. All conditional equations can be expressed instead by permitting conditional expressions within other kinds of equations, and are thus only convenient syntactic sugar that provide no new meaning.

3.2. Events

The propagate and delay equations, as well as the distinction between assignments and constraints, requires a discussion of events. Events correspond roughly to user commands. There are two kinds of events, standard events and implementor-defined events. Each *standard event* coincides with a primitive operation accessible to the user, including at least:

Create	Replace the current placeholder with a newly created instance of a specified language construct.
Delete	Remove the subtree rooted at the current node from the syntax tree and replace it with a placeholder.
Clip	Save a copy of the current subtree in a register.
Insert	Replace the current placeholder with a copy of a previously clipped subtree.
Enter	Move the editing cursor to a specified child of the current node.
Exit	Move the editing cursor to the parent of the current node.

The same set of standard events may appear in the semantic description of every environment just as the same set of standard commands are available in every environment. In contrast, an *implementor-defined event* is an identifier introduced by the implementor for a particular environment. An implementor-defined event corresponds to a user command available only in the specific environment, such as Execute or CrossReference a Pascal program.

```
production

    equation_1
    . . .
    equation_n

    event_1 -->

        equation_1,1
        . . .
        equation_1,m

    . . .

    event_p -->
        equation_p,1
        . . .
        equation_p,q

/* When equations are attached to an event, the event name is given first,
followed by "-->", followed by the equations written in arbitrary order. */
```

Figure 3-6: Attaching Equations to Events

Each action equation associated with a particular production may or may not be *attached to* a particular event. Some action equations associated with a particular production may be attached to one particular event, other action equations associated with the same production may be attached to a different event, and still other equations may not be attached to any event at all. Figure 3-6 illustrates a number of equations associated with the same production: equation$_1$ through equation $_n$ are not attached to any event, equation$_{1,1}$ through equation$_{1,m}$ are attached to event$_1$, and so on, through equation$_{p,1}$ through equation$_{p,q}$ attached to event$_p$.

```
goal symbol ::= component1: type
                ...
                componentn: type

    eventa On component1 -->
        equations

    eventb On component1 -->
        equations

    ...

    eventz On componentn -->
        equations

/* When the "On" keyword appears, the inherited event is associated with the
named component; otherwise, the synthesized event is associated with the goal
symbol of the production. */
```

Figure 3-7: Inherited Events and Equations

The semantic equations of attribute grammars may define the value of an attribute associated either with a component of the production (the corresponding attribute is *inherited*) or with its goal symbol (the attribute is *synthesized*). Events are similarly inherited or synthesized. The events shown in figure 3-6 are associated with the goal of the production, and thus synthesized. An inherited event, with its attached equations, is associated with a component name as shown in figure 3-7. In this case, the event name must be declared for each goal symbol that is a legal type for the component. Unlike the attributes of attribute grammars, the same event may appear in both productions defining a node and thus may be both inherited and synthesized. Further, the same event may be inherited multiply with respect to the same production, due to multiple associations of the same event with incidentally the same element of a sequence (for example, "event$_a$ On component$_n$[i] -->" and "event$_a$ On component$_n$[j] -->" where the ith element is also the jth). The multiple sets of equations attached to the event are concatenated, as if they had not been associated with different productions and/or different component descriptions.

Action equations that are not attached to a particular event fill the same role as the semantic equations of attribute grammars in the sense that they may be reevaluated when the program changes. It does not matter which particular user command caused the program modification, since all are treated as subtree replacements. These equations are said to be *permanently active*. In contrast, the collection of action equations attached to a particular event are *active* only when the event is explicitly selected by a user command or explicitly propagated by a propagate equation, explained shortly. These equations are *passive* at all other times. Only active equations may be evaluated, and an equation activiated by an event immediately becomes passive again after its evaluation. The collection of action equations attached to an event describe the semantics processing, or tool operation, for the user command that corresponds to the event.

An assignment equation is attached to an event. When activated by selection of the event, it computes the value denoted by its right hand side and assigns this output to the location on its left hand side. A constraint equation cannot be attached to an event. Whenever an input to its right hand side changes in value, it updates the location on its left hand side to maintain the equality. The distinction is necessary because constraints must always hold, as invariants, while assignments are evaluated exactly once when activated. Constraints are typically reevaluated in response to subtree replacements, but may also be reevaluated when an assignment changes the value of an input to a constraint.

```
Propagate event To destination
```

Figure 3-8: Propagate Equation

The user explicitly selects an event by moving the editing cursor to the node and entering the command corresponding to the desired event. Propagation of events from one node to another is done with a *propagate* equation, as depicted in figure 3-8. When activated, the equation propagates the given event to the indicated destination node. This has the effect of activating certain equations associated with the production that defines the destination node, in particular, all those equations attached to the named event; if this set is empty, then no new equations are activated. As with the arguments of semantic equations in attribute grammars, the destination is normally restricted to the children, siblings, parent and other ancestors (reached through uplevel addressing [42]). However, it is also possible to propagate from an identifier definition to its use(s) or from a use to its definition(s), as described in the next section.

```
Delay Until event At receiver
```

Figure 3-9: Delay Equation

The final kind of action equation is the *delay* equation, which has the form shown in figure 3-9. When activated, a delay equation suspends all currently active equations until the named event is selected for the indicated receiver node. Like the destination node of the propagate equation, the receiver node is restricted to the parent, ancestors, siblings and children of the current node. When the event is selected with respect to the receiver node, the previously suspended equations are reactivated. The event may be selected either by a user command or by a propagate equation resulting from a user command. In the latter case, the previously suspended and now activated equations are in addition to any equations that may be active at the time of the event. The receiver node of a delay equation is optional; if omitted, then the delay equation refers to the selection of the named event when the editing cursor is at any node.

When a group of equations are attached to the same event, both as synthesized and inherited, there is a specific ordering among the different kinds of equations. In particular, any delay equations are evaluated first and, in effect, simultaneously. Thus all other equations attached to the same event are suspended by the delay equation(s); if there are multiple delays, then all the named events must be selected for their receivers to reactivate the suspended equations. If there are no delay equations, then any assignments and conditionals attached to the event are evaluated in any order (except as noted below) consistent with the dependencies among inputs and outputs of the assignments and the inputs of the expression parts of the conditionals. Any constraints, and conditionals not attached to the event, whose inputs are among the outputs of these assignments are also evaluated if and only if the outputs are different than their previous values. Any propagate equations are evaluated last; any equations activated by these equations are, in effect, activated simultaneously.

This ordering among action equations is complicated by the conditional equation. Neither the then part nor the else part equations are themselves activated until after the expression has been evaluated, as soon as possible consistent with the partial ordering described above; an alternative semantics would evaluate these expressions as late as possible, but some such restriction is necessary to avoid non-deterministic behavior. After the value of the expression has been determined, then the appropriate set of equations are simultaneously activated, and the above rule applies regarding the previously active equations as well as the newly activated equations.

The main components of action equations paradigm have now been introduced. Section 4 describes the application of action equations to the description of programming language control constructs such as conditional statements, loops and procedure calls. Section 5 considers interactive execution of programs, including stream input/output and some typical features of symbolic debuggers. Section 6 discusses the translation and run-time support algorithms for generation of LBEs from action equations.

4. Description of Control Structures

4.1. Flow of Control

```
if { Execute, Continue }

if ::= condition: EXPRESSION
       thenpart: STATEMENT

    Execute -->
        Propagate Execute To condition

    Continue On condition -->
         If condition.value
         Then Propagate Execute To thenpart
         Else Propagate Continue To self

/* The if symbol declares two events, Execute and Continue.  The if production
defines two components, condition and thenpart.  EXPRESSION and STATEMENT are
each defined by several alternative productions, not shown.
"Component.attribute" accesses the named attribute of the named component.
Self always indicates the node representing the goal of the associated
production, in this case the if node, as opposed to one of its components or
attributes.  */
```

Figure 4-1: If Statement Syntax and Semantics

Figure 4-1 demonstrates the use of implementor-defined events and propagation of events in a simple description of interpretation. The implementor defines the Execute event to specify the execution of an if statement. When the Execute event is applied to an instance of the if production, the propagate equation selects the Execute event for the condition child of the if node. After any semantics processing involving the condition node are completed (including for example the setting of its value attribute), then the condition child propagates the Continue event to itself (the condition child). This Continue event activates the conditional equation. If the value of the value attribute is true, the Execute event is propagated to the thenpart child. If not, the if statement has completed execution, and the Continue event is propagated to itself (the if statement). Thus, the implementor-defined Continue event fills the role of the continuation of denotational semantics.

```
= { value: boolean
    Execute, Continue }

= ::= operand1: EXPRESSION
      operand2: EXPRESSION

    Execute -->
        Propagate Execute To operand1

    Continue On operand1 -->
        Propagate Execute To operand2

    Continue On operand2 -->
        Propagate Continue To self

    Continue -->
        value := (operand1.value == operand2.value)

/* Value is an attribute of the = symbol — as well as every other EXPRESSION
symbol.  Terminal symbols such as boolean are given in italics.  */
```

Figure 4-2: Action Equations for = Production

Events and equations for one conditional expression, the = production, are shown in figure 4-2. When the Execute event is propagated to the = operator, the two operands are computed in order and then the value attribute of the = node is set to the result of comparing the two operands. Calculation of expression values does not, however, necessarily require this rather cumbersome action equations apparatus. Purely applicative expressions are handled in a natural way by pure attribute grammars, as demonstrated by Reps' and Teitelbaum's desk calculator [53], so this is not discussed further in this article. Expressions involving (potentially recursive) function calls and (multiple assignment) variables require a run-time stack, as discussed later in this section.

```
compound ::= body: sequence of STATEMENT

    Execute -->
        Propagate Execute To body[1]

    Continue On body[any] -->
        Propagate Execute To body[next]

    Continue On body[last] -->
        Propagate Continue To self

/* The event declarations are omitted, as they are in further examples.  The
sequence type is indicated in italics.  "Component[N]" refers to the Nth element
of the sequence component; "Component[any]" refers to any element of the
sequence.  Next accesses the element following the current one, if any, while
last refers to the last element of the sequence.  */
```

Figure 4-3: Compound Statement Syntax and Semantics

Figure 4-3 illustrates how event propagation works for a compound statement (*i.e.*, a sequencer). The basic idea is that the Execute event propagates from the compound statement to the first statement in the body of the compound statement, from the first statement to the next statement in the body, *etc.* In the case of the last statement where multiple inherited events "Continue On body[any]" and "Continue On body[last]" both apply, both attached equations are executed. But "Propagate Execute To body[next]" has no effect since body[next] evaluates to nil.

```
compound ::= body: sequence of STATEMENT

    Execute -->
        If body = nil
        Then Propagate Continue To self
        Else Propagate Execute To body[1]

    Continue On body[any] -->
        Propagate Execute To body[next]

    Continue On body[last] -->
        Propagate Continue To self
```

Figure 4-4: Compound Statement Syntax and Semantics, Revised

This discussion of the compound statement, and the previous example involving the conditional statement and = operator, have been simplified in that they do not consider the possibility that the body of the compound statement is empty, the condition and/or the thenpart of the conditional statement is missing, or one or both operands of the = operator are meerly placeholders, respectively. The analogous issue arises in the semantic equations of attribute grammars, and is solved there by requiring the implementor to provide *completing* productions, which define the value of the attributes for every potential placeholder. Action equations also take this approach, and the implementor must explicitly treat the possibility of empty sequences and missing components. The compound statement example is revised accordingly in figure 4-4, where nil denotes an empty sequence; the rest of the examples in this article could be completed similarly, but this is not done to keep the examples simple.

```
goto ::= label: identifier

    Execute -->
        Propagate Execute To @label.defsite

labeled ::= label: identifier
            body: STATEMENT

    Execute -->
        Propagate Execute To body

/* The "@" operator dereferences the defsite attribute of the label component
to access the actual definition node elsewhere in the syntax tree. */
```

Figure 4-5: Goto Statement Syntax and Semantics

In order to describe the semantics of branch statements, some mechanism is needed to find the destination of the branch. This is done through identifier definition-use links. Several extensions to attribute grammars have been proposed [9, 27, 28] that improve the efficiency of incremental attribute evaluation by linking the definitions and uses for each identifier. A change in an attribute value at a definition site is propagated along the links to dependent attributes at its use sites. Any one of these schemes can be used as the basis for propagating an event from the goto statement to the corresponding labeled statement as depicted in figure 4-5.

Figure 4-6 shows how the operation of a general loop statement is described using action equations. In this example, the initialization is performed first. Then the condition is tested. If true, the body of the loop is executed. Now reinitialization, condition testing and the body are repeated until the condition becomes false.

Notice that the propagate equations in this example denote a circular dependency. The

```
loop ::= initialization: STATEMENT
         condition: EXPRESSION
         body: STATEMENT
         reinitialization: STATEMENT

    Execute -->
        Propagate Execute To initialization

    Continue On initialization, reinitialization -->
        Propagate Execute To condition

    Continue On condition -->
        If condition.value
        Then Propagate Execute To body
        Else Propagate Continue To self

    Execute On body -->
        Propagate Execute To reinitialization

/* Multiple components (initialization and reinitialization) for inherited
events is introduced as shorthand, meaning the event is selected at either
node.*/
```

Figure 4-6: Loop Statement Syntax and Semantics

condition propagates to the body, the body propagates to the reinitialization and the reinitialization propagates to the condition. Although circular attribute grammars are problematical for non-incremental evaluation [15] and rarely handled by incremental evaluators (work by Walz and Johnson is a notable exception [61]), circularities among propagate equations pose no difficulties. If the user of a generated environment writes an infinite loop, then the propagation never terminates, to preserve correct dynamic semantics processing; if the loop does terminate, then the propagation terminates accordingly.

4.2. Procedure Call and Return

A likely syntax description for a procedure definition, with its formal parameters and local variables, is shown in figure 4-7. The Execute event and attached equations for the procedure production are omitted, since they are essentially identical to those for the compound statement. The procedure symbol has an AR attribute that acts as a template for the procedure's activation record during execution. Frame is a non-terminal symbol, where the details of any particular frame node are computed by constraints just as is done by attribute grammars. For example, the size (in bits, bytes or words) of each formal and local might be computed from its type and then its offset within the activation record determined by the cumulative size (and required

```
procedure { AR: frame }

procedure ::= name: identifier
              formals: sequence of vardef
              locals: sequence of vardef
              body: sequence of STATEMENT

vardef { offset: integer
         size: integer }

vardef ::= id: identifier
           type: TYPE

/* STATEMENT and TYPE are each defined by several alternative productions, not
shown. */
```

Figure 4-7: Procedure Definition Syntax

alignments). This would require each implementor to define a suitable representation for each datatype in his language [56]. One alternative would be to represent each data item as a node; this is much less efficient at execution time but much more expedient at environment description time.

```
program { stack: sequence of frame }

call ::= name: identifier
         actuals: sequence of EXPRESSION

    Execute -->
        program.stack := Insert (@name.defsite).AR # program.stack
        Propagate Execute To actuals[1]

    Continue On actuals[any] -->
        <storage for parameter in top stack frame> :=
            actuals[any].value
        Propagate Execute To actuals[next]

    Continue On actuals[last] -->
        <storage for program counter in top stack frame> := self
        Propagate Execute To @name.defsite

/* Stack is an attribute of program.  The parentheses cause the higher
precedence ".AR" attribute access to apply to the result of the "@" operator.
"#" adds a new element to a sequence.  The determination of <storage for ...
in top stack frame> depends on the description of frames, not shown.  */
```

Figure 4-8: Call Statement Syntax and Semantics

The syntax and semantics of the procedure call statement are given in figure 4-8. The run-time stack is represented as a sequence of frame nodes maintained as an attribute of the program node. Execution begins by applying the internal version of the user-level Insert command to insert a copy of the procedure's activation record at the top of the stack. The actual parameter expressions are then executed, and their resulting values stored in the corresponding slots in the activation record. After the last parameter is available, a reference to the call statement is saved as the program counter and then execution propagates to the procedure definition.

Remember that the apparent circularity of program.stack on both sides of the equation is not a problem, since it occurs in an assignment rather than a constraint. Such 'circularities' are necessary for maintaining history information, where the new value of an attribute is computed by directly modifying its old value. Potential circular dependencies among constraints are handled as in attribute grammars, by separating into in and out attributes where the synthesized out attribute is the appropriate function of the inherited in attribute.

```
return ::=

    Execute -->
        program.stack := Delete program.stack[1]
        Propagate Continue To <program counter in top stack frame>

/* The return production has no components.  For a function rather than a
procedure, the corresponding return would have an EXPRESSION component. Access
to <program counter in top stack frame> depends on the frame mechanism. */
```

Figure 4-9: Return Statement Syntax and Semantics

Figure 4-9 gives the equations for execution of a return statement. The top stack frame is removed from the stack using the Delete command, and the continuation propagates to the original call statement.

5. Interactive Execution and Debugging

5.1. User Input/Output

Figure 5-1 illustrates one mechanism for representing sequential I/O, for either the terminal display and keyboard or ASCII files. For simplicity, each channel consists of both an input stream and an output stream, where each stream is a sequence of buffered text lines. Standard input and standard output are combined in the first channel.

```
program ::= ...
            IO: sequence of channel

channel ::= name: identifier
            input: sequence of text
            output: sequence of text

/* The other components of a program are omitted. */
```

Figure 5-1: Input and Output Streams

```
write ::= expr: EXPRESSION

    Execute -->
        Propagate Execute To expr

    Continue On expr -->
        program->IO[1]->output := program->IO[1]->output # expr.value
        Propagate Continue To self

/* "Node->component" accesses the named component of the named node, where the
node is either the goal symbol of the production or an ancestor. */
```

Figure 5-2: Write Statement Syntax and Semantics

Figure 5-2 gives the syntax and semantics of a simple write statement. When the Execute event is applied to an instance of the write production, the Execute event is propagated to compute the value of the expression. On the continuation, the text representation of this value (as determined by the implementation of the underlying environment generation system) is concatenated to the end of the output stream. The output stream is automatically redisplayed on the screen after every update. Various kinds of *unparse schemes* have been proposed for defining the concrete syntax necessary for displaying the program [24, 29, 53] or distinct views of the program [21, 50]. The action equations paradigm assumes the availability of one of these mechanisms for display purposes.

The read statement is slightly more difficult and requires a delay equation. The first equation attached to the Execute event for the read production, given in figure 5-3, requests the user to select the Create event to add a new last element to the input stream. The delay equation has the effect of suspending program execution until the user has entered a new line of input by appending to the sequence of text lines that represents the standard input. Only then is the last (new) element of the input sequence stored as the value of the variable given in the read

```
read ::= variable: identifier

    Execute -->
        Delay Until Create At program->IO[1]->input[last]
        <storage for variable> := program->IO[1]->input[last]
        Propagate Continue To self

/* <storage for variable> uses whatever mapping the implementor defines for
the environment and store, such as a stack of frames as described in the
previous section. */
```

Figure 5-3: Read Statement Syntax and Semantics

statement.

5.2. Program Suspension and Continuation

```
break ::=

    Execute -->
        Delay Until Continue At self
```

Figure 5-4: Break Statement Syntax and Semantics

The delay equation is also instrumental in specifying debugging facilities such as breakpoints and singlestepping. Figure 5-4 shows hows a breakpoint might be described. This example follows the precedent set by Feiler in his thesis [17] (and elsewhere [16]) as to how the user specifies a breakpoint before or after a particular statement. It assumes that the programming language has been extended by a special break statement. The user designates a breakpoint by inserting a break statement at the desired position in the program text. The interpreter suspends program execution when the Execute event is propagated to the break node, presumably by an equation for some other node.

The user continues from a breakpoint by entering the Continue command when the editing cursor is pointing to the break statement. Selecting the Continue event at some other position in the program would activate the equations attached to the Continue event for the corresponding production, effectively starting up a separate execution thread at that position. A conditional breakpoint might be defined by adding an expression to the break statement and enclosing the delay equation inside a conditional equation.

```
program { singlestep: boolean }

program ::= ...

    Singlestep -->
        singlestep := not singlestep

STATEMENT

    Execute -->
        If program.singlestep
        Then Delay Until Resume

/* Singlestep is an attribute of the program symbol.  Associating a collection
of action equations with the STATEMENT symbol is introduced as a shorthand for
separately associating the collection with each of the alternative STATEMENT
productions. */
```

Figure 5-5: SingleStepping

The description of singlestepping is similar. Figure 5-5 depicts Singlestep as an implementor-defined event that toggles singlestepping on and off, by changing the value of the singlestep attribute. The delay equation is associated with every STATEMENT production. If singlestep mode is on, then the interpreter suspends before the execution of each statement, until the Resume event. Since no receiver is specified in the delay equation, it does not matter where the editing cursor is when the user enters the Resume command. When the user selects the Resume event, the interpreter awakens and continues execution with the current statement.

```
trace ::= variable: identifier

    Execute -->
        program->IO[2]->output :=
            program->IO[2]->output
            # variable # " = " # <value of variable> # '^M'
        Propagate Continue To self

/* The second I/O channel is designated by the implementor for tracing
variables. <value of variable> uses whatever mapping the implementor defines
for the environment and store. '^M' represents a carriage-return.  */
```

Figure 5-6: Trace Statement Syntax and Semantics

Tracing is another debugging facility that can be described by extending the target programming language with a special statement. As illustrated in figure 5-6, a variable might be traced by inserting a trace statement with the variable name at every point where display of the variable's value is desired. The trace statement is executed similarly to the write statement: the variable's current value is appended to the output stream of a designated I/O channel and displayed using the standard unparse mechanism. Alternatively, the explicit trace statement can be avoided by attaching a conditional equation to the Execute event for the assignment statement, to perform the trace when any variable in some list (given by an input channel) is assigned.

6. Implementation Algorithms

The implementation of action equations consists of two parts, translation and run-time support. Both parts involve an adaptation of the Reps, Teitelbaum and Demers algorithms [52] for generation of LBEs from attribute grammars. Reps' algorithms work roughly as follows. The translator takes as input the environment description and produces as output (1) various tables reflecting the syntax description; (2) a *local dependency graph* for each production representing the dependencies among the attributes that appear in its semantic equations; and (3) a procedure for each semantic equation, which carries out the actual evaluation of the equation.

After each subtree replacement, Reps' run-time support constructs a *scheduling graph* by grafting together two projections of the local dependency graph for the root of the replacement subtree, one denoting the transitive dependencies among the attributes of the node *via* its parent and siblings, and the other the transitive dependencies among the attributes of the node *via* the subtree. The attributes represented in the scheduling graph are reevaluated in the order given by a topological sort of the graph. The attributes represented by *independent* vertices (*i.e.*, those vertices with no incoming edges) are reevaluated first.

If the execution of a semantic equation results in a value different from the previous value of the attribute, then the scheduling graph is *expanded* to include the projected local dependency graphs for all attributes that depend directly on the changed attribute. The expansion involves adding edges representing transitive dependencies for all of these attributes that were not previously part of the scheduling graph. Whether or not the attribute changed in value, it and all its outgoing edges are now removed from the graph and evaluation continues with those attributes now represented by independent vertices. This process continues until the scheduling graph becomes empty, which is guaranteed to happen eventually if the attribute grammar is non-circular. (Algorithms to detect circularity in an attribute grammar are exponential [30], so whether or not a given attribute grammar is non-circular is often determined by inspection.) This evaluation algorithm is asymptotically optimal in the sense that the number of attribute evaluations is proportional to the number of attributes that are necessarily reevaluated. (The efficiency may be improved by maintaining additional data structures, making it possible to avoid all unnecessary evaluations.)

The adapted set of algorithms operate as follows. During translation of action equations, syntax tables and procedures are generated similarly to attribute grammars. The important distinction is that a local dependency graph is constructed for each event, whether synthesized or inherited, associated with each production. The graph represents the dependencies among the equations attached to the event rather than the attributes that appear in these equations; this is necessary because the outputs of action equations may be placed in locations within an attribute or within the syntax tree and these locations may be computed during action equation evaluation. In each graph, there are no incoming edges for each delay equation (to ensure that they are evaluated first), an outgoing edge from every delay equation to every other kind of equation, and an incoming edge from every other kind of equation to every propagate equation. There is also a local dependency graph for the set of equations — constraints and conditionals — not attached to any event.

After each subtree replacement, a scheduling graph is constructed from the projected local dependency graphs for the equations not attached to any event and also the two local dependency graphs (synthesized and inherited) for the equations attached to the standard event corresponding to the user command that caused the subtree replacement. In response to each user command corresponding to a cursor movement or an implementor-defined event, a scheduling graph is constructed from the two local dependency graphs for the equations attached to that event. In either case, the run-time support then follows the topological sort/graph expansion process described above.

As explained previously, the evaluation of several delay equations is treated as simultaneous and results in saving the scheduling graph together with a representation of the required event/receiver pairs. Once the full set of events has been selected, in any order and spread out over any period of time, the saved scheduling graph is grafted together with the then current scheduling graph. The evaluation of several propagate equations is also treated as simultaneous, and results in a new scheduling graph (by definition, the previous graph is empty except for the propagate equations), which includes the two local dependency graphs for all the propagated event/destination pairs.

The incremental action equation evaluation algorithm is asymptotically optimal in the same sense as the base incremental attribute grammar evaluation algorithm. In the case of a subtree replacement, constraints are treated as if they were semantic equations and evaluated *via* the identical mechanism. In the case of equations attached to an event, each equation is evaluated exactly once for each selection of the event as required by the semantics of action equations, and the number of constraint evaluations is proportional to the number of constraints that must be reevaluated.

7. Conclusions

The purpose of this article is to demonstrate that attribute grammars can be easily extended to specifying dynamic semantics in addition to static semantics. The action equations paradigm does this by making the attribute grammar itself dynamic, where some semantic equations are active and others are passive. Equations are changed from passive to active to passive again according to external user commands and internal computations involving the propagate and delay equations. Action equations also augment attribute grammars with limited side-effects, which make it possible to maintain the state of program execution and the history of user interactions with the environment.

This extension of attribute grammars was developed to permit generation of LBEs that support both static and dynamic semantics. Attribute grammars previously permitted generation of environments that support only static semantics. Action equations can also be applied outside LBEs to generate interpreters and debuggers, just as attribute grammars have been used to generate compilers.

Acknowledgements

The bulk of this research was conducted as part of my PhD thesis [36] at Carnegie Mellon University. I would like to thank my advisor, Nico Habermann, for his advice and support throughout my graduate student career. I would also like to thank the other members of my committee, Scott Fahlman, Elaine Kant and Bill Riddle, for their contributions. Discussions with David Garlan helped me work out the details of the action equations paradigm. I have more recently collaborated with David to develop Meld [37], an experimental language combining object-oriented and dataflow programming paradigms; Meld's syntax and semantics are loosely based on action equations and on the views David proposed in his thesis [22]. Simon Kaplan pointed out an error in one of the examples given in my dissertation, which is corrected here; Simon has also worked with me on parallel and distributed incremental evaluation algorithms for attribute grammars [38]. We have recently applied this work to a concurrent extension of action equations [40]. Finally, I would like to thank Dave Ackley, Nico Habermann, Josephine Micallef and the anonymous referees for their critical comments on earlier versions of this article; the comments of one of the referees were particularly useful and led to vast improvements in the form and content of the article.

A parallel/distributed implementation of Meld [41] has been completed by Nicholas Christopher, Seth Strumph and Shyhtsun (Felix) Wu under the direction of Wenwey Hseush and later Steve Popovich. Gaea, a rapid prototyping system for interpreter/debuggers based on the action equations paradigm, is currently being implemented by Travis Winfrey with the participation of Kok-Yung Tan, Matsuki Yoshino and Semyon Dukach. Both Meld and Gaea are being developed at Columbia University. Gail Kaiser is now supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens and Sun, by the Center of Advanced Technology and by the Center for Telecommunications Research.

References

u

1. Ambriola, Vincenzo, Kaiser, Gail E. and Ellison, Robert J. An action routine model for ALOE. Tech. Rept. CMU-CS-84-156, Carnegie Mellon University, Department of Computer Science, August, 1984.

2. Archer, James E. Jr. and Devlin, Michael T. Rational's experience using Ada for very large systems. First International Conference on Ada Programming Language Applications for the NASA Space Station, June, 1986, pp. B.2.5.1-B.2.5.11.

3. Bahlke, Rolf and Snelting, Gregor. "The PSG system: from formal language definitions to interactive programming environments". *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 547-576.

4. Balzer, Robert. "A 15 year perspective on automatic programming". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1257-1268.

5. Barbuti, R., Bellia, M., Degano, P., Levi, G., Dameri, E., Simonelli, C. and Martelli, A. Programming environment generation based on denotational semantics. In *Theory and Practice of Software Technology*, North-Holland Pub. Co., New York, 1983.

6. Bodwin, James, Bradley, Laurette, Kanda, Kohji, Litle, Diane and Pleban, Uwe. Experience with an experimental compiler generator based on denotational semantics. SIGPlan '82 Symposium on Compiler Construction, June, 1982, pp. 216-229.

7. Delisle, Norman M., Menicosy, David E. and Schwartz, Mayer D. Viewing a programming environment as a single tool. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, April, 1984, pp. 49-56.

8. Demers, Alan, Reps, Thomas and Teitelbaum, Tim. Incremental evaluation for attribute grammars with applications to syntax-directed editors. 8th Annual ACM Symposium on Principles of Programming Languages, January, 1981, pp. 105-116.

9. Demers, Alan, Rogers, Anne and Zadeck, Frank Kenneth. Attribute propagation by message passing. SIGPlan '85 Symposium on Language Issues in Programming Environments, June, 1985, pp. 48-59.

10. Despeyroux, Thierry. Executable specification of static semantics. Semantics of Data Types International Symposium, New York, June, 1984, pp. 215-233.

11. Donzeau-Gouge, Veronique, Huet, Gerard, Kahn, Gilles, and Lang, Bernard. Programming environments based on structured editors: the Mentor experience. In Barstow, David R., Shrobe, Howard E. and Sandewall, Erik, Ed., *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984, pp. 128-140.

12. Donzeau-Gouge, Veronique, Kahn, Gilles, Lang, Bernard and Melese, B. Documents structure and modularity in Mentor. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, April, 1984, pp. 141-148.

13. Engels, G., Gall, R., Nagl, M. and Schafer, W. "Software specification using graph grammars". *Computing 31* (1983), 317-346.

14. Farrow, Rodney. "Generating a production compiler from an attribute grammar". *IEEE Software 1*, 4 (October 1984).

15. Farrow, Rodney. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. SIGPlan '86 Symposium on Compiler Construction, June, 1986, pp. 85-98.

16. Feiler, Peter H. and Medina-Mora, Raul. "An incremental programming environment". *IEEE Transactions on Software Engineering SE-7*, 5 (September 1981), 472-482.

17. Feiler, Peter H. *LOIPE a language-oriented interactive programming environment based on compilation technology*. Ph.D. Th., Carnegie Mellon University, May 1982. CMU-CS-82-117..

18. Feiler, Peter H., Jalili, Fahimeh and Schlichter, Johann H. An interactive prototyping environment for language design. 19th Hawaii International Conference on System Sciences, January, 1986, pp. 106-116.

19. Freeman, Peter. "A conceptual analysis of the Draco approach to constructing software systems". *IEEE Transactions on Software Engineering SE-13*, 7 (July 1987), 830-844.

20. Ganzinger, Harald, Ripken, Knut and Wilhelm, Reinhard. Automatic generation of optimizing multipass compilers. Information Processing 77, New York, 1977, pp. 535-540.

21. Garlan, David. Flexible unparsing in a structure editing environment. Tech. Rept. CMU-CS-85-129, Carnegie Mellon University, Department of Computer Science, April, 1985.

22. Garlan, David. *Views for tools in integrated environments*. Ph.D. Th., Carnegie Mellon University, May 1987. CMU-CS-87-147..

23. Ghezzi, Carlo and Mandrioli, Dino. "Augmenting parsers to support incrementality". *Journal of the ACM 27*, 3 (July 1980), 564-579.

24. Habermann, A. N. and Notkin, D. "Gandalf: software development environments". *IEEE Transactions on Software Engineering SE-12*, 12 (December 1986), 1117-1127.

25. Henderson, Peter and Weiser, Mark. Continuous execution: the VisiProg environment. 8th International Conference on Software Engineering, August, 1985, pp. 68-74.

26. Hoover, Roger and Teitelbaum, Tim. Efficient incremental evaluation of aggregate values in attribute grammars. SIGPlan '86 Symposium on Compiler Construction, June, 1986, pp. 39-50.

27. Hoover, Roger. Dynamically bypassing copy rule chains in attribute grammars. 13th Annual ACM Symposium on Principles of Programming Languages, January, 1986, pp. 14-25.

28. Horwitz, Susan and Teitelbaum, Tim. "Generating editing environments based on relations and attributes". *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 577-608.

29. Hudson, Scott E. and King, Roger. Implementing a user interface as a system of attributes. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, December, 1986, pp. 143-149.

30. Jazayeri, M., Ogden, W. F. and Rounds, W. C. "The intrinsically exponential complexity of the circularity problem for attribute grammars". *Communications of the ACM 18*, 12 (December 1975).

31. Johnson, S. C. and Lesk, M. E. "Language development tools". *The Bell System Technical Journal 57*, 6 (July-August 1978), 2155-2175.

32. Johnson, Gregory F. and Fischer, Charles N. Non-syntactic attribute flow in language based editors. 9th Annual ACM Symposium on Principles of Programming Languages, January, 1982, pp. 185-195.

33. Johnson, Gregory F. and Fischer, C. N. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. 12th Annual ACM Symposium on Principles of Programming Languages, January, 1985, pp. 141-151.

34. Johnson, Gregory F. GL — a denotational testbed with continuations and partial continuations as first-class objects. SIGPlan '87 Symposium on Interpreters and Interpretive Techniques, June, 1987, pp. 165-176.

35. Gail E. Kaiser and Elaine Kant. "Incremental Parsing Without A Parser". *The Journal of Systems and Software 5*, 2 (May 1985), 121-144.

36. Kaiser, Gail E. *Semantics of structure editing environments*. Ph.D. Th., Carnegie Mellon University, May 1985. CMU-CS-85-131..

37. Kaiser, Gail E. and Garlan, David. "Melding software systems from reusable building blocks". *IEEE Software* (July 1987), 17-24.

38. Kaiser, Gail E., Kaplan, Simon M. and Micallef, Josephine. "Multiuser, distributed language-based environments". *IEEE Software* (November 1987), 58-67.

39. Gail E. Kaiser, Peter H. Feiler, Fahimeh Jalili and Johann H. Schlichter. "A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation". *Software — Practice & Experience 18*, 8 (August 1988), 733-748.

40. Kaiser, Gail E. and Kaplan, Simon M. Rapid prototyping of concurrent programming languages. 8th International Conference on Distributed Computing Systems, June, 1988, pp. 250-255.

41. Kaiser, Gail E. Concurrent Meld. Workshop on Object-Based Concurrent Programming, :ptember, 1988. To appear.

42. Kastens, U., Hutt, B. and Zimmermann, E.. *Lecture Notes in Computer Science*. Volume 141:*GAG: A Practical Compiler Generator*. Springer-Verlag, Heidelberg, 1982.

43. Knuth, Donald E. "Semantics of context-free languages". *Mathematical Systems Theory 2*, 2 (June 1968), 127-145.

44. Lamb, David Alex . "IDL: sharing intermediate representations". *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 297-318.

45. Medina-Mora, Raul. *Syntax-directed editing: towards integrated programming environments*. Ph.D. Th., Carnegie Mellon University, March 1982. CMU-CS-82-113..

46. Notkin, David S. *Interactive structure-oriented computing*. Ph.D. Th., Carnegie Mellon University, February 1984. CMU-CS-84-103..

47. Partsch, H. and Steinbruggen, R. "Program transformation systems". *Computing Surveys 15*, 3 (September 1983), 199-236.

48. Paulson, Lawrence. A semantics-directed compiler generator. 9th Annual ACM Symposium on Principles of Programming Languages, January, 1982, pp. 224-233.

49. Raskovsky, Martin R. Denotational semantics as a specification of code generators. SIGPlan '82 Symposium on Compiler Construction, June, 1982, pp. 230-244.

50. Reiss, Steven P. Graphical program development with PECAN program development systems. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, April, 1984, pp. 30-41.

51. Reiss, Steven P. An approach to incremental compilation. SIGPlan '84 Symposium on Compiler Construction, June, 1984, pp. 144-156.

52. Reps, Thomas, Teitelbaum, Tim and Demers, Alan. "Incremental context-dependent analysis for language-based editors". *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 449-477.

53. Reps, Thomas and Teitelbaum, Tim. The Synthesizer Generator. SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments, April, 1984, pp. 41-48.

54. Reps, Thomas, Marceau, Carla and Teitelbaum, Tim. Remote attribute updating for language-based editors. 13th Annual ACM Symposium on Principles of Programming Languages, January, 1986, pp. 1-13.

55. Scott, Dana and Strachey, Christopher. Toward a mathematical semantics for computer languages. Tech. Rept. Technical Monograph PRG-6, Oxford University Computing Laboratory, August, 1971.

56. Shebs, Stan and Kessler, Robert. Automatic design and implementation of language datatypes. SIGPlan '87 Symposium on Interpreters and Interpretive Techniques, June, 1987, pp. 26-37.

57. Smith, Douglas R., Kotik, Gordon B. and Westfold, Stephen J. "Research on knowledge-based software environments at Kestrel Institute". *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1278-1295.

58. Snodgrass, Richard and Shannon, Karen. *Lecture Notes in Computer Science*. Volume 244: Supporting flexible and efficient tool integration. In *Advanced Programming Environments*, Conradi, Reidar, Didriksen, Tor M. and Wanvik, Dag H., Eds., Springer-Verlag, Berlin, 1986, pp. 290-313.

59. Teitelbaum, Tim and Reps, Thomas. "The Cornell Program Synthesizer: a syntax-directed programming environment". *Communications of the ACM 24*, 9 (September 1981), 563-573.

60. Teitelman, Warren and Masinter, Larry. "The Interlisp programming environment". *IEEE Computer 14*, 4 (April 1981), 25-34.

61. Walz, Janet A. and Johnson, Gregory F. Incremental evaluation for a general class of circular attribute grammars. SIGPlan '88 Conference on Programming Language Design and Implementation, June, 1988, pp. 209-221.

62. Waters, Richard C. "KBEmacs: where's the AI?". *The AI Magazine VII*, 1 (Spring 1986), 47-56.

63. Wegman, Mark N. Parsing for structural editors. 21st Annual Symposium on Foundations of Computer Science, October, 1980, pp. 320-327.

Rapid Prototyping of Concurrent Programming Languages

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Simon M. Kaplan
University of Illinois
Department of Computer Science
Urbana, IL 61801

ABSTRACT

We propose a new technology for rapid prototyping of concurrent programming languages. The designer of a new language specifies its syntax and semantics in a formal notation. Our system generates a parallel interpreter for the language and provides run-time support for the synchronization primitives and other facilities in the language.

Introduction

The allure of distributed computing systems has led to the development of many concurrent programming languages. One problem is that design progresses much more quickly than implementation, and there has been little opportunity to experiment with many of these new languages. The search for the right communication and synchronization primitives would be aided by mechanisms for rapid prototyping of concurrent programming languages.

We propose a new technology for automatic generation of concurrent interpreters from formal specifications of the programming languages. Our technology consists of formal notation and supporting algorithms. The formal notation is called *action equations*, which are attribute grammars extended by the concepts of *events* and *unification*. The supporting algorithms include (1) preprocessing algorithms to generate the interpreters and (2) evaluation algorithms embedded in the generated interpreters.

This paper overviews attribute grammars, explains the extension to the action equations paradigm, and presents the synthesis of action equations with a unification strategy as our means for specifying and implementing synchronization primitives. We illustrate our approach by giving a specification of CSP. We discuss our supporting algorithms and then describe the concurrent interpretation of an example CSP program. The paper concludes with a brief comparison to related work.

Attribute Grammars

Action equations are a strict superset of *attribute grammars*, which were introduced by Knuth [16] for specifying the context-sensitive properties of programming languages. An attribute grammar augments each production in a context-free grammar with *semantic equations*, which define the context-sensitive rules associated with the production. Each equation defines the value of an attribute as a function of terminal grammar symbols and other attributes. These other attributes are defined in turn by equations that augment the same or a different production. Synthesized attributes are those associated with the nonterminal grammar symbol on the left hand side of the production and inherited attributes are those for the terminal and nonterminal symbols on the right hand side. A program is represented as a parse tree where each node is decorated by the corresponding attributes.

```
GC ::= guard: EXPRESSION
       body: STATEMENT
       error: string
       code: text

    If guard.type = "boolean"
    Then error := "<-- type error"
    Else error := ""

    code := "if (" guard.code ") " body.code

= ::= operand1: EXPRESSION
      operand2: EXPRESSION
      type: TYPE
      code: text

    If operand1.type = operand2.type
    Then type := "boolean"
    Else type := "undefined"

    code := "(" operand1.code ") == ("
             operand2.code ")"
```

Figure 1: Portion of Attribute Grammar

The first production in figure 1 shows the CSP guarded command [7], used for the do and if statements. Our attribute grammar notation follows the Interface Description Language (IDL) [26] convention of naming the components of productions (guard and body) and listing together with the components the names and types of the synthesized and inherited attributes of the grammar symbol on the left hand side (error and code are synthesized attributes). The first equation for the GC production defines the value of the error attribute of the GC symbol as a function of the type attribute of the guard symbol. The type attribute is defined separately for each EXPRESSION production. For example, the = production shown defines its type attribute as a function of the type attributes of the two operand symbols. GC's second equation defines its code attribute as a function of the code attributes of the guard and body symbols.

Attribute grammars have long been used for rapid prototyping of compilers for sequential languages [6, 5]. The compiler-compiler takes as input an attribute grammar for the desired programming language and produces a compiler for the language. The translator component of the compiler-compiler typically produces language-specific tables from the attribute grammar, which are used by a language-independent attribute evaluation algorithm [15, 2] included as part of the generated compiler. Once the parse tree is constructed, the attribute evaluator decorates its nodes with a consistent set of attribute values. This is generally possible only if there are no (nonconverging) circularities among the equations — *e.g.*, "a := f(b)" and "b := g(a)". Attribute evaluation has the effect of detecting any static semantic (context-sensitive) errors as well as producing object code. After evaluation terminates, the compiler might report errors by traversing the parse tree in prefix order, printing non-null error attributes (with surrounding context) as it finds them. If no error messages are found, then it might write the code attribute at the root of the program to a separate file.

Attribute grammars are equally applicable to compilation of sequential and concurrent languages, but unfortunately equally inapplicable to interpretation of either kind of language. The problem is that attributes are by definition derived solely from the program text, given the set of semantic equations. The values of attributes, once computed, remain the same; attribute values represent *static* properties of programs. Interpretation requires maintenance of *dynamic* properties, such as the run-time stack and the contents of memory. Attribute grammars are not suitable for expressing such properties.

However, it is exactly the dynamic properties of concurrent programming languages that are interesting. Concurrent languages are naturally more complex than sequential languages because they combine all the problems of sequential programming with the additional problems of synchronization. Concurrent interpreters are useful for testing and debugging sequential behavior within a process, but are more important for following the flow of communication among processes. As more and more new language features are proposed, rapid prototyping becomes more and more desirable. It is necessary to develop a formal notation for specifying semantics of concurrent languages that is sufficiently expressive to support automatic generation of concurrent interpreters. We follow an operational approach in order to produce relatively efficient interpreters.

Action Equations

We have previously presented *action equations* as an extension of attribute grammars that supports rapid prototyping of interpreters for sequential programming languages [11]. In this paper we sketch this support, and then extend action equations to concurrency.

Action equations as previously defined are simply attribute grammars augmented with the notion of *events*. An event corresponds to an externally initiated activity, such as invoking an interpreter. Certain equations are *attached to* particular events, meaning these equations define the dynamic semantics of the event for the particular production. Equations are attached to events in two forms: "*<event>* --> *<equation(s)>*" and "*<event>* On *<component>* --> *<equation(s)>*". The first is analogous to the notion of a synthesized attribute, associating the event and its equations with the grammar symbol on the left hand side of the production; the right follows the notion of an inherited attribute, associating the event and its equations with a grammar symbol on the right hand side. Action equations introduce a new kind of equation with the form "Propagate *<event>* To *<destination>*", where *<destination>* is a grammar symbol. This permits the semantics of an event for one symbol to be defined in terms of (1) the same event for a different symbol, (2) a different event for a different symbol, and/or (3) a different event for the same symbol.

```
IF ::= body: sequence of GC

     RUN -->
          Propagate RUN To body[1]

     CONTINUE On body[any] -->
          Propagate RUN To body[next]

     CONTINUE On body[last] ->
          Propagate CONTINUE To self

GC ::= guard: EXPRESSION
       body: STATEMENT

     RUN -->
          Propagate RUN To guard

     CONTINUE On guard -->
          If guard.value Then Propagate RUN To body
          Else Propagate CONTINUE To self

     CONTINUE On body -->
          Propagate CONTINUE To self
```

Figure 2: Portion of Action Equations

The equations in figure 1 define static properties — static semantic analysis and code generation — and are thus not attached to any event. The equations in figure 2 define the dynamic semantics of interpretation, so they are attached to events representing interpretation. These equations specify the interpretation of the CSP if statement and its guarded command list, where the RUN event corresponds to the invocation of the interpreter on the particular language construct (essentially, a high-level program counter) and the CONTINUE event corresponds to the continuation introduced by denotational semantics [27].[1] self always refers to the symbol on the left hand side of the production and value is an attribute of each EXPRESSION production.

The first event for the IF production defines the RUN event for the IF symbol in terms of the RUN event for the first element of the body symbol. The second defines the CONTINUE event for any element of the body in terms of the RUN event for the next element, if any. The third defines the continuation of the last element of the body as the same as the continuation of the IF symbol. The events for the GC production are similar. Here the CONTINUE event on the guard is defined in terms of its value attribute as well as in terms of other events.

Action equations as explained above can be used for rapid prototyping of interpreters for sequential languages. The interpreter generator takes as input the action equations for the desired programming language and produces an interpreter for the language. The translator component of the generator produces language-specific tables for a language-independent evaluation algorithm. Since action equations include attribute grammars as a proper subset, the evaluator decorates the nodes of the parse tree with a consistent set of attribute values. Static semantics errors may be detected and reported as previously described.

Interpretation is initiated by a user activity, such as selecting a node in the parse tree and giving a command corresponding to an event associated with the production that defines the node. This has the effect of *activating* the equations attached to that event. Each of these equations is evaluated exactly once, in the order implied by their input/output dependencies. Any propagate equations among

[1]Note that since essentially every production sends a CONTINUE event to itself, this equation could be added automatically by the translator for all except special cases, which would then be indicated in the specifications.

the activated equations have the effect of activating additional equations attached to the named events for the indicated symbols. This process may or may not terminate, depending on patterns of circularities among the equations — "X --> Propagate Y To self" and "Y --> Propagate X To self" is a pathological case. In contrast to attribute grammars, circularities are sometimes necessary, for example, to model "while true do ...".

Action equations for the interpretation of sequencing, looping, branching, potentially recursive subroutine call and subroutine return are given elsewhere [10]. There we demonstrate that action equations are a simple means (none of the specifications is longer than a page!) for specifying the dynamic semantics of sequential programming languages.

Synchronization via Unification

Naively, it might appear that a simple extension of events would be sufficient to specify the synchronization primitives required for concurrent languages. For example, it might seem that we could add arguments to events, and then specify a send-and-continue statement with "Propagate SEND(message) To receive_statement" and the corresponding receive statement with "SEND(message) --> equations". Unfortunately, this does not work.

Consider a concurrent program with processes *P* and *Q*, where *P* contains several send statements and *Q* contains several receive statements. Since *P* and *Q* execute in parallel, it is not possible for process *P* to know *a priori* which of the several possible receive points should be the destination of a particular send. During different executions of the same program, the same send might be matched with different receives and the same receive with different sends. Therefore, it is impossible to fill in the "receivestatement" portion of the proposed propagate equation; similarly, it would be impossible to determine in advance all necessary information for any similar extension of events. The matching between send statement and receive statement can only be resolved when both communicating processes are ready.

This problem is very similar to the *unification* [19] mechanism in Prolog [3]. For this reason we turn to a unification-based mechanism for specifying a language's synchronization primitives.

We associate with each parse tree a database of tuples, which is shared among all processes. We define two new kinds of equations that operate on the database: *assert* and *block*. Both operations take as arguments a tuple of expressions, where an expression may be a grammar symbol, an attribute, or a function of these. Each grammar symbol or attribute that appears as an element of the tuple may be suffixed with an exclamation point ("!") that marks it as *read-only* in the sense of the read-only variables of Concurrent Prolog [24].[2] Marking variables read-only inhibits the direction of unification, since these values cannot be changed as a result of unification. Unlike Prolog and Concurrent Prolog, variables need not begin with an uppercase letter: everything is a variable unless marked read-only.

[2] Concurrent Prolog actually uses the question mark ("?") for this purpose, but we find this confusing for the reader of the specification.

The semantics of these equations are as follows.

Assert	The assert equation attempts to unify its argument tuple with an entry currently in the database using Concurrent Prolog's style of unification. If unification succeeds, certain components and attributes of the participating parse tree nodes are instantiated to the results of the unification and the other tuple that participated in the unification is automatically retracted (*i.e.*, removed from the database). If the unification fails, the tuple is inserted into the database and execution continues normally as defined by the action equations.
Block	The block equation is exactly the same as assert, except that if unification fails, execution waits until another tuple arrives with which the argument tuple can unify.

```
(* Send and Wait *)

SEND ::= receiver: identuse
         message: EXPRESSION

    RUN -->
        Block(message!, self!, receiver!)
        Propagate CONTINUE To self

(* Send and Continue *)

SEND ::= receiver: identuse
         message: EXPRESSION

    RUN -->
        Assert(message!, self!, receiver!)
        Propagate CONTINUE To self

(* Receive *)

RECEIVE ::= sender: identuse
            variable: identuse

    RUN -->
        Block(variable.lvalue, sender!, self!)
        Propagate CONTINUE To self

(* Anonymous Receive *)

RECEIVE ::= sender: identuse
            variable: identuse

    RUN -->
        Block(variable.lvalue, sender, self!)
        Propagate CONTINUE To self
```

Figure 3: Action Equations for Synchronization Primitives

Augmenting action equations with the assert and block equations is sufficient to implement all known (to us) synchronization primitives. These include send-and-wait, send-and-continue, receive and anonymous receive. Figure 3 gives the action equations that define the dynamic semantics of these four primitives; we omit the equations to allocate variables and evaluate expressions.

The RUN event for the send-and-wait statement is defined as a block on the tuple consisting of its message, some identification of itself (*identuse* represents an identifier use site, *identdef* an identifer definition site), and the name of the desired receiver. When the tuple unifies, both tuples are retracted and the sending process continues. The RUN event for the send-and-continue statement is defined identically, except that assert is substituted for block. The RUN event is defined as an assert of the tuple consisting of its message, identification and receiver name, and immediately continues. The RUN event for the receive statement is defined as a block on the tuple consisting of the location (lvalue) of a variable, the name of the desired sender, and some identification. When the tuple unifies, the effect is to transmit the message from the sender to the variable location of the receiver. The RUN event for the anonymous receive is defined as a block on the tuple consisting of the location of a variable, the name of any sender, and its identification.

```
CALL ::= receiver: identuse
         argument: EXPRESSION
         return: identuse

    RUN -->
        Assert(argument!, self!, receiver!)
        Propagate WAIT To self

    WAIT -->
        Block(return.lvalue, self!, receiver!)
        Propagate CONTINUE To self

PROCEDURE ::= sender: identuse
              argument: identdef
              body: STATEMENT
              result: identdef

    RUN -->
        Block(argument.lvalue, sender, self!)
        Propagate RUN To body

    CONTINUE On body -->
        Assert(result.rvalue!, sender!, self!)
        Propagate CONTINUE To self
```

Figure 4: Action Equations for Remote Procedure Call

Remote procedure call, broadcast to a known set of receivers, and receipt from a known set of senders can be implemented using extensions of these primitives. For example, figure 4 defines a simplified remote procedure call, where the local stub that deals with a name server and the network and the remote stub that issues the RUN event to the selected remote procedure are subsumed into our run-time support (the other omitted details are the same as for local subroutine calls [11]). The RUN event for the CALL production initiates the remote call with the assert equation, and then issues a WAIT event to activate the equations that wait for the return from the remote procedure. Semaphores and monitors can be implemented using these techniques and the 'encapsulated resource' approach as described in [8]. Anonymous broadcast (broadcast to an unknown number of receivers) requires *persistent* assertions (no automatic retraction on unification) and the addition of timestamps to each tuple so that receivers can determine which message they should read. Virtual clocks [17] are sufficient for the timestamps, because comparison is always among messages from the same sender.

```
(* Propagate Equation *)

PROPAGATE ::= event: EVENT
              destination: NODE

    Assert(event!, destination!)

(* Attaching Equations to an Event *)

ATTACH ::= event: EVENT
           equations: sequence of EQUATION

    PersistentBlock(event!, self!) --> equations
```

Figure 5: Special Cases of Unification

As an aside, note that the propagate equation and attaching equations to events — the original extensions from attribute grammars to action equations — can both be treated as special cases of unification with persistent tuples, where the event name becomes an element of the tuple. See figure 5.

CSP Specification

```
PROGRAM ::= body: sequence of PROCESS

    RUN -->
        Propagate RUN To body[all]

PROCESS ::= name: identdef
            locals: sequence of identdef
            body: sequence of STATEMENT

    RUN -->
        Propagate RUN To body[1]

    CONTINUE On body[any] -->
        Propagate RUN To body[next]
```

Figure 6: Portion of Action Equations for CSP

CSP's send-and-wait and receive statements were defined in the previous section. Figure 6 shows the top-level program/process specifications.

Supporting Algorithms

The implementation of sequential action equations involves an adaptation of Reps' algorithm for incremental attribute grammar evaluation [21, 22]. This algorithm restores consistency among attribute values after a subtree replacement in the parse tree representing the program. It re-evaluates only those attributes whose values may have been affected by the subtree replacement, retaining the old values of all other attributes. This is achieved using a scheduling graph, called the *model*, that represents the direct and transitive dependencies among the attributes of the parse tree. The equations that define the attributes are re-evaluated in an order consistent with a topological sort of the model, starting with an attribute at the point of the subtree replacement and avoiding re-evaluating those attributes that could not have changed in value because none of the attributes on the right hand side of its defining equation have changed. Reps applied this algorithm to static semantic analysis within language-based editors [23].

Our adaptation does not assume language-based editing, or any form of editing; it is instead a basis for interpretation. During preprocessing of action equations, a *local dependency graph* is constructed for each event associated with each production. The graph represents the dependencies among the equations attached to the event. In particular, there is a vertex for each attribute; an equation is reflected in the graph by an arc leaving each attribute on the right hand side of the equation and entering the attribute on the left hand side.

When an event arrives at run-time — *i.e.*, during interpretation — a model is constructed to represent the transitive dependencies among the equations attached to the event. Initially, the model is a copy of the local dependency graph for the event with respect to the production that defines the current node in the parse tree. For sequential execution, the equations are evaluated in an order consistent with a topological sort of the model. If a new event is propagated, the model is *expanded* — by adding the corresponding local dependency graph — to reflect the equations attached to the event. This process repeats until the model becomes empty.

To support concurrent evaluation of action equations, we adapt our parallel/distributed algorithm for incremental evaluation of attribute grammars [14, 12] in the same manner that we adapted Reps' algorithm for the sequential case. Concurrency within a process is supported as follows. Each time a vertex in the model becomes independent (*i.e.*, has no incoming arcs) during the topological sort, a separate process is spawned to first evaluate the corresponding equation and then expand the model if necessary. When there are multiple independent vertices, the corresponding equations can be evaluated concurrently by separate processes. Modifications to the model are treated as critical sections.

This means that interpretation can even proceed concurrently *for a sequential language*, with the crucial exception that the evaluation algorithm forces interdependent language constructs to be executed in the correct dataflow order. In particular, if one language statement depends on a side-effect of a previous language statement,[3] then this will be reflected in the dependencies among the corresponding action equations, so the topological sort will result in interpreting the statements in the correct order. Otherwise, we achieve maximum parallelism provided there are sufficient processors to simultaneously execute the concurrent processes.

Concurrency among processes, whether on the same or different machines, is supported by associating a separate model with each 'distributable unit', such as the PROCESS production in figure 6. The synchronization among processes is handled by unification, which operates in the obvious way except for our automatic retraction and how we treat multiple unification. We assume an implicit 'cut' following every assertion, so only the first found unification is accepted (and retracted). Unfortunately, the database used for unification is currently a centralized resource. We are working on extending the high availability/reliability algorithm [13] we developed for distributed attribute evaluation to permit decentralization by replication.

CSP Example

```
    P:: [ integer i;
          i := 5 ;
          Q!i;
        ]

 || Q:: [ integer j ;
          P?j;
          j := j+1 ;
        ]
```

Figure 7: CSP Example

We illustrate how the generated interpreter works for CSP with the simple example program in figure 7. Execution of the two processes works as follows: When the PROGRAM node (as defined in figure 6) receives the RUN event from the user, the activated action equation propagates the RUN event to the two PROCESS nodes, *P* and *Q*. This in turn creates two disjoint models, one for each process. The interpreter then selects all independent equations in either model and evaluates them. In this case, the only equations are the two propagate equations, which are simultaneously executed. They propagate RUN to the first statement in each process, initiating the body of the process.

Interpreting *P* involves propagating a RUN event to the assignment "i := 5". This sets *i* to "5" and selects CONTINUE on itself, indicating to the parent PROCESS node that it has completed its execution. This propagates RUN to the next statement, the CSP send statement. The RUN event for the SEND node (figure 3 — Send-and-Wait) instructs the interpreter to delay until a unifiable tuple comes along (if *Q* has executed more quickly, this may already be there). The message (*i*), self (*P*) and receiver (*Q*) are all read-only, *i.e.*, they cannot change as a result of the unification. If *Q* is running ahead of *P*, and has reached its receive statement, the unification succeeds, the tuples are retracted, and CONTINUE is propagated to the PROCESS node once again. As there are no more statements in the body, the model for *P* becomes empty and thus execution of *P* halts.

Process *Q* executes simultaneously with *P* (figure 6). When the RUN event is selected, it propagates RUN to the RECEIVE node, which blocks until *P* places the matching tuple in the database (figure 3). The self and send parameters in the tuple are read-only, so unification can succeed in this case only on an exact match. The third parameter, variable_lvalue, is not read-only and gets matched with the message of the sender, thereby transmitting the value of the variable *i* in process *P*. This is the desired behavior for CSP; in other languages we can be more flexible where needed. Once unification is complete, the RECEIVE node propagates CONTINUE to itself, which propagates RUN to the assignment statement. When the assignment is complete, a CONTINUE is propagated once more, terminating the interpretation of *Q* since the model becomes empty.

Interpretation of *P* and *Q* can proceed in any real-time order permitted by the action equations evaluation algorithm. Thus, *P* can execute faster than *Q*, or slower. Either way, the processes synchronize on the send and receive statements, as one cannot con-

This example is of course specific to CSP, but our notation and algorithms can be used for rapid prototyping of arbitrary concurrent languages. Experimentation is facilitated by specifying alternative constructs as action equations and using our generation and supporting algorithms to quickly try them out.

Conclusions

The primary contribution of this research is the generation of parallel interpreters — for both sequential and concurrent languages — from formal language specifications. We use a unification-based approach to the specification of synchronization primitives.

Several other concurrent debugging systems have been developed [18, 1, 4, 20, 25]. They share our goal of allowing the user to focus on the interaction between processes. All of these systems are language-specific, although [9] describes a general, data-oriented style of debugging applicable to a range of concurrent object-oriented languages. Our work is unique in that it allows the *generation* of interpreters for concurrent languages. Our low overhead implementation mechanism permits experimentation with new concurrency primitives to keep pace with design.

Acknowledgements

Travis Winfrey is working with us on an implementation of action equations called GAEA. We'd like to thank Yael Cycowicz, Dan Yellin and the referees for their useful comments on earlier versions of this paper.

Kaiser is supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, in part by the Columbia University Center for Telecommunications Research, and in part by a Digital Equipment Corporation Faculty Award. Kaplan is supported in part by a grant from the AT&T Corporation.

[3]We determine such dependency either optimistically assuming no aliasing or pessimistically assuming maximum possible aliasing.

References

[1] P. Bates and J. Wileden.
High-Level Debugging of Distributed Systems: The Behavioural Abstraction Approach.
Technical Report COINS 83-29, University of Massachusetts, 1983.

[2] Gregor V. Bochmann.
Semantic Evaluation from Left to Right.
Communications of the ACM 19(2):55-62, February, 1976.

[3] W.F. Clocksin and C.S. Mellish.
Programming in Prolog.
Springer-Verlag, Berlin, 1987.

[4] R. S. Curtis and L. D. Wittie.
BUGNET: A Debugging System for Parallel Programming Environments.
In *3rd International Conference on Distributed Computing Systems*, pages 394-399. October, 1982.

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

[6] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, 1977.

[7] C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[8] C.A.R. Hoare.
Communicating Sequential Processes.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[9] Wenwey Hseush and Gail E. Kaiser.
Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language.
In *ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging.* Madison, WI, May, 1988.
To appear.

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

[11] Gail E. Kaiser.
Generation of Run-Time Environments.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 51-57. Palo Alto, CA, June, 1986.
Special issue of *SIGPLAN Notices*, 21(7), July 1986.

[12] Gail E. Kaiser, Simon M. Kaplan and Josephine Micallef.
Multiuser, Distributed Language-Based Environments.
IEEE Software :58-67, November, 1987.

[13] Gail E. Kaiser and Simon M. Kaplan.
Reliability in Distributed Programming Environments.
In *Sixth Symposium on Reliability in Distributed Software and Database Systems*, pages 45-55. Kingsmill—Williamsburg, VA, March, 1987.

[14] Simon M. Kaplan and Gail E. Kaiser.
Incremental Attribute Evaluation in Distributed Language-Based Environments.
In *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 121-130. Calgary, Alberta, Canada, August, 1986.

[15] Uwe Kastens.
Ordered Attribute Grammars.
Acta Informatica 13:229-256, 1980.

[16] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[17] Leslie Lamport.
Time, Clocks and the Ordering Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[18] Thomas J. LeBlanc and John M. Mellor-Crumley.
Debugging Parallel Programs with Instant Replay.
Technical Report, University of Rochester, September, 1986.

[19] Alberto Martelli and Ugo Montenari.
An Efficient Unification Algorithm.
ACM Transactions on Programming Languages and Systems 4(2):258-282, April, 1982.

[20] B.P. Miller and J.D. Choi.
Breakpoints and Halting in Distributed Programs.
Technical Report, University of Wisconsin-Madison, July, 1986.

[21] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[22] Thomas Reps.
Generating Language-Based Environments.
The MIT Press, Cambridge, MA, 1984.

[23] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 41-48. Pittsburgh, PA, April, 1984.
Special issue of *SIGPLAN Notices*, 19(5), May 1984.

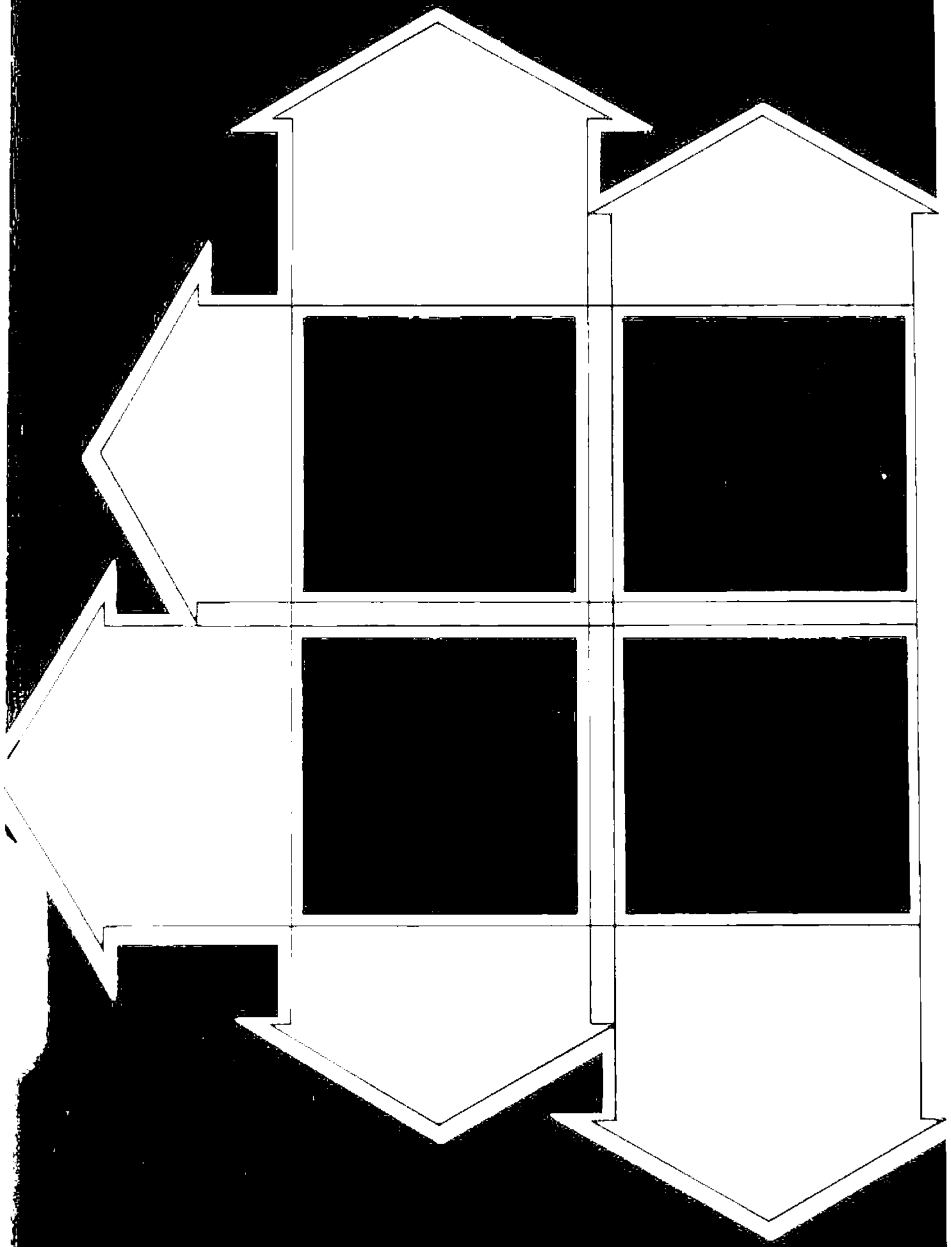
[24] Ehud Y. Shapiro.
A Subset of Concurrent Prolog and Its Interpreter.
Technical Report TR-003, Institute for New Generation Computer Technology, February, 1983.

[25] E.T. Smith.
Debugging Tools for Message-Based, Communicating Processes.
In *4th International Conference on Distributed Computing Systems*, pages 303-310. May, 1984.

[26] Richard Snodgrass and Karen Shannon.
Supporting Flexible and Efficient Tool Integration.
In Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), *Lecture Notes in Computer Science.* Volume 244: *Advanced Programming Environments*, pages 290-313. Springer-Verlag, Berlin, 1986.

[27] Joseph E. Stoy.
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.
The MIT Press, Cambridge, MA, 1977.

PROCEEDINGS



The 8th International Conference on Distributed Computing Systems

San Jose, California
June 13-17, 1988

SPONSORED BY

THE COMPUTER SOCIETY
Technical Committee
on Distributed Processing

THE INSTITUTE OF ELECTRICAL
AND ELECTRONICS ENGINEERS, INC

IEEE

Computer Society Order Number 865
Library of Congress Number 87-83544
IEEE Catalog Number 88CH2541-1
ISBN 0-8186-0865-X

COMPUTER
SOCIETY
PRESS