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 runtime 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

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.

```plaintext
IF :: body: sequence of GC
    RUN -> Propagate RUN To body
    CONTINUE On body[exp] -> Propagate RUN To body[inst]
    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]. 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

---

2Note 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.
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 "receive statement" 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 

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, identname an identifier 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

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

2

252
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: iden
tid

argument: EXPRESSION

return: iden
tid

RUN -->

Assert (argument!, self!, receiver!)

Propagate WAIT to self

WAIT -->

Block (return.value, self!, receiver!)

Propagate CONTINUE to self

**PROCEDURE ::=**

sender: iden
tid

body: STATEMENT

result: iden
tid

RUN -->

Block (argument.value, sender, self!)

Propagate RUN to body

CONTINUE On body -->

Assert (result.value!, 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!) -->

**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: iden
tid

locals: sequence of iden
tid

body: sequence of STATEMENT |
| RUN -->
| Propagate RUN to body[1] |
| CONTINUE On body[next] -->
| 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 attributes 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 the side-effect of a previous language statement, 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:: [ integer j ;
    j := 3*i ;
    ]
]
```

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 move quicker, 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 value, 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 continue until a unifiable result of the variables is found (in process P).

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 over-head 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 GAIA. We’d like to thank Yael Ccowicz, 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.

2We determine such dependency either optimistically assuming no aliasing or pessimistically assuming maximum possible aliasing.
References


