Computer Languages, Vol. 5, pp. 163 to 189
Pergamon Press Ltd 1980. Printed in Great Britain

0096-0551/80 1201-0163$02.00/0

CONCUR: A LANGUAGE FOR CONTINUOUS, CONCURRENT PROCESSES

RICHARD M. SALTER[1]*, TERENCE J. BRENNAN[2] and DANIEL P. FRIEDMAN[2]†

[1]Drexel University, Philadelphia, PA 19104, and [2]Indiana University Bloomington, IN 47405, U.S.A.

(*Received* 13 *December* 1979; *in revised form* 22 *August* 1980)

Abstract—Hendrix's robot modeling system presented a simulation method in which time is represented as a continuous phenomenon. This paper introduces the language CONCUR, which realizes Hendrix's concept through an extension of the LISP environment. CONCUR uses generalized procedures (scenarios) operating in a data-driven mode to implement Hendrix's events. The heart of CONCUR is a generalized pattern-matcher which permits operators within the patterns to bind variables and modify the match process. We include several detailed examples in addition to an implementation of the pattern matcher.

Continuous simulation Robot modeling World modeling Processes

1. INTRODUCTION

Hendrix made two major contributions in his paper [1] on robot modeling systems. The first was to robotics exclusively. Since the robot represents its environment with a world model, which is a structure consisting of objects, relationships and operators, Hendrix suggested that we consider world models with many causal agents. Such a model would be a collection of on-going processes, much like the processes in a simulation system.

Hendrix's second contribution was in the design of a new simulation system appropriate for artificial intelligence research. In this system "events" are characterized by (1) initiation conditions; (2) continuation conditions (i.e. conditions which predicate the continued instantiation of the event); and (3) effects on the world. The system becomes responsible for event invocation and duration. In terms of the robot model, the robot is freed of its omnipotence—its participation in every event. The world becomes a universal interface between events.

One resulting innovation is the perception of time as a continuous phenomenon. This can be simulated by using time dependent functions as parameters. Any reference to such a parameter results in the evaluation of the expression which represents the function at the given point in model time. Similarly, initiation conditions and continuation conditions are implemented as collections of facts regarding the world. If all initiation conditions hold at a given moment, the event initiates and continues as long as all of the continuation conditions are present.

Hendrix simulates parallelism by utilizing a process monitor, which maintains a set of process models and the world model as well. Events are controlled from this monitor by control blocks which exist throughout the duration of a given process invocation. The monitor acts as a medium for communication between each process and the rest of the world by monitoring the control blocks and creating and destroying blocks as needed. Progress in time is simulated within the monitor by advancing the model clock between moments of activity. With this methodology, Hendrix is able to achieve (as described in the abstract in Ref. [1]) "a mechanism which makes possible the modeling of (1) simultaneous, interactive processes; (2) processes characterized by a continuum of gradual change; (3) involuntarily activated processes (such as the growing of grass); and (4) time as a continuous phenomenon."

* This author's work supported in part by the National Science Foundation under grant number MCS 80-04130 and by the Office of Naval Research under grant number N00014-80-C-0752.

† This author's work supported in part by the National Science Foundation under grants numbered MCS 75-06678 AOI, MCS 79-04183 and MCS 77-22325.

Hendrix's simulation system was originally implemented in FORTRAN. The notation was more specific than was necessary, for example time-dependent variables in Hendrix's notation were distinguished, and every reference to a time-dependent variable was mediated by the system. Maintenance of time-dependent variables in this way required much machinery. (Lowrance and Friedman [2] faithfully implemented Hendrix's notation.) Hendrix sacrificed elegant notation to allow for an efficient implementation, envisioning that implementation as a complex program manipulating complex data structures. For example, time-dependent variables were distinguished as such in Hendrix's system.

We envision a different type of implementation, a system that does away with the need for Hendrix's notational devices. For example, we want time-dependent variables to be indistinguishable from those that are time-independent. In order to clarify and generalize Hendrix's ideas, we choose to implement his world modeling system as a language CONCUR. CONCUR is implemented in and is an extension of LISP, which the reader is assumed to know.

By implementing Hendrix's system as a language, much power and flexibility is achieved. Events may perform arbitrarily complex calculations at any point in a program. A construct invented for a specific use, such as * for time-dependent variables, can be made generally available. The language therefore makes more general use of the power of the Hendrix system.

CONCUR embodies three major innovations. First, CONCUR makes no explicit reference to simulation. On the surface, an event simply contains an initiation condition, manipulates the world model, and specifies a continuation condition. Simulation is the resultant side-effect of these actions. An event does not explicitly refer to other events; it influences other events indirectly, through its effects on the world model. The programmer's environment is much simpler.

The second innovation is CONCUR's event initiator, which is a generalized pattern matcher. An event is a demon, a data-driven function. CONCUR does not use "if-removed" or "if-used" demons [3], but instead employs logical formulae to specify the state of the world model which should initiate events. Event initiation can be made very simple, exact and efficient.

The third innovation is the unification of LISP. Functions, assertions and variables are seen to be instances of a general entity, called a pattern, and are evaluated using identical methods. As a result, they are indistinguishable. Moreover, an expression is indistinguishable from the function it invokes. This unification of LISP is accomplished through sophisticated pattern matching.

CONCUR utilizes much of the methodology of earlier knowledge-based problem-solving systems. The use of pattern-matching for event invocation was suggested by PLANNER [4]. The event scenarios, which use and-or trees in their initiation conditions, strongly suggest production rule systems such as MYCIN [5]. CONCUR differs fundamentally from these systems inasmuch as it is not generally interactive, and it maintains a control structure which is data-driven rather than goal-driven. CONCUR, unlike the robot system STRIPS [16], is not problem oriented, but does operate using inference rules as provided by the scenarios. What is unique about CONCUR is that it uses generalized procedures to drive a system in which time appears as a factor. CONCUR was introduced in Ref. [6] as a generalization of the system presented in Ref. [2].

In the next section we present an overview of CONCUR, and develop the language more formally in Sections 3 and 4. Section 5 contains some examples, and our conclusions appear in Section 6.

2. THE BASIC ELEMENTS OF CONCUR

In this section we shall introduce the reader to CONCUR by developing a single event, a robot in motion. First, however, we must consider a crucial point: the representation of the world model. The most natural way to represent this structure is as an extension of the LISP "environment", or "association list". While the LISP structure is

limited to assigning values only to atoms we impose no such restrictions. Thus, more complex entities which we shall call items, assertions or patterns can be given values, and the list of such item/value pairs comprises the world model. The notation for such a pair will be "item ≡ value".

Some entries in the world model describe the current state of the world by either having assertions as items with value TRUE (T), or by assigning current values as parameters. Thus one can describe a robot's location by either asserting (robot at A) ≡ TRUE or assigning (robot location) ≡ A. In an optimal implementation either could be used. The subset consisting of such world model elements shall be called the state of the world model (SWM).

Among the assertions in the SWM may be some which match the components of an event initiation condition. If these assertions produce values which cause a total match (described below) with the initiation condition, then the event body (which is the world model "value" of its initiation condition "item") is invoked. The item/value pair "initiation condition" ≡ "event body" will be called a scenario [4].

As a simple example, consider a robot which is to move from outside to inside a room. The command (robot move in room) ≡ TRUE is placed in the SWM. If the initiation condition of the event which moves the robot consists only of the assertion (robot move in room) then the event can be invoked, since the expression matches the SWM assertion. To change the location, we use ((robot out room) → (robot in room)). The operator → replaces the item field of the pair (robot out room) ≡ TRUE with (robot in room), yielding the new pair (robot in room) ≡ TRUE and effectively changing the SWM as desired. (→ is an assignment operator which alters the item constituent of an SWM entry. Its more common counterpart, ←, will appear as well further on).

Since the task has now been completed, we must remove the command from the SWM with (≢ (robot move in room)). The scenario for the event is

```
((robot move in room)                                    (1)
   ≡
 (block
     ((robot out room) → (robot in room))
     (≢ (robot move in room))  ))
```

("block" is a function which evaluates all of its arguments).

Already at this point several important issues must be clarified. Although the "initiation condition" ≡ "event body" form is clear (so that the above scenario could appear exactly as shown in the world model), we have informally introduced three functions, block, →, and ≢, which have a side-effect on the world model. Indeed, this is the only action taken. We must, however, clarify how these functions treat their arguments. By introducing complex patterns as arguments it will also turn out that we have introduced more complex evaluation schemes (although it appears here that neither → nor ≢ evaluates its arguments, this is not quite the complete story). The method by which functions will be defined and argument-handling determined will be discussed in Section 4, but if → is determined to be strict [8] in its left argument, in the sense that it is undefined if that argument does not match an item in the world model, the event will not work when the robot is already inside the room, because the assertion (robot out room) is not part of the world. The event would therefore abort.

To solve this problem, we should ensure that (robot out room) is in the world, i.e. (robot out room) should also be part of the initiation condition. Thus two conditions must hold; that is to say, the conjunction of the conditions must hold. Denoting the conjunction by (& (robot move in room) (robot out room)), the event becomes

```
((& (robot move in room)                                 (2)
   (robot out room))
   ≡
 (block
     ((robot out room) → (robot in room))
     (≢ (robot move in room))  ))
```

(NOTE: The use of & in the above suggests that initiation conditions can be written as logical formulae. We can complete the analogy by defining ($\vee$ P1 ... PN) to hold if one of the Pi matches an SWM assertion, and ($\neg$ P) to hold only when P does not match anything in the SWM. T and NIL are assertions that are respectively always part of, and always not part of, the SWM. Any logical formula built from assertions can be used as an initiation condition.)

An event to move from inside to outside a room could also be written, and would be similar. It's desirable to write a single event to cover both cases. In such an event, entering and exiting must be distinguished; a simple distinction is the command specification. The commands (robot move in room) and (robot move out room) differ only in the direction specification.

In order to introduce this kind of generality, we must be able to bind the specified direction and use the bound value in the event body. The value will depend upon the assertion present in the SWM which initiates the event. We express this binding syntactically in the pattern by prefixing the local variable with =, to indicate that the component is a template for matching any SWM assertion which could instantiate the prefixed variable. In our example we could use (robot move =to room) as a template for (robot move out room) or (robot move in room), which when matched binds "to" to "out" or "to" to "in" respectively.

To write the event body, we must somehow be able to use $\rightarrow$ with the unspecified argument patterns. Since the robot has to be somewhere, the pattern (robot at ?x), where ?x indicates a universal (but consistent) template, will always seek the SWM item describing the robot's location and so we can use this for our left argument. The right argument uses the binding: (robot *to room), where * indicates evaluation of the prefixed quantity before matching, and so we get

```
((robot move =to room)                                    (3)
≡
 (block
   ((robot ?x room) → (robot *to room))
   (≠ (robot move *to room))      )).
```

Note first that $\rightarrow$ and $\neq$ must both be designed to evaluate those components of its arguments for which evaluation is explicitly called. Thus the arguments are "quasi-quoted" [17] or, using our terminology, "unlisted" in our design for $\rightarrow$ and $\neq$. We also note that if *to and ?x happen to be identical (so that the robot was told to go where it already was), there is no change except the deletion of the command.

An obvious generalization of (3) is having more places for the robot to move. Suppose locations A, B, C, D, and E are in a room. Let (robot move to X) be the command and let the robot's location be denoted (robot at Y). The scenario is very similar:

```
((robot move to =to)                                      (4)
≡
 (block
   ((robot at ?x) → (robot at *to))
   (≠ (robot move to *to))   )).
```

Now we shall introduce the concept of duration to the events by adding the SWM assertions (speed robot) and (distance between A and B), with corresponding values, so that the duration required for the execution of this event is ((distance between A and B)/(speed robot)). We add to the scenario the function (resume after t), where t is evaluated, which acts to separate the preceding and succeeding statements by an elapsed model time of t. The resulting scenario is

```
((&(robot move to =to)                                    (5)
   (robot at =from))
≡
 (block
```

```
(≠ (robot move to *to))
((robot at *from) → (robot moving from *from to *to))
(resume after ((distance between *from and *to)/
                (speed robot)))
((robot moving from *from to *to) → (robot at *to))  )).
```

The execution of the resume function causes model time to be consumed. After the given amount of model time elapses, resume terminates, and the next line is executed (All other expressions in the block, before and after the resume, can be viewed as executing simultaneously.) In (5), (speed robot) is an assertion in the world model with a value used in the computation, and (distance between *from and *to) also matches an assertion in the SWM whose value is used. Scenario (5) also includes the first appearance of an arithmetic operator, /, which we have assumed behaves in the usual fashion and evaluates its arguments. This shall be true for +, − and × as well.

Our final version of this event introduces the crucial concept of continuity. Scenario (5), the most sophisticated so far, only allows for three discrete states to be placed in the SWM (and hence made known to the rest of the events). We could conceive of the robot as being on a straight doubly infinite track with each point on the track denoted by a real number. In moving from "from" to "to" (now real numbers) the robot's location at model time t would be

$$(\text{from} + (\text{speed robot}) \times (t - t_0) \times (\text{sign } (\text{to} - \text{from})))$$

where t_0 is the time at event initiation (we have assumed that all arithmetic functions evaluate their arguments). The robot's location, now a time dependent variable, must be identified with this formula.

We now represent the robot's location as a parameter which is given a value in the SWM, e.g. ((robot location) ≡ 2.5). At times when the robot is in motion, the value of this parameter will be the formula described above. The assignment operator ←, which unlists its arguments, can be used to make the replacement. We use the global variable time, which always evaluates to the current model time, and as a result of unlisting, t_0 becomes *time. Thus it would seem that having

```
((robot location) ← (*from + *(speed robot) × (time − *time)
                                × *(sign (to − from))))
```

would be sufficient. However, performing the evaluation *(robot location) would produce the formula itself, and not its value, which would require **(robot location). This is undesirable since it requires any event referencing (robot location) to know when a formula is present—this fact should be invisible. We solve this problem by having * coerce any leading operators in the value it produces and by passing a "quoted" * as a prefix to the formula. The quoting is accomplished by the prefix |, derived from a LISP character which performs a similar service, referred to as an "unspecial". The combined prefix |* signifies to any unlisting that * is to be passed into the function literally, effectively delaying its invocation. Thus

```
((robot location) ← |* (*from + (time − *time)
                          × *(speed robot)
                          × *(sign (to − from))))
```

will put, say

$$(\text{robot location}) \equiv *(4.1 + (\text{time} - 2.121) \times 3.1 \times -1.0)$$

into the SWM. The complete scenario for a robot in motion is therefore

```
((robot move to =to)                                    (6)
≡
(block
  (≠ (robot move to *to)
```

```
((robot location) ← |*(*from + (time − *time)
                        × *(speed robot)
                        × *(sign (to − from))))
(resume after ((abs (to − (robot location)))
                    / (speed robot)))
((robot location) ← *to)  )).
```

Both unlist and unspecial will be described more fully in Section 3.

We can generalize this event by removing the robot from the track and allowing it to move freely within an N × M rectangle. Each location is a point (x,y), with $0 \leqslant x \leqslant N$ and $0 \leqslant y \leqslant M$. When it is in motion the value of (robot location) in the SWM becomes (F_1, F_2), where each F_i has a similar form as the formula in (6). We could write a scenario analogous to (6) by introducing the functions *one* and *two* (which produce the first and second elements, respectively, of their arguments). However, it is also necessary to augment the initiation condition to reflect the finite robot domain, i.e. the final location (to1 to2) should be inside the rectangle. This can be accomplished by an SWM assertion, dimension ≡ (N, M), and an initiation condition of the form

```
(&(robot move to (=to1 =to2))
    *(0 ⩽ to1 ⩽ (one dimension))
    *(0 ⩽ to2 ⩽ (two dimension))).
```

Using * in an initiation condition in this manner indicates evaluation of these pattern predicates before matching (recall: T always and NIL never matches the SWM).

Another important facility for complex events is the ability to allow the duration to be controlled by SWM conditions. For instance, the robot could move as described above unless it receives a new command to move. By enhancing the previously defined resume statement to (resume after t unless c), where c is a pattern that is unlisted, any event continuing over time can continuously check the SWM for a match with c, and terminate the waiting period as soon as such a match no longer succeeds. (The "unless c" is, of course, optional).

Finally, we allow for the fact that there may be more than one robot present in the room by using "robot" as a variable. We have the following scenario:

```
((& (=robot move to (=to1 =to2))                                    (7)
  *(0 ⩽ to1 ⩽ (one dimension))
  *(0 ⩽ to2 ⩽ (two dimension)))
≡
 (block
  (≢ (*robot move to (*to1 *to2))
  ((*robot location) ←
    (|*(*(one (*robot location)) + (time − *time)
       × (speed *robot) × (sin (angle (*robot location) (to1 to2))))
    |*(*(two (*robot location)) + (time − *time)
       × (speed *robot) × (cos (angle (*robot location) (to1 to2))))))
(resume after ((distance (*robot location) (to1 to2))/(*speed robot))
       unless (& (¬(robot move to ??)
              |*(eq (speed *robot) *(speed *robot))))
((*robot location) ← *(*robot location)))  ).
```

Here (cos θ), (sin θ) and (angle v1 v2) are obvious.

The continuation conditions in the resume statement require that no other move command be present, and that the robot's speed remains constant. Functions which normally require an SWM value, such as the arithmetic operators, distance, cos, sin and angle evaluate their arguments, while those which are driven by pattern matches, such as the continuation condition part of the resume (following "unless") unlist theirs. These specifications will turn out to be at the user's discretion.

This last event is sophisticated and powerful, but must fit into the world and interface cleanly with all other events. For example, if a robot is grasping an object and the robot moves, the object moves with it. The event that does this can be represented by the following scenario:

```
((& (=robot grasp =object)
    (*object movable))
  ≡
 ((*object location) ← |*(robot location))).
```

Any reference to this object's location will unknowingly become a reference to the robot's location.

We have not even begun to cover the complex control problems of even this simple case (for example, with several robots moving simultaneously, there is the possibility of collisions). What we have done is present a medium in which we can deal with such problems. This medium relies on decomposing the control into a system of cooperating modules which have the advantage of being easy to write and general enough to allow changes in the simulation to affect only a few modules (thus promoting experimentation). To summarize this section, we have shown the following about CONCUR:

1. The principal data structure, the SWM, is modeled after the LISP association list, taking the form

item ≡ value.

2. An event appears in the world model as a scenario:

initiation condition ≡ event body.

3. The initiation condition is a pattern, which is matched against the SWM, seeking to invoke the event body.

4. Arithmetic, and relational operations, and operations which change the world model both initiation conditions and event bodies. The effect of an event appears as a side-effect to the SWM.

5. Pattern prefixes are available for the sophisticated pattern–matching used in all aspects of CONCUR's activity. These will be discussed in the next section.

6. Time-dependent values are indistinguishable from those constant over time.

7. Event duration can be controlled by continuation conditions, or be fixed.

8. Complex control problems can be handled as small interacting modules.

3. FUNDAMENTAL PROPERTIES

The heart of CONCUR lies in the generality of its pattern matcher. This is accomplished through the use of operator prefixes (some of which appeared in the last section), and an approach which reduces the distinction between pattern template and data. In this section we shall examine the operator prefixes in detail, and show how pattern matching can be used to control the simulation achieved by CONCUR. A detailed description of the implementation of the pattern matcher will be given in the next section.

In all, there are ten prefixes which can be concatenated to the left of a pattern: *, :, $, !, ?, ', =, $\hat{*}$, $\hat{:}$, and $\hat{\$}$ (the last three shall be referred to as "modified"). The purpose of these prefixes is to change expressions during the matching process, or alter the process itself.

We have already seen *, =, and ? used in the last section, and have discussed them informally. Let us now consider these operators once again in a more formal setting.

The binding operator, =, should not be confused with ←, as the relationship is analogous to that between LISP lambda variables and setq. Each scenario maintains a copy of the SWM onto which these bindings are added, and which disappear once the event has terminated. Thus the scope of such bindings can only extend into the event (and any subsequent event invoked from within). It should be noted that a match in which one of the elements is prefixed by = will not be continued into the substructure of the prefixed element. Thus matching =(robot location) against (robot at A) (or, notationally, =(robot location) μ (robot at A)) will succeed, in spite of the fact that (robot

location) does not itself match (robot at A), and produce the pair (robot location) ≡ (robot at A). When it appears, = must be the leftmost prefix.

The evaluator, *, is analogous to LISP eval but faces one additional problem in that since list structures can be given values it is impossible to distinguish between variable or constant and functional evaluation, as in the case of eval. The order of attempted evaluation is as follows:

1. atoms:
   (a) CONCUR evaluated SWM value of matching assertion
   (b) primitive value;
2. prefixed expressions: evaluation after applying prefixes (i.e. unlisting);
3. quoted expressions: expression;
4. lists:
   (a) CONCUR evaluated SWM value of matching assertion
   (b) Application of primitive function to CONCUR-evaluated arguments.

(Primitive values include numbers and truth values; primitive functions include LISP primitives.) These semantics make possible the writing of functions as SWM pairs, as discussed below.

The unlist operator, :, causes the prefixed expression to be returned with all internal prefixed expressions processed. We saw in the last section that unlisting is an important technique for handling function arguments, as it substitutes values for identifiers at various points in a pattern argument. We have also seen that it is possible to pass an operator literally into an unlisted expression by protecting it with |. Another way of protecting an expression is by quoting it, whereby the unlisting will not continue into the expression (note that the quote itself can be protected by |). Unlist is specified as follows:

: atom = atom
: ⟨op⟩⟨expression⟩ = appl(⟨op⟩, : ⟨expression⟩)
: |⟨op⟩⟨expression⟩ = ⟨op⟩ : ⟨expression⟩
: '⟨expression⟩ = ⟨expression⟩
: (S1 .. Sn) = (:S1 .. :Sn)

where appl(a,b) represents the application of operator a to pattern b.

The prefix ? represents a consistent universal match with any other atom. The atom ?? represents a universal match which need not be consistent. The operator ' represents the usual QUOTE.

Much versatility is achieved using the remaining prefixes. Let us first consider the spanning operator, !. With the machinery already developed, we are not able to implement the action of functions which take an arbitrary number of arguments (like LISP FEXPRs and MACROs). In such functions the list of arguments is bound to a single variable. To accomplish this we introduce the spanning operator for the purpose of matching arbitrarily many arguments.

Whenever ! appears as a prefix, the other pattern is spanned until the match can continue at the same level (or otherwise fail). If ! prefixes the same element more than once, the span must be consistent. For example, (robot !do for me) will match (robot go to the store for me), (robot (go to) the (store) for me), and (robot (go to (the store)) for me), but not (robot (go to the store for me)) or (robot (go to the store) (for me)). If we have (robot !do for me now and !do for me later), both instances of !do must match the same span, thus (robot go to store 1 for me now and go to store 2 for me later) will not match.

The operators which we have examined so far perform useful and necessary alterations on patterns but it is often necessary to alter a pattern in a special way, such as transforming X to (quote X), or to test a pattern before attempting the match. Do-it-yourself operators can be designed using the extension prefix, $. Syntactically the $ serves as a delimeter for an extension expression, i.e. $e$exp, which is itself modeled as a SWM pair.

The semantics of the extension operator are simple. The "item" of the extension

expression is matched against the prefixed operand. If this match succeeds, then the unlisted "value" of the extension expression is returned (this unlisting occurs in an environment in which any bindings obtained during the "item" match are maintained). Otherwise the overall match fails.

For example, the aforementioned alteration is accomplished by $(=X ≡ (quote *X))$Y. In another example, the operand is tested as to whether it is a list by $((=!X) ≡ *X)$. As a convenience, and for the purposes of recursion, expressions of the form $*X$ will substitute the extension expression bound to X. Thus placing listp ≡ ((=!X) ≡ *X) in the SWM allows $*listp$var to test for a list.

The three modified operators, $\hat{*}$, $\hat{:}$, and $\hat{\$}$ are identical to their unmodified counterparts except that they are applied to the corresponding element in the pattern being matched. Thus if place ≡ house, then (robot go to $\hat{*}$house) would match (robot go to place), since the evaluator is applied to "place" before matching (this is equivalent to (robot go to house) μ (robot go to *place)). As another example, if =$\hat{\$}$*listp$\hat{\$}$var appears in a pattern, the binding will take place only if the element being matched is a list, so that $\hat{\$}..\hat{\$}$ acts as a filter for the binding operator. The use of *one* and *two* in (7) of Section 2 could have been eliminated by placing in the initiation condition $\hat{\$}$((robot location) ≡ *(robot location))$\hat{\$}$(=from1 =from2), binding each component separately (much like an embedded lambda in LISP). Finally, to illustrate the power of using recursion in defining operators, we present the wff operator, which is designed to detect logical formulas. Defining a logical formula using BNF style [9] as

⟨logic form⟩ ::= atom |
(AND ⟨logic form⟩{⟨logic form⟩})|
(OR ⟨logic form⟩{⟨logic form⟩})
(NOT ⟨logic form⟩)

then we have

wff ≡ ((&(∨ ??
(AND !$\hat{\$}$*wff$\hat{\$}$X)
(OR !$\hat{\$}$*wff$\hat{\$}$X)
(NOT $\hat{\$}$*wff$\hat{\$}$X))
=X)
≡
*X)

If modified operators are to be applied to a pattern possessing its own operator string, then the string of modified operators is essentially concatenated onto the left of the other string (but to the right of =). Thus $\hat{:}$*expl μ = *$e$exp2 results in expl μ = :**$e$exp2. We can therefore require, without loss of generality, the following syntax for operator strings:

⟨op string⟩ ::= {=}{$\hat{*}$, $\hat{:}$, $\hat{\$}e\hat{\$}$}{!}{$\hat{*}$, $\hat{:}$, $\hat{\$}e\hat{\$}$}{*, :, e}|?|'.

It should be pointed out that those modified operators to the right of ! are applied term by term, while those to the left are applied to the entire spanned list.

The generality of these pattern operators makes it possible to define functions within the framework of the world model. Consider, for example, the assignment function ←, which was used frequently in the last section. It is possible to describe a "call" to this function with the template (=$\hat{:}$before ← =$\hat{:}$after), which will in fact bind the arguments. We can use the pattern matching mechanism to control functions by writing function definitions using this specification as initiation condition (this may appear difficult since functions of this sort seem to be outside the language, but recall that * has the capability of passing the evaluation into LISP). The pattern matching machinery will then execute function calls naturally within the context of its normal operation. Functions therefore become small events (or events become large functions).

Specifically, a function specification does two things:

(1) The specification describes the format of an invocation of the function. For example, (=:before → =:after) will match any three element expression whose middle element is →. Thus any such expression becomes an invocation of the → function. If an expression matches a function specification, then the expression invokes that function.

(2) The specification describes the treatment of parameters. It can call for evaluation, unlisting and/or binding. For example, (= :before→ = :after) specifies that the first and third elements of the matched expression are to be unlisted and both values are to be bound. This use of = is analogous to binding parameters in events.

Any prefixes may be used in a specification. The result of all this is that a function used in manipulating the SWM can itself be placed there. This includes LISP functions as well.

In the light of this technique for handling functions we can explain our syntax for arithmetic operations. We use the specification (=*left + =*!right) for addition, and similar specifications for the other operators, and the reader should recognize this as being able to handle strings of arithmetic operations, e.g. (A + B × C + D), which are associated to the right. The spanning operator can also be used to implement a LISP FEXPR by simply writing (fn =!param) or (fn =!*param), if one wants the arguments evaluated, as in list.

With the concept of function as event, we see that it is possible to implement a single data structure—the world model or association list—to hold functions (specification ≡ function body), events (initiation condition ≡ event body) and SWM data (item ≡ value). The distinction between each of these different "types" is reduced by the fact that pattern matching is used in each case as the accessing tool. There is no distinction made between values obtained through functional application or as a result of evaluating constants, and the fact that a scenario has been invoked is felt only through side-effects to the SWM.

Because of the similarity in the evaluation of functions, assertions and initiation conditions, the differences between them is much less pronounced than in other systems. For example, (a b) can be a variable name. When the expression (a b) is evaluated, the association list is scanned for a match to (a b). If (a b) ≡ X, for some value X, then the match of (a b) with itself allows X to be accessed. Hence (a b) can act like a variable. As another example, consider a world with several objects, each classified as movable or immovable. Movability for a particular object is determined by the pattern (obj movable). Two solutions are immediately possible. First, assertions of the form (bkt movable) ≡ TRUE could be in the SWM. Second, a function can be used: (=obj movable) ≡ *(member obj '(...)), where (...) is the list of movable objects. Both assertions and functions are evaluated the same way, hence the set of assertions is indistinguishable to the user from the function. Many other examples can be found by blurring the distinctions between functions and assertions.

Initiation conditions use &, ∨ and ¬ to form logical formulas. If we also allow these logical connectives to appear in assertions, a third solution to the movability problem can be devised. We use the assertion (movable (∨ ...)) ≡ TRUE, where once again ... represents the movable objects. Any movable object A will match (∨ ...), so (movable A) will match (movable (∨ ...)).

To reiterate, applications of functions, initiation of events and evaluation of functions or variables are all controlled by the pattern matcher, which scans the association list for matches between its constituents.

4. THE PATTERN MATCHER

Pattern matching is usually seen as matching a pattern against data. Patterns are viewed as generalized expressions, which can be used to describe and match various data. Each datum is seen as representing only one piece of information.

In CONCUR, the distinction between pattern and data, like many other distinctions, is not made. The three solutions to the movability problem above provide a good

example. The conventional view would restrict us to placing an assertion for each movable object into the SWM. However, using generalized data such as the function (movable =X) ≡ *(member X (...)), or the pattern (movable (V ...)) ≡ TRUE, in a setting in which knowledge of the method used to represent the data is not required clearly adds power and versatility to the system.

This power derives from the ability of "data" to contain operators and logical formulas. We shall not distinguish between "pattern" and "data", and only use the term "pattern". All pattern matching takes place between pairs of patterns.

Recall that the act of matching patterns is denoted by P1 μ P2. Since both arguments to the match are patterns, we can also consider P2 μ P1, and we shall require P1 μ P2 = P2 μ P1. Before considering the structure of the pattern matcher, we shall examine the syntax of patterns. This is given by the following BNF:

⟨full pattern⟩ ::= ⟨atom⟩ | ?⟨atom⟩ | ?? | ⟨logical formula⟩ |
⟨s-pattern⟩
⟨logical formula⟩ ::= ⟨s-pattern⟩ |
(& ⟨logical formula⟩{⟨logical formula⟩}) |
(∨ ⟨logical formula⟩{⟨logical formula⟩}) |
(¬ ⟨logical formula⟩)
⟨s-pattern⟩ ::= ⟨pattern element⟩ |
(⟨pattern element⟩{⟨pattern element⟩})
⟨pattern element⟩ ::= ⟨full pattern⟩ | ⟨op string⟩⟨full pattern⟩

where ⟨atom⟩ is analogous to LISP atom, and ⟨op string⟩ was defined in the previous section.

Patterns are therefore LISP-like s-expressions which can be modified element-wise by prefixes and overall by logical functions.

We now describe the LISP implementation of the pattern matcher which appears in the appendix. It should be noted at the onset that this particular implementation was designed to accept LISP input, so that some modification to the operator-prefixing syntax was made. Here operator strings have become "pseudo lists", so that =:*$(e1 e2)$robot becomes (= : ^*(e1 e2) $ robot). To maintain the distinction between these structures and s-patterns, certain data structuring functions were required. It is possible to implement the operator syntax exactly using explode (or read macross), however this would not do away with the data-structure problem (we would then have to distinguish pattern elements from atoms), and would make the code in Appendix II unnecessarily complicated. The functions prop, cmop, op, push and pop are used to detect unmodified, modified, or either type of prefix, and to add on or delete a prefix to or from the left of an operator string, respectively. We shall not concern ourselves with this any further.

The overall concept of the pattern matcher is to accept a pair of patterns and return a structure called the match record if the match succeeds (and NIL otherwise). This structure is composed of the local association list (the world model updated with local bindings) and a second association list used only by the pattern matcher for maintaining consistency in matches involving ! and ?. The latter consistency table is accessed via the function consistent, which is used to check and update this structure when necessary.

As described above, performing matches against the set of world model items is the principal control structure of the language, and so to facilitate such matches we structure the world model to consist of a list of all "items" followed by a list of their "values" in corresponding locations. We have designed the constructor (record) and accessing (assq) functions to reflect this (the consistency table is also implemented this way so that consistent can use record and assq). The most striking result of this is the simplicity of worldmatch.

A match between patterns P1 and P2 begins with a call to pmatch with an initial match record consisting of the world model and a null consistency table. Pmatch checks for the various full pattern possibilities in either argument (atom; logical formula—and, or, not; pattern element; s-pattern; in that order), and disposes of the match accordingly.

The iftype–result construct which appears here is a programming tool for checking both P1 and P2 for a particular category.

Once one of the patterns falls into a category, the match is sent on (to almatch, andm, orm, notm, elm, or smatch, respectively) and the remaining categories must be checked for the other pattern. Matches between atoms are one form of bottom match and require exact matching (i.e. eq), except for ??. If only one of the patterns is an atom the match can continue if the other is either a non-trivial pattern element or a logical formula.

The presence of one of the three logical connectives will set up calls to pmatch involving the constituent arguments, with the local SWM continually being updated with bindings as the match proceeds. Andm is called ahead of orm or notm because of a special control structure which must be inserted in the event that the match is between conjunctive and disjunctive patterns. In such a match it is possible that more than one of the disjuncts can successfully match a given conjunct, however the bindings which result from proceeding with one of these might block further success in the match while those from another might not. Our implementation of andm always introduces the necessary backtracking mechanism since any given pattern can be regarded as a disjunction of one argument. The relationship between "and" and "or" is important to the matching of scenario initiation conditions since a match against the world can be implemented as a match against a pattern obtained by "or-ing" the items of the world model. The backtracking requirements are built in as a part of the general matching structure. Orm and notm are guaranteed simpler patterns, and so are implemented in an obvious fashion.

The most difficult (and most common) aspect of matching comes in matching s-expressions of pattern elements. In fact, with the exception of matches between atoms, the other possibilities only result in recursive calls which must eventually end up here. The obvious design is for success to depend on being able to recursively match corresponding cars and cdrs, but this is complicated by the presence of operator prefixes at various levels, and especially by the use of spanning. We must also take into account the fact that some operators (i.e. = and !) do not require that the match continue into their arguments.

Smatch is used to set up the required series of recursive calls, but is able to oversee the process of applying operators and spanning. It is realized as a nest of applications which strips off and stacks bind, span and modified operators when they are present in the cars of each pattern, and creates an environment for further processing.

The spanning mechanism, when required, is set up at the center of this nest before processing continues and the operators are applied. The span is detected during the transfer of modified operators to holding stacks and communicated by binding the atom "spanner" to variable TSP1 or TSP2 (otherwise these variables are bound to "nospanner"). The span starts with a single argument and backtracks until the rest of the match succeeds, or the arguments are exhausted, thus returning the smallest possible span. No prefixes are removed from spanned arguments, but they can eventually be applied since modified operators from the string in which ! appears are applied argument by argument or to the entire spanned list, as indicated.

A special case is encountered when simultaneous spanning is called for. We must either ignore both spans or allow the prefixes to be interpreted from the elements containing the spans. We have chosen the latter, but problems of consistency arise, for example how to interpret a span-prefixed element that appears inside a spanned list. In our examples we have not come upon any "double spanning" but this clearly deserves further consideration.

The spanning control lies in the functions span and dblspan. The match process continues in ematch where any called for binding takes place (operator application takes place in the call to ematch using applyops for the stacked modified operators and transform for the others). If a bottom match has not yet been reached, then the function digon will carry the match into the cars of each pattern. Otherwise moveon will continue the match directly into the cdrs. Success occurs when and if both cdrs are simultaneously NIL.

Except for the binding and spanning operators, the prefixes are defined directly as

LISP functions so that they may themselves be applied (some special treatment of the extension prefixes is however required). Modified prefixes are defined to be their unmodified counterparts, since the modification is detected in the matching process. The details of these functions implement the descriptions given in the previous section.

Additional descriptions of the particular LISP functions used to implement the pattern matcher appear in the appendix as comments. Amongst the more noteworthy programming tools is ncond, a variation on cond, which allows the user to access the result of the test expression in the result expression as the value of the atom DITTO.

5. EXAMPLES

(A) *A continuous example*

We now describe an example of modeling a continuous simulation in the Hendrix system using CONCUR. The world is a room, containing a robot called Robby, a bucket, a water tap, and the tap's valve. There are six locations in the room: A, B, C, D, E, and F. Robby is at B, the bucket is at C, the tap is at D, and the valve is at E. The world is shown in Fig. 1, and the entire world model is given in Fig. 2. The commands that are required to be in the SWM for Robby to fill a bucket of water are in Fig. 3.

What follows is the set of scenarios for modeling this activity. We can introduce a useful mnemonic for reading the scenarios by substituting "some" or "a" for = and "that" or "the" for *. Thus (=robot turn =v at =r) is read "some robot turn some valve at some rate". We now consider each of the scenarios.

SCENARIO 1: Robot turns on valve

```
.1  ((&
.2   (=robot turn =v at =r)
.3   (*robot at =loc)
.4   (*v at *loc)
.5   *(0 < r)
.6   *(r ⩽ (maxrate *v)))
          ≡
.7   (block
.8   (≠ (*robot turn *v at *r))
.9   ((*v rate) ← *r)))
```

Referring to Fig. 3, robot, v, and r would be bound to Robby, vlv and .324 respectively.

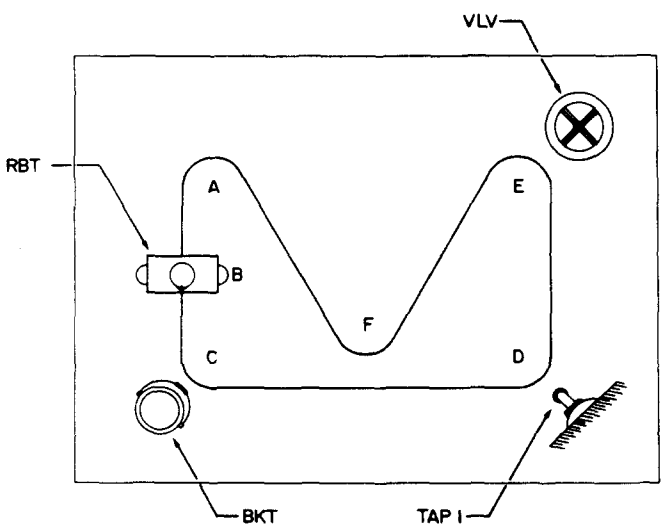

Fig. 1. Robby's World (reprinted with permission from Ref. [2]).

As the SWM places Robby at B and vlv at E, this scenario would fail to initiate at this time, since (1.4) would evaluate to NIL. The scenario can be read as follows:

"Some robot turns on some valve at some rate
That robot is at some location
That valve is at that location
That rate is less than the maximum of that valve
≡
delete command
that valve rate becomes that rate."

```
((A is a location) = TRUE)    ((path A to B) = 10.0)
((B is a location) = TRUE)    ((path B to C) = 10.0)
((C is a location) = TRUE)    ((path C to D) = 40.0)
((D is a location) = TRUE)    ((path D to E) = 20.0)
((E is a location) = TRUE)    ((path E to F) = 35.0)
((F is a location) = TRUE)    ((path F to A) = 35.0)

((Robby is a robot) = TRUE)     ((tap1 is a tap) = TRUE)
((Robby at B) = TRUE)           ((tap1 immovable) = TRUE)
((Robby movable) = TRUE)        ((tap1 at D) = TRUE)
((speed limit Robby) = 25.0)    ((graspable tap1) = TRUE)

((bkt is a bucket) = TRUE)    ((vlv is a valve) = TRUE)
((bkt movable) = TRUE)        ((vlv immovable) = TRUE)
((capacity bkt) = 100.0)      ((vlv controls tap1) = TRUE)
((bkt at C) = TRUE)           ((vlv rate) = 0.0)
((graspable bkt) = TRUE)      ((vlv at E) = TRUE)
((content bkt) = 0.0)         ((graspable vlv) = TRUE)
                              ((maxrate vlv) = 17.5)
```

Fig. 2. The State of the World model.

```
((Robby to to C at 5.0) = TRUE)
((Robby grasp bkt) = TRUE)
((Robby go to D at 5.0) = TRUE)
((Robby release bkt) = TRUE)
((Robby go to E at 5.0) = TRUE)
((Robby turn vlv at .324) = TRUE)
((Robby go to D at 5.0) = TRUE)
((Robby grasp bkt) = TRUE)
((Robby go to B at 5.0) = TRUE)
```

Fig. 3. Commands required for Robby to fill the bucket.

SCENARIO 2: Robot grasps object

```
.1  ((&
.2   (=robot grasp =obj)
.3   (graspable *obj)
.4   (¬ (*robot grasping *obj))
.5   (*robot at =loc)
.6   (*obj at *loc))
          ≡
.7  ((*robot grasp *obj) → (*robot grasping *obj)))
```

2.1–2.6 "Some robot (is to) grasp some object
That object is graspable
That robot is at some location
That object is at that location

2.7: As in (1) of Section 2, we accomplish insertion and deletion by replacing the left-hand side of the deleted command.

SCENARIO 3: Robot grasping immovable object is immovable

```
.1  ((&
.2   (=robot grasping =obj)
.3   (*robot movable)
.4   (*obj immovable))
          ≡
.5   ((*robot movable) → (*robot immovable)))
```

3.1–3.4 "Some robot (is) grasping some object
That robot (is) movable
That object is immovable"

3.5 Insert and delete as before.

Note that (Robby grasping bkt) was added to the SWM by Scenario 2, hence Scenario 3 could only be initiated following Scenario 2's termination.

SCENARIO 4: Robot releases object

```
.1  ((&
.2   (=robot releases =obj)
.3   (*robot grasping *obj))
          ≡
.4   (block
.5   (≢ (*robot releases *obj))
.6   (≢ (*robot grasping *obj))))
```

4.1–4.5: "Some robot releases some object
That robot (is) grasping that object"

4.4–4.6: Delete release request (since release occurs instantaneously) and "grasping state" from the world model.

Once again we see that in this SWM Scenario 2 will precede Scenario 4.

SCENARIO 5: Robot becomes movable

```
.1  ((&
.2   (=robot immovable)
.3   (*robot is a robot)
.4   (¯(&(*robot grasping =obj)
          (*obj immovable))))
          ≡
.5   ((*robot immovable) → (*robot movable)))
```

5.1–5.3: "Some robot is immovable
That robot is a robot"

5.4: "It is not the case that the robot is grasping some object and that object is immovable"

5.5: robot becomes movable

The "robot grasp" and "robot release" events manipulate the assertion (robot grasping obj). The events concerning the robot's movability are separate. The two groups of events communicated only through the "grasping" assertion. This design has two advantages:

1. "Grasping" assertions could have other consequences, which would be implemented as other events. Changing the consequences is simply adding or removing events, not changing existing events.

2. Each group of events deals with only a small portion of the world model. The events are less susceptible to changes in the world model.

SCENARIO 6: Robot moves between connected points along directed path

```
.1  ((&
.2   (=robot go to =to at =speed)
```

.3 (*robot movable)
.4 (*robot at =from)
.5 *(0 < speed)
.6 *(speed ⩽ (speed limit *robot))
.7 (path *from to *to))
≡
.8 (block
.9 (≠ (*robot go to *to at *speed))
.10 ((*robot at *from) → (*robot moving from *from to *to))
.11 (resume after ((path *from to *to)/speed))
.12 ((*robot moving from *from to *to) → (*robot at *to))))

6.1–6.4: "Some robot is to go to some to-location at some speed
That robot is movable
That robot is at some from-location"
6.5–6.6: "speed between 0 and maximum for robot"
6.7: "There is a path from the from-location to the to-location"
6.9: Delete the command
6.10: robot in motion; exact location cannot be determined
6.11: event persists for as long as it takes for the robot to travel
6.12: robot has arrived.

Note that this scenario can be altered to allow movement along an undirected path by replacing (6.7) with

.7a (∨ (path *from to =$\hat{\$}$(*to ≡ *to)$\hat{\$}$ y)
.7b (path *to to =$\hat{\$}$(*from ≡ *from)$\hat{\$}$ y))
.7c (path =x to *y)

and (6.11) by

.11a (resume after ((path *x to *y)/speed)).

SCENARIO 7: Grasped object moves with robot

.1 ((&
.2 (=robot moving =x to =y)
.3 (*robot grasping =object))
≡
.4 (block
.5 ((*object at ??) → (*object moving from *x to *y))
.6 (resume after infinity unless (*robot moving from *x to *y))
.7 ((*object moving from *x to *y) → (*object at *y))))

7.7–7.3: "Some robot (is) moving from some x to some y
That robot is grasping some object"
7.5: "That object is put into a state of motion."
7.6: Continuation condition terminates when robot arrives

(See Scenario 6).
7.7: Object has arrived.

This event is an "object watcher".

SCENARIO 8: Robot moves between two points

.1 ((&
.2 (=robot go to =x at =speed)
.3 (*robot at =y)
.4 (¬ (path *y to *x))

```
.5  (path *y to =z))
          ≡
.6  (:(*robot go to *z at *speed ≡ TRUE))
```

8.1–8.3: "Some robot go to some x at some speed
That robot (is) at some y"

8.4: "No path from that y to that x"

8.5: "But there is a path from that y to some z"

8.6: This is not a recursive call, but rather will bring about the initiation of Scenario 6 (assuming all other initiation conditions are satisfied). The original command remains; it will not involve this scenario while the robot is in motion to the neighboring point, but will again become viable once the robot has arrived, at a later point in model time.

SCENARIO 9: Filling the bucket

```
.1  ((&
.2   (=bucket is a bucket)
.3   (*bucket at =loc)
.4   (=tap at *loc)
.5   (*tap is a tap)
.6   (=valve controls *tap)
.7   *(0 < (*valve rate))
.8   *((content *bucket) < (capacity *bucket)))
          ≡
.9  (block
.10  (:(water flowing from *tap to *bucket) ≡ TRUE)
.11  ((content *bucket) ←| *(*(content *bucket) + (time − *time)
                 ×(*valve rate)))
.12  (resume after ((capacity *bucket)
                        −(content *bucket)/(*valve rate))
            unless (& (*bucket at *loc)
                        | *((*valve rate) ← *(valve rate))))
.13  (≠ (water flowing from *tap to *bucket))
.14  ((content *bucket) ← *(content *bucket))))
```

9.1–9.6: "Some bucket is a bucket
That bucket is at some location
Some tap is at that location
That tap is a tap
Some valve controls that tap"

9.7–9.8: Valve is on; bucket is not full.

9.10: Tell world water is flowing.

9.11: The filling of the bucket represents a continuous process and is therefore represented by a formula.

9.12: The continuation condition is also given by a formula.

Note that the time dependence of this formula is derived from (content *bucket), which was itself made time-dependent in (9.11).

NOTE: Scenario 6 could have been written similarly so that the robot's location could always be determined by evaluating an equation; however, we choose for simplicity to "break contact" with the robot whenever it is between two designated points (i.e. the "robot moving..." command). A model was presented in Section 2 in which the location of the robot can be determined at any point in model time.

The events as written permit several robots, buckets, taps and valves. Coördination may require additional events be added to the world model.

(B) *A discrete example*

As an additional example, we wish to describe the implementation of semaphores [10]. We create a type "semaphore" and indicate a variable of that type by a world model relation

((sema is a semaphore) ≡ TRUE).

We also have the following two scenarios:

```
((&                                  ((&
 (∨ =sem)                             (P =sem)
 (*sem is a semaphore))               (*sem is a semaphore)
      ≡                               *(0 < *sem))
 (block                                    ≡
 (*sem ← *(*sem + 1))                 (block
 (≢ (∨ *sem))))                       (*sem ← *(*sem −1))
                                      (≢ (P *sem)))).
```

We can initialize as many semaphores as we like, at any time, by introducing into the world model relations of the form,

```
.
.
(sema ≡ 1)
((sema is a semaphore) ≡ TRUE)
(semb ≡ 1)
((semb is a semaphore) ≡ TRUE)
.
.
```

Finally we must require that all scenarios placing P or V calls into the world model have the following form:

```
.
.
(:(X somesemaphore) ≡ TRUE)
(resume unless (X somesemaphore))
.
.
```

(Where X represents P or ∨).

6. CONCLUSIONS

Let us first look at CONCUR as the language of a world modeling system. As a direct descendent of Hendrix's system, CONCUR implements its most important ideas. The Hendrix model, like its predecessors, divides the world into knowledge and dynamic processes—the "world model" and the events. CONCUR models both (as the SWM and event scenarios) as a part of a more general world model, the association list. The usual concept of "world model" is directly implemented as part of the syntax.

Another syntactical feature is the use of logical formulas, which make Hendrix's event characteristics—initiation and continuation conditions—easily expressible as conditions on the world model. Hendrix proposed two types of initiation conditions. Symbolic initiation conditions give assertions which must be present in the world, while numeric initiation conditions are Boolean expressions which must evaluate to TRUE. Both of these appear in CONCUR as instances of a single entity, the pattern, with the difference being that so-called numerical conditions are evaluated before matching. A single initiation condition pattern, the conjunction of each individual condition, is all that is required.

Events are invoked as demons of the world model. The various components of the initiation condition together become a "call" to the event when they are all present in the SWM. Events do not explicitly invoke each other. The result of an active event is changes in the world model, which can occur gradually over time, and includes possible changes in other scenarios since these are world model elements. Such "meta-events" are a natural step towards evolutionary world models, but are syntactically equivalent to the scenarios they manipulate, and can themselves be subject to manipulation by similar entities.

Event continuation can be governed by expressions similar to those which act to initiate them, and an event can, throughout its duration, model gradual continuous change. An invoked event can remain suspended over time and have various parameters appear to change gradually to the rest of the world. Event duration can be specified by a fixed amount of time, or by continuation conditions which are patterns which allow the event to continue so long as they are able to match the world model.

An event can affect the world model only through use of four functions: insert ($\equiv$), delete ($\not\equiv$), transform ($\rightarrow$), and assign ($\leftarrow$). Hendrix presents three categories of such effects, initial, gradual, and final, which alter the world at various stages in the event's lifetime. Initial and final effects are realized in CONCUR as expressions appearing before and after, respectively, the resume statement, while gradual effects are realized by formulas.

An important advantage gained by using the CONCUR structure as the basis for a simulation system is the ability to produce different simulation techniques through the careful design of event scenarios and slight variations in the overall control structure. The latter consists only of a function which "patrols" the SWM attempting matches against initiation conditions. User interface is implemented with the operator console event, and can therefore be controlled by that event's initiation condition.

Thus to obtain a clock pulse simulation using the significant event method [11], we model the clock as an SWM variable and have it updated by the patrol function following each scan. The user may be contacted at each point in time by having an operator initiation condition of TRUE; or, more realistically, user interaction may be required only at specific points in time or as a result of the state of the model, and as we have seen such symbolic and/or numerical complexities can be naturally represented as initiation conditions. A quiescent period [11] for a variable in such a simulation corresponds to the (invisible) use of a formula to represent its value. It need not be parameterized by a time interval alone but may rely also on symbolic conditions to determine its duration. This allows for a stricter concept of "significance" and a less complex setting for introducing probabilistic models. Both quiescent and active periods are controlled by the same type of entity—event scenarios—with the latter corresponding to those events which are instantaneous (i.e. not possessing a resume), and as such no further distinction is made.

Movement of the system in time is modeled by scenarios executing time dependent resume statements; model activity at any given moment can in fact be measured by the scenarios that are currently suspended by resume calls. A sophisticated scanning function can therefore employ a "next event" technique in advancing the model (as long as the operator event is taken into consideration) since a world model state can only change at those times when events can be initiated or become active. In addition it is also possible to utilize event scheduling [12] since events have the power to modify each other.

The flexibility available for simulation is due entirely to an expanded notion of the event concept and CONCUR's ability to deal with this generality. It is therefore reasonable to consider utilizing CONCUR to expand this concept beyond the realm of simulation and into the more common area of programming structures. In particular it provides a natural setting in which the various multiprogramming structures can be studied.

CONCUR may actually turn out to be more powerful as a general modeling tool than it may appear from the robot simulation examples. For example, implementing the natural parallelism inherent in the simulation would actually make possible the modeling of the various constructs used in concurrency (such as semaphores [10], Hoare moni-

tors [13], serializers [14], fork and join [15], and communicating sequential processes [9]) as CONCUR scenarios and thus produce a unified setting for the study of these systems. An example such as the consumer-producer problem [10] can easily be visualized in the CONCUR framework. In fact, since interactions with the user can be modeled as the operator console event, and all programming can be interfaced through an invocation of this event (or metaevent, which is the same thing), all of the power of continuous change and event duration is thereby available to the user.

The goal of CONCUR was to be a language expressing Hendrix's ideas about world modeling, but the result has been the development of a powerful syntax which is not a world modeling system, but rather a system in which world modeling can be done. It is in fact three different languages at once: (1) a pattern match interpretative system which erases many of the distinctions between variable and function which appear in LISP; (2) a continuous data-driven event initiator which can be used in realistic simulations; and (3) a general modeling tool for continuous and discrete concurrent processing systems. It may be that in this last area, in which the concept of event appears in its most interesting and exciting guise, that the full impact of CONCUR will be realized.

REFERENCES

1. G. Hendrix, Modeling simultaneous actions and continuous processes, *Artif. Intell.* **4**, 145–180 (1973).
2. J. Lowrance and D. P. Friedman, Hendrix's model for simultaneous actions and continuous processes: an introduction and implementation, *Int. J. Man–Mach. Stud.* **9**, 537–581 (1977).
3. D. V. McDermott and G. J. Sussman, The CONNIVER reference manual, A. I. Memo 295a, MIT A.I. Lab, Cambridge (1974).
4. C. Hewitt, Procedural embedding of knowledge in PLANNER, *IJCAI* 167–182 (1977).
5. R. Davis, B. Buchanan and E. Shortliffe, Production rules as a representation for a knowledge-based consultation program, *Artif. Intell.* **8**, 15–45 (1977).
6. T. Brennan and D. Friedman, CONCUR: A language for continuous concurrent processes, (preliminary report) Indiana University (1977).
7. C. Hewitt, Viewing control structures as patterns of passing messages, *Artif. Intell.* **8**, 323–364 (1977).
8. J. Vuillemin, Correct and optimal implementation of recursion in a simple programming language, *J. Comput. Syst. Sci.* **9**, 332–354 (1974).
9. C. A. R. Hoare, Communicating sequential processes, *CACM* **21**, 666–677 (1978).
10. E. W. Dijkstra, Co-operating sequential processes, *Programming Languages* (Edited by F. Genuys). Academic Press, New York (1968).
11. A. F. Babich, J. Grason and D. L. Parnas, Significant event simulation, *CACM* **18**, 323–329 (1975).
12. G. S. Fishman, *Concepts and Methods in Discrete Event Simulation*. Wiley, New York (1973).
13. W. H. Kaubisch, R. H. Perrott and C. A. R. Hoare, Quasiparallel programming, *Software—Practice and Experience* **6**, 341–356 (1976).
14. R. Atkinson and C. Hewitt, Synchronization in actor systems, *Fourth ACM Symposium on Principles of Programming Languages*, pp. 267–280 (1977).
15. J. B. Dennis and E. C. Van Horn, Programming semantics for multi-programmed computations, *CACM* **9**, 143–155 (1966).
16. R. Fikes and N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artif. Intell.* **2**, 189–208 (1971).
17. W. V. O. Quine, *Word and Object*. MIT Press, Cambridge, MA (1960).

APPENDIX I

Summary of Notation

Symbol	*Name*	*Action*
*	eval	Evaluates the pattern prefixed
*̂	Mod. eval	Evaluates the pattern matched
:	unlist	Unlists the pattern prefixed
:̂	Mod. unlist	Unlists the pattern matched
$	extend	Creates an extended operator that manipulates the value of the pattern prefixed
$̂	Mod. extend	Creates an extended operator that manipulates the value of the pattern matched
!	span	Spans a sublist of the pattern matched
=	bind	Binds the value of the prefixed pattern to the value of the matched pattern
?	universal	Indicates universal consistent match with prefixed atom
|	unspecial	Indicates that the prefixed operator is to remain in the unlisted pattern
??	free	An atom that matches any atomic pattern
'	quote	Quotes the prefixed pattern
≡	insert	Adds an identifier-value pair to the association list

≠	delete	Removes an identifier-value pair from the association list
→	transform	Transforms one identifier into another identifier
←	assign	Transforms one value into another. Is the assignment operator, usually notated :=
resume	resume	Resumes computation of the event at the specified time or condition.

APPENDIX II

CONCUR PATTERN MATCHER

1. PRINCIPAL FUNCTIONS

```
(DEFPROP MATCH
 (LAMBDA (P1 P2) (PMATCH P1 P2 (LIST WM NIL)))
EXPR)

(DEFPROP WORLDMATCH
 (LAMBDA (P) (MATCH P (CONS @V (CAR WM))))
EXPR)

(DEFPROP PMATCH
 (LAMBDA (P1 P2 R)
  (NCOND ((OR (NULL P1) (NULL P2)) NIL)
         ((IFTYPE (LAMBDA (P) (ATOM P))) (RESULT ALMATCH))
         ((IFTYPE (LAMBDA (P) (EQ (CAR P) @&))) (RESULT ANDM))
         ((IFTYPE (LAMBDA (P) (EQ (CAR P) @V))) (RESULT ORM))
         ((IFTYPE (LAMBDA (P) (EQ (CAR P) @\))) (RESULT NOTM))
         ((IFTYPE (LAMBDA (P) (OP (CAR P)))) (RESULT ELM))
         (T (SMATCH (CAR P1) (CAR P2) (CDR P1) (CDR P2) R))))
EXPR)

(DEFPROP SMATCH
 (LAMBDA (P1 P2 C1 C2 R)
  (APPLY
   (FUNCTION
    (LAMBDA (BP1 BS1 CO1)
     (APPLY
      (FUNCTION
       (LAMBDA (BP2 BS2 CO2)
        (APPLY
         (FUNCTION
          (LAMBDA (TP1 TS2 TSP1)
           (APPLY (FUNCTION (LAMBDA (TP2 TS1 TSP2) (SPAN? TSP1 TSP2)))
                  (TRANSFER BP2))))
         (TRANSFER BP1))))
      (BIND? P2 CO1))))
   (BIND? P1 @DIGON)))
EXPR)

(DEFPROP EMATCH
 (LAMBDA (EP1 EP2 CONT R)
  (COND
   ((OR (NULL EP1) (NULL EP2)) NIL)
   (T
    ((LAMBDA (R1) (COND ((NULL R1) NIL) (T (CONT R1))))
     (BS2 EP2 EP1 (BS1 EP1 EP2 R)))))))
EXPR)
```

2. SPAN CONTROL

```
(DEFPROP SPAN?
 (LAMBDA (T1 T2)
  (COND
   ((AND (EQ T1 @SPANNER) (EQ T2 @SPANNER))
    (DBLSPAN (NCONS TP1) (NCONS TP2) C1 C2))
   ((EQ T1 @SPANNER) (SPAN P2 (TRANSFORM TP1 R) C2 C1 TS2 BS2 BS1 R))
   ((EQ T2 @SPANNER) (SPAN P1 (TRANSFORM TP2 R) C1 C2 TS1 BS1 BS2 R))
   (T
    (EMATCH (APPLYOPS (TRANSFORM TP1 R) TS1)
            (APPLYOPS (TRANSFORM TP2 R) TS2)
            CO2
            R))))
EXPR)

(DEFPROP SPAN
 (LAMBDA (P1 P2 C1 C2 TS1 BS1 BS2 R)
  ((LABEL
    SPAN1
    (LAMBDA (PP1 C1)
     (COND
      (((LAMBDA (PP)
         (AND2 (CONSISTENT (LIST @! P2) PP R)
               (EMATCH PP P2 @MOVEON DITTO)))
        (APPLYOPS PP1 TS1)))
      ((NULL C1) NIL)
      (T (SPAN1 (SNOC (CAR C1) PP1) (CDR C1))))))
   (NCONS P1)
   C1))
EXPR)

(DEFPROP DBLSPAN
 (LAMBDA (P1 P2 C1 C2)
  (COND
   ((AND2 (CONSISTENT (LIST @! P1) P2 R)
          (SPAN P2 (APPLYOPS P1 TS1) C2 C1 TS2 BS2 BS1 DITTO)))
   ((NULL C1) NIL)
   (T (DBLSPAN (SNOC (CAR C1) P1) P2 (CDR C1) C2))))
EXPR)
```

3. FULL PATTERN PROCESSORS

```
(DEFPROP ALMATCH
 (LAMBDA (P1 P2 R)
  (COND
   ((ATOM P2) (COND ((OR (EQ P2 @??) (EQ P1 P2)) R)))
   ((EQ (CAR P2) @?)
    ((LAMBDA (P)
      (COND ((ATOM P) (CONSISTENT P2 P1 R))
            ((OP (CAR P)) (SMATCH P1 P2 NIL NIL R))))
     (POP P2)))
   ((EQ (CAR P2) @&) (ANDM P2 P1 R))
   ((EQ (CAR P2) @V) (ORM P2 P1 R))
   ((EQ (CAR P2) @\) (NOTM P2 P1 R))
   ((OP (CAR P2)) (SMATCH P1 P2 NIL NIL R))))
EXPR)
```

```
(DEFPROP ANDM
 (LAMBDA (P1 P2 R)
  ((LABEL
    ANDMATCH
    (LAMBDA (P1 P2 R)
     (COND
      ((NULL R) NIL)
      ((NULL P1) R)
      (T
       ((LABEL
         ANDOR
         (LAMBDA (P3)
          (COND ((NULL P3) NIL)
                ((ANDMATCH (CDR P1) P2 (PMATCH (CAR P1) (CAR P3) R)))
                (T (ANDOR (CDR P3))))))
        P2)))))
   (CDR P1)
   (COND ((OR (ATOM P2) (NEQ (CAR P2) @V)) (LIST P2))
         (T (CDR P2)))
   R))
EXPR)

(DEFPROP ORM
 (LAMBDA (P1 P2 R)
  ((LABEL
    ORMATCH
    (LAMBDA (P1 P2)
     (COND ((NULL P1) NIL)
           ((PMATCH (CAR P1) P2 R))
           (T (ORMATCH (CDR P1) P2 R)))))
   (CDR P1)
   P2))
EXPR)

(DEFPROP ELM
 (LAMBDA (P1 P2 R) (SMATCH P1 P2 NIL NIL R))
EXPR)


(DEFPROP NOTM
 (LAMBDA (P1 P2 R) (COND ((NOT (PMATCH (CADR P1) P2 R)))
EXPR)
```

4. S-PATTERN PROCESSORS

```
(DEFPROP BIND?
 (LAMBDA (P C)
  (COND
   ((AND (NOT (ATOM P)) (EQ (CAR P) @=)) (LIST (POP P) @BIND @MOVEON))
   (T (LIST P @NOBIND C))))
EXPR)

(DEFPROP DIGON
 (LAMBDA (R) (NCOND ((PMATCH EP1 EP2 R) (MOVEON DITTO))))
EXPR)

(DEFPROP MOVEON
 (LAMBDA (R)
  (COND ((AND (NULL C1) (NULL C2)) R) (T (PMATCH C1 C2 R))))
EXPR)
```

```
(DEFPROP TRANSFORM
 (LAMBDA (P R)
  (COND
   ((ATOM P) P)
   ((EQ (CAR P) @//)
    (CONS (CAR P) (PUSH (CADR P) (TRANSFORM (POP (CDR P)) R))))
   ((EQ (CAR P) @$)
    (APPLY (CAR P)
           (LIST (CADR P) (TRANSFORM (POP (CDDR P)) R) (CAR R))))
   ((PROP (CAR P))
    (APPLY (CAR P) (LIST (TRANSFORM (POP P) R) (CAR R))))
   ((CMOP (CAR P)) NIL)
   (T P)))
EXPR)

(DEFPROP APPLYOPS
 (LAMBDA (P S)
  ((LABEL
    APP1
    (LAMBDA (P1 S1)
     (COND
      ((NULL S1) P1)
      ((EQ (CAR S1) @$^)
       (APP1 (APPLY (CAR S1) (LIST (CADR S1) P1 (CAR R))) (CDDDR S1)))
      (T (APP1 (APPLY (CAR S1) (LIST P1 (CAR R))) (CDR S1))))))
   ((LABEL
     APP2
     (LAMBDA (P2 S2)
      (COND
       ((NULL S2) P2)
       ((EQ (CAR S2) @$^)
        (APP2
         (MAPCAR
          @(LAMBDA (P) (APPLY (CAR S2) (LIST (CADR S2) P (CAR R))))
          P2)
         (CDDDR S2)))
       (T
        (APP2
         (MAPCAR @(LAMBDA (P) (APPLY (CAR S2) (LIST P (CAR R)))) P2)
         (CDR S2))))))
    P
    (CAR S))
   (CDR S)))
EXPR)

(DEFPROP TRANSFER
 (LAMBDA (P)
  ((LABEL
    TRANS
    (LAMBDA (P1 S)
     (COND
      ((OR (ATOM P1) (NOT (OP (CAR P1))) (PROP (CAR P1)))
       (LIST P1 (CONS NIL S) @NOSPANNER))
      ((EQ (CAR P1) @!)
       (APPLY
        (FUNCTION
         (LAMBDA (P2 S2 SP) (LIST P2 (CONS (CDR S2) S) @SPANNER)))
        (TRANS (POP P1) NIL)))
      ((EQ (CAR P1) @$^)
       (TRANS (POP (CDDR P1))
              (APPEND (LIST (CAR P1) (CADR P1) (CADDR P1)) S)))
      (T (TRANS (POP P1) (CONS (CAR P1) S))))))
   P
   NIL))
EXPR)
```

5. PATTERN OPERATORS

```
(DEFPROP BIND
 (LAMBDA (P1 P2 R)
  (COND ((NULL R) NIL) (T (CONS (RECORD P1 P2 (CAR R )) (CDR R)))))
EXPR)

(DEFPROP NOBIND
 (LAMBDA (P1 P2 R) R)
EXPR)

(DEFPROP *
 (LAMBDA (P R)
  (NCOND
   ((ATOM P) (NCOND ((WORLDEVAL P R) (APPLY @: DITTO)) (T (EVAL P))))
   ((EQ (CAR P) @//) (PUSH @// (IGNORE (CADR P) (* (POP (CDR P)) R))))
   ((PROP (CAR P)) (* (APPLY (CAR P) (LIST (POP P) R)) R))
   ((CMOP (CAR P)) (IGNORE (CAR P) (* (POP P) R)))
   ((EQ (CAR P) @QUOTE) (CADR P))
   ((WORLDEVAL P R) (APPLY @: DITTO))
   ((GET (CAR P) @EXPR)
    (APPLY (CAR P) (MAPCAR @(LAMBDA (P1) (* P1 R)) (CDR P))))
   ((NULL (CDR P)) (CAR P))
   (T NIL)))
EXPR)

(DEFPROP :
 (LAMBDA (P R)
  (COND ((NULL R) NIL)
        ((ATOM P) P)
        ((EQ (CAR P) @QUOTE) (CADR P))
        ((EQ (CAR P) @//) (PUSH (CADR P) (: (POP (CDR P)) R)))
        ((PROP (CAR P)) (APPLY (CAR P) (LIST (: (POP P) R) R)))
        ((CMOP (CAR P)) (IGNORE (CAR P) (: (POP P) R)))
        (T (MAPCAR @(LAMBDA (P) (: P R)) P))))
EXPR)

(DEFPROP $
 (LAMBDA (E P R)
  (COND ((EQ (CAR E) @*) ($ (TRANSFORM E R) P R))
        (T (: (CADR E) (CAR (PMATCH (CAR E) P (NCONS R))))))))
EXPR)

(DEFPROP *^
 (LAMBDA (P R) (* P R))
EXPR)

(DEFPROP :^
 (LAMBDA (P R) (: P R))
EXPR)

(DEFPROP $^
 (LAMBDA (E P R) ($ E P R))
EXPR)

(DEFPROP ?
 (LAMBDA (P R) (LIST @? P))
EXPR)
```

6. MATCH RECORD OPERATORS

```
(DEFPROP CONSISTENT
 (LAMBDA (P1 P2 R)
  (NCOND ((NULL R) NIL)
         ((ASSQ P1 (CDR R)) (COND ((EQUAL DITTO P2) R)))
         (T (CONS (CAR R) (RECORD P1 P2 (CDR R))))))
EXPR)

(DEFPROP RECORD
 (LAMBDA (P1 P2 R) (CONS (CONS P1 (CAR R)) (CONS P2 (CDR R))))
EXPR)

(DEFPROP ASSQ
 (LAMBDA (P R)
  ((LABEL
    ASSQ1
    (LAMBDA (R1 R2)
     (COND ((NULL R1) NIL)
           ((EQUAL (CAR R1) P) (CAR R2))
           (T (ASSQ1 (CDR R2) (CDR R2))))))
   (CAR R)
   (CDR R)))
EXPR)
```

7. DATA STRUCTURE OPERATORS

```
(DEFPROP PROP
 (LAMBDA (A) (MEMBER A @(* : ? $)))
EXPR)

(DEFPROP CMOP
 (LAMBDA (A) (MEMBER A @(*^ :^ ! = $^)))
EXPR)

(DEFPROP OP
 (LAMBDA (A) (OR (PROP A) (CMOP A)))
EXPR)

(DEFPROP PUSH
 (LAMBDA (A B)
  (COND ((OR (ATOM B) (NOT (OP (CAR B)))) (LIST A B)) (T (CONS A B))))
EXPR)

(DEFPROP POP
 (LAMBDA (P) (COND ((OP (CADR P)) (CDR P)) (T (CADR P)))
```

8. PROGRAMMING TOOLS

```
(DEFPROP LISTP
 (LAMBDA (A) (NOT (ATOM A)))
EXPR)

(DEFPROP SNOC
 (LAMBDA (A L) (APPEND L (LIST A)))
EXPR)
```

```
(DEFPROP NCOND
 (LAMBDA (L E)
  (COND
   ((NULL L) NIL)
   (T
    ((LAMBDA (DITTO)
      (COND (DITTO (EVAL (CADAR L)))
            (T (EVAL (CONS @NCOND (CDR L)) E))))
     (EVAL (CAAR L) E)))))
FEXPR)

(DEFPROP IFTYPE
 (LAMBDA (F E)
  (COND ((EVAL (LIST (CAR F) @P1) E) (LIST P1 P2 R))
        ((EVAL (LIST (CAR F) @P2) E) (LIST P2 P1 R))))
FEXPR)

(DEFPROP RESULT
 (LAMBDA (F E) (COND ((NULL DITTO) NIL) (T (APPLY (CAR F) DITTO E))))
FEXPR)

(DEFPROP AND2
 (LAMBDA (F E) (NCOND ((EVAL (CAR F) E) (EVAL (CADR F) E))))
FEXPR)

(DEFPROP WORLDEVAL
 (LAMBDA (P R)
  (COND
   (((LAMBDA (N) (COND ((NUMBERP N) (LIST N R)))) (TRANSFORM P R)))
   (T
    ((LABEL
      WORLD1
      (LAMBDA (R1)
       (NCOND
        ((NULL (CAR R1)) NIL)
        ((PMATCH P (CAAR R1) (LIST R NIL))
         (LIST (CADR R1) (CAR DITTO)))
        (T (WORLD1 (CONS (CDAR R1) (CDDR R1)))))))
     R))))
EXPR)

(DEFPROP IGNORE
 (LAMBDA (P1 P2)
  (COND ((OR (ATOM P2) (NOT (OP (CAR P2)))) (LIST P1 P2))
        (T (CONS P1 P2))))
EXPR)
```

About the Author—DANIEL PAUL FRIEDMAN received the B.S. degree in Mathematics in 1967 from the University of Houston and the M.A. and Ph.D. degrees in Computer Science from The University of Texas at Austin in 1969 and 1973 respectively. He is an Associate Professor of Computer Science at Indiana University at Bloomington. His research interests include applicative programming languages, languages and models for distributed computing, formal semantics, graph processing and AI programming languages.

About the Author—TERRENCE JAMES BRENNAN received the B.A. degree With Honors in Computer-Science in 1977 from Indiana University at Bloomington. He is a consultant for Northern Illinois University at Dekalb.

About the Author—RICHARD M. SALTER received the A.B. degree in Mathematics from Oberlin College in 1973 and the M.A. and Ph.D. degrees in Mathematics from Indiana University at Bloomington in 1975 and 1978 respectively. He is an Assistant Professor of Mathematics and Computer Science at Drexel University. His research interests include applicative programming languages, A.I. programming languages, multiprocessing, and continuous simulation.