CONCURRENT EVENT RECOGNITION

Technical Report CSE 93-09

Patrick Hill

Department of Computer Science and Engineering
Auburn University
Auburn University, AL 36849

August 4, 1993

Concurrent Event Recognition

Patrick Hill
Department of Computer Science and Engineering
Auburn University
Auburn University, AL 36849

Abstract: Event recognition in concurrent systems is more difficult due to the indeterminacy of program events. Program events are specified using data path expressions. An efficient method of concurrent event recognition is presented that uses Augmented Transition Networks to represent the language of data path expressions. The implementation of this method is discussed and examples are shown.

Keywords: concurrency, event recognition, data path expression, augmented transition network, predecessor automata, shuffle automata, specification language.

Concurrent Event Recognition

I. Statement of the Problem

Concurrent Event Recognition

Recognizing concurrent events is complicated by the fact that events can occur simultaneously. The order of events is known only after the events occur. To insure a concurrent system is performing as anticipated, and to monitor the progress of a concurrent system, it is useful to have a tool that accepts a specification for events and confirms that the events that actually occur are permissible events according to the specification.

Augmented transition networks, or ATNs, were devised to parse natural languages. ATNs are finite state machines (FSMs), with extensions. Whereas FSMs can only recognize regular languages, ATNs can recognize type 0 languages. ATNs are used to implement a concurrent event recognizer in this project.

Expressions for specifying the behavior of parallel programs are called *Data Path Expressions,* or DPEs. DPEs are regular expressions extended with concurrent operators in [Hseush and Kaiser, 90]. This project uses DPEs as a specification language that defines the acceptable program events. The program events are represented by strings in the language defined by the DPE. For example, given a sequential DPE: `(A + B); C*;D;E` determine if the string `ADE` is in the language.

II. Previous Work in This Area

DPE Operators

DPEs are an extension of path expressions described in Campbell and Habermann [73]. Path expressions are an extension of regular expressions. DPEs are expressions containing the usual regular expression operators for sequencing (;), exclusive selection (+), repetition (*), with the addition of new operators for concurrency (&) and concurrent closure (@). For example given two events **A** and **B** :

A ; B	means **A** precedes **B**.
A + B	means either **A** occurs or **B** occurs, but not both.
A*	means **A** occurs sequentially zero or more times.
A & B	means **A** and **B** occur concurrently.
A@	means **A** occurs concurrently zero or more times.

DPEs can be used to specify the behavior of program execution or program events, [Hseush and Kaiser, 90; Ponamgi, Hseush, and Kaiser, 91]. The operators describe the relationship between a program's events. Events can be grouped using parentheses, and operator precedence can be overridden using parentheses as in regular expressions. For example **A & B*** is not the same as **(A & B)***.

DPE Hierarchy

Using DPEs, Hseush and Kaiser [90] classify concurrent behavior into 5 subclasses according to the syntax of allowable expressions. The syntax constraints of each subclass is shown in the following table:

5 Subclasses of Concurrent Behavior

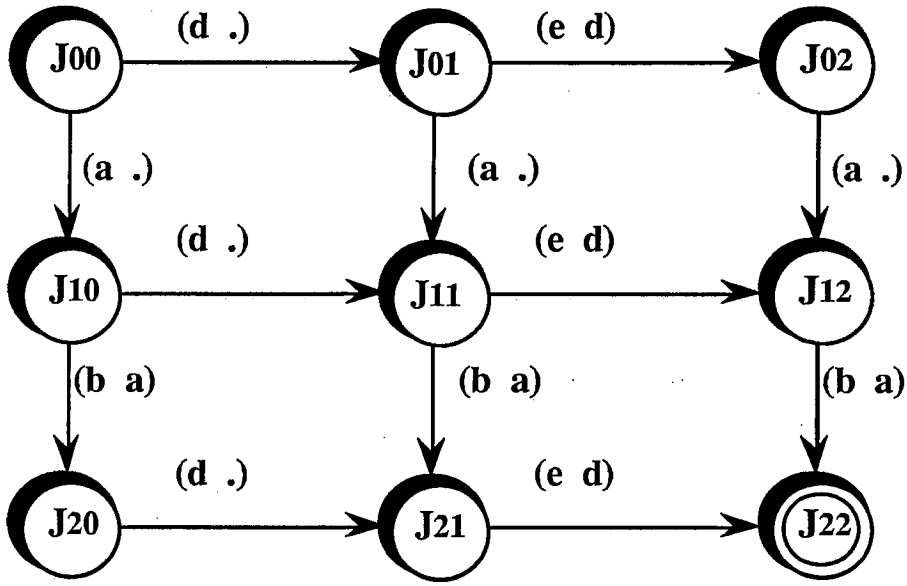
Description	Operators Allowed
Sequential Behavior	; + *
Limited Safe Concurrency	; + * **limited &**
Safe Concurrency	; + * &
Limited Unsafe Concurrency	; + * & **limited @**
Unsafe Concurrency	; + * & @

The first subclass expresses only sequential behavior. The second subclass expresses limited concurrency, in which process splitting (i.e. forking) is not permitted following a program branch, but program branching is permitted following a process split. The third subclass expresses general bounded parallelism. The fourth subclass permits unbounded parallelism, but without the ability to join an unknown number of threads. The fifth subclass describes general concurrency. This project is limited to DPEs that describe safe concurrent events. Therefore, the third subclass, safe concurrency, is the focus of this project.

Predecessor Automata

Hseush and Kaiser [90] introduce *predecessor automata*, or PAs, as an implementation model for concurrent event recognition. PAs are defined as FSMs, except for the transitions from one state to another. An FSM moves from one state to another state when an incoming event matches an event on an arc leaving the current state of the FSM. A PA moves from one state to another state when an incoming event combined with the set of events that causally precedes it matches an event and its predecessor set on an arc leaving the current state of the PA. The predecessor set contains only a trace of preceding events, not all possible events. As with FSMs, a string is recognized if there is no more input when the PA reaches a final state. Given a DPE without the concurrent closure operator @, a PA can be constructed to recognize strings in the language defined by the DPE. An example of a PA that represents the DPE `(D ; E) & (A ; B)` is shown below.

PAj: (D ; E) & (A ; B)



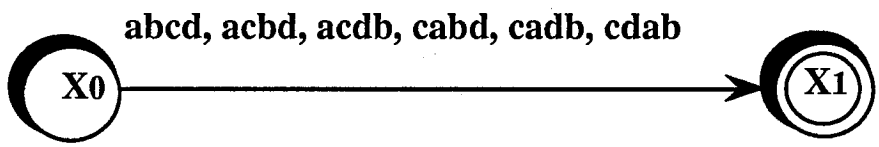
Shuffle Automata

Bates [88] uses *shuffle automata,* or SAs, to construct an event recognizer for distributed systems. SAs are also defined as FSMs, except for the transitions from one state to another. Where an FSM allows a single event, an SA processes sets of incoming events. An SA moves from one state to another state when a subset of the incoming event set matches a set of events on an arc leaving the current state. The events are *shuffled* until a set of events that matches the events on an arc is found; otherwise, if no match is found, the events are not accepted. The shuffle of the DPE `(A ; B) & (C ; D)` expression is:

```
A;B & C;D = abcd + acbd + acdb + cabd + cadb + cdab
```

A transition graph of an SA that represents this expression is shown below.

SAx: (A ; B) & (C ; D)



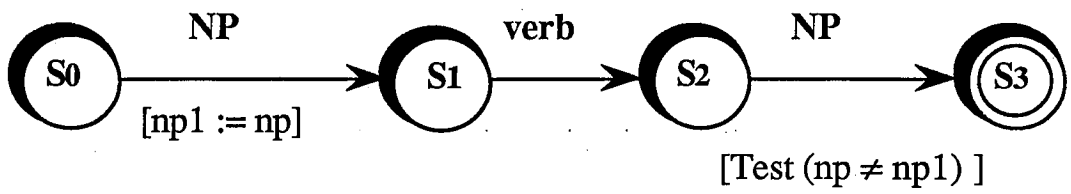
III. Augmented Transition Networks

From FSMs to ATNs

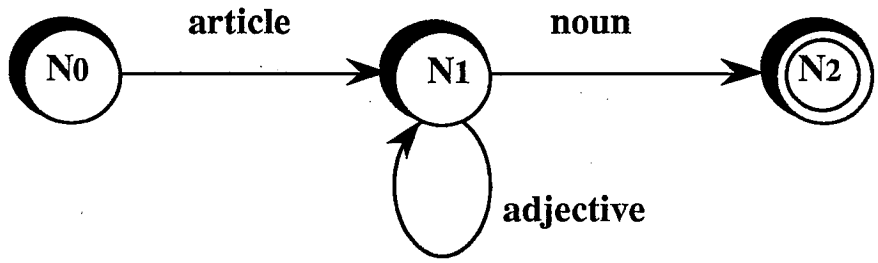
ATNs were developed by W. Woods in 1970 for parsing natural languages. ATNs are FSMs with extensions to increase their expressiveness. If we add recursion to non-deterministic FSMs we get recursive transition networks, or RTNs. In RTNs, the arcs between states can contain not only terminal symbols, but also non-terminals naming transition networks. This adds to the expressive power of the FSM. RTNs are equivalent in generative power to context-free grammars, but allow for more efficient parsing algorithms, according to Woods. [70] RTNs are sufficient to handle all sequential DPEs and some concurrent DPEs, but they become unwieldy for most DPEs that include the concurrency operator.

It is easy to extend RTNs by "augmentation" to allow context dependence. By adding the ability to have conditions on the arcs and a set of structure building actions to be executed if possible conditions are met, we have ATNs. An ATN can build a partial structural description of a sentence as it proceeds from state to state through the network. The pieces of this partial description can be held in registers. Flags or other indicators may also be held in these registers. These extensions make ATNs equivalent in power to Turing machines. ATNs contain the set of all RTNs which contain the set of all FSMs. Throughout the rest of this paper, only the term ATN will be used.

An example of an ATN is shown below:



NP:



The expression `[np1 := np]` assigns the result returned in the register `np` by the ATN `NP` to the register `np1`. The expression `[Test (np ≠ np1)]` tests that the contents of register `np` is not equal to the register `np1` after the ATN `NP` is visited a second time. The function of these registers and tests is to prevent the same noun phrase from being returned both times the ATN `NP` is visited.

The networks as transition graphs as shown above are used primarily as a descriptive device. The transitions from one state to another can just as well be described analytically in the form of a system of moves from state to state. The reliance on networks is useful only when the grammars are simple. For larger and more realistic grammars, the network representation is of little help, and one must rely on the more analytical definition of an ATN. [Woods, 70; Kreutzer and McKenzie, 90] The analytical representation of ATNs is usually represented as a procedure in a Lisp-like language. The examples in this paper are shown in the language Scheme. [Clinger and Rees, 91] The analytical representation of the first ATN in the previous example looks like this:

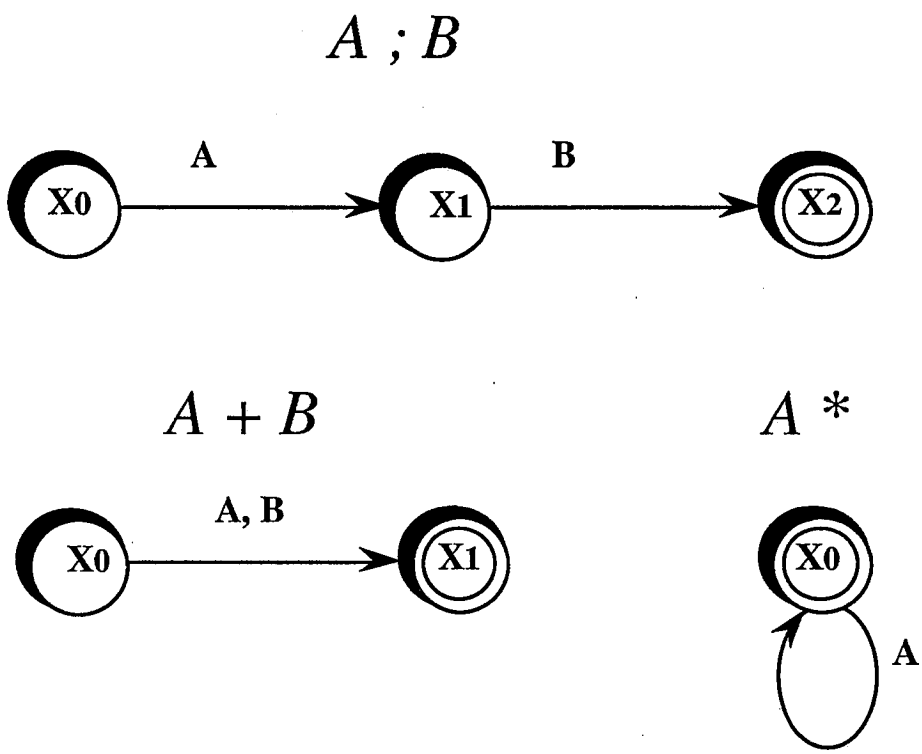
```
(define ExampleATN
  (MakeATN
    (STATE 'S0 (PUSH NP  #t (SETR 'np1 'np) (TO 'S1)))
    (STATE 'S1 (CAT (MakeCategory 'verb) #t (TO 'S2)))
    (STATE 'S2 (PUSH NP  #t                 (TO 'S3)))
    (STATE 'S3 (POP #t differentNP?))))
```

The state label is given after the `STATE` keyword, followed by the arcs that may be `PUSH` arcs that name another ATN, or `CAT` arcs that list categories following the `MakeCategory` keyword. Each arc has a test that may be `#t` to indicate that the arc is always valid. Next is the option to set a register variable with the `SETR` keyword. Then the label of the next state follows the `TO` keyword. The last state should always contain a `POP` arc that also has a validation test and may have a procedure that returns the value of the ATN. In the case the procedure named `differentNP?` returns the value of the ATN only if the contents of register `np` is different than the contents of register `np1`. If no procedure is needed, then `**` is used to return the value of the last word processed or the value returned from a `POP'`ed ATN. [Kreutzer and McKenzie, 90]

IV. Recognizing the Language of DPEs using ATNs

Implementing Sequential DPEs

We can use ATNs to represent the language of DPEs. The ATNs representing the expressions `A ; B`, `A + B`, and `A*` are shown below.



The corresponding analytic representation of these ATNs looks like this:

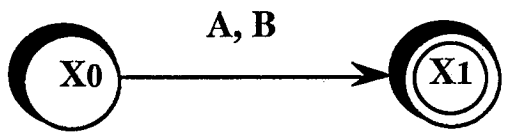
```
(define ATNseq    ;;;  A;B
  (MakeATN
   (STATE 'X0 (CAT (MakeCategory 'A) #t (TO 'X1)))
   (STATE 'X1 (CAT (MakeCategory 'B) #t (TO 'X2)))
   (STATE 'X2 (POP ** #t))))

(define ATN+      ;;;  A+B
  (MakeATN
   (STATE 'X0 (CAT (MakeCategory 'A 'B) #t (TO 'X1)))
   (STATE 'X1 (POP ** #t))))

(define ATN*      ;;;  A*
  (MakeATN
   (STATE 'X0 (CAT (MakeCategory 'A 'B) #t (TO 'X0))
              (POP ** #t))))
```

Representing Composite DPEs

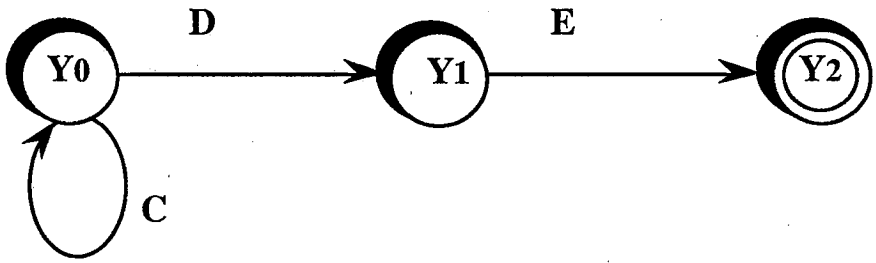
Being able to refer to an entire ATN by its name, rather than its constituent states makes it much easier to compose ATNs. It is this property that is the biggest advantage over FSMs when generating ATNs.

Suppose you have two ATNs. ATNx represents `A + B` and ATNy represents `C* ; D ; E`. The network representation of these two DPEs is shown in the figure below.

ATNx: A + B



ATNy: C ; D; E*



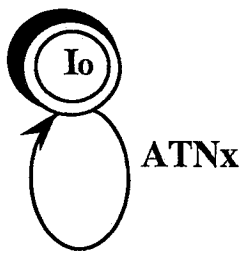
The analytic form of these two ATNs looks like something similar to this:

```
(define ATNx
  (MakeATN
   (STATE 'X0 (CAT (MakeCategory 'A 'B) #t (TO 'X1)))
   (STATE 'X1 (POP ** #t))))
```

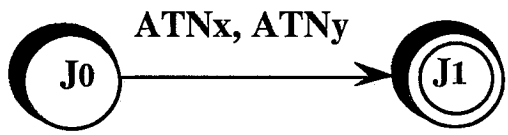
```
(define ATNy
  (MakeATN
   (STATE 'Y0 (CAT (MakeCategory 'C) #t (TO 'Y0))
              (CAT (MakeCategory 'D) #t (TO 'Y1)))
   (STATE 'Y1 (CAT (MakeCategory 'E) #t (TO 'Y2)))
   (STATE 'Y2 (POP ** #t))))
```

Now suppose we wanted to sequentially compose ATNx and ATNy to form the expression representing `(A+B); (C*; D; E)`. If we were using FSMs, it would be necessary to know all the final states of ATNx and connect them to all the start states of ATNy. (Imagine ATNx having 20 final states to appreciate the complexity of this task). But this is not necessary with ATNs (or RTNs), since arcs can contain complete ATNs. You can just create a new ATN that is the composition of the old ATNs. Forming the Kleene closure, or the selective composition is equally simple. The networks and their corresponding analytic representations are shown below.

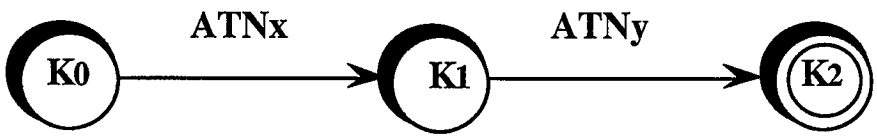
*ATNi: ATNx**



ATNj: ATNx+ ATNy



ATNk: ATNx ; ATNy



```
(define ATNi
  (MakeATN
   (STATE 'I0 (PUSH ATNx #t (TO 'I0))
              (POP ** #t))))
(define ATNj
  (MakeATN
   (STATE 'J0 (PUSH ATNx #t (TO 'J1))
              (PUSH ATNy #t (TO 'J1)))
   (STATE 'J1 (POP ** #t))))

(define ATNk
  (MakeATN
   (STATE 'K0 (PUSH ATNx #t (TO 'K1)))
   (STATE 'K1 (PUSH ATNy #t (TO 'K2)))
   (STATE 'K2 (POP ** #t))))
```

Implementing Concurrent DPEs

Using the ability to PUSH a named ATN facilitates the composition of ATNs containing the sequencing, selection and repetition operators, but by itself is not powerful enough to construct an ATN representing a DPE that contains the concurrent operator. Using PUSH states will not suffice, as they do not return until the entire pushed ATN is matched or fails: a partial match is not possible. The concurrency operator requires us to make use of the registers and tests of ATNs. By carefully constructing the conditions on transitions, ATNs can be applied to the task of recognizing strings defined by concurrent DPEs. But first it is necessary to explain the use of the concurrency operator in more detail.

From the point of view of event recognition, we are interested in recognizing events after they occur. The concurrency operator doesn't constrain the order of events, only the possible events. If `A`, `B`, `C` and `D` are machines that accept the single events a, b, c and d respectively, then after the events have occurred, the following are true:

```
A & B       = ab + ba
A;B & C     = abc + acb + cab
A;B & C;D = abcd + acbd + acdb + cabd + cadb + cdab
```

This idea is the basis of Shuffle Automata, discussed previously in section two.

First consider the simplest case: a DPE that contains only the concurrency operator, for example: `A & B & C`. The ATN representing this DPE can be implemented similar to `(A + B + C)*` where the repetition represented by the `*` is constrained to one iteration for each different machine. How can this constraint be enforced? By carefully formulating the test on the `POP` arc leaving the final state of this ATN. The ATN would look like this:

```
(define ABC&            ;;; A & B & C
 (MakeATN
  (STATE 'S0 (CAT (MakeCategory 'A) #t (TO 'S0))
             (CAT (MakeCategory 'B) #t (TO 'S0))
             (CAT (MakeCategory 'C) #t (TO 'S0))
             (POP #t ConditionsMet?))))
```

What keeps this ATN from accepting `abca`? The procedure `ConditionsMet?` on the `POP` arc. `ConditionsMet?` is a procedure that returns `true` only if precise conditions are met. In this case the conditions are that `A`, `B` and `C` occur at least and at most once. The `ConditionsMet?` procedure will be different for each DPE, but this is not a problem when using Lisp-type languages where unnamed procedures are easy to implement.

Next consider the case where the operands to the concurrency operator are ATNs. As explained previously, it is not feasible to `PUSH` the ATNs, as is done with the other DPE operators. Instead, the operand ATNs themselves are used in the `ConditionsMet?` procedure to determine if the proper constraints are enforced. Now the concurrent ATN must send the proper terminal symbols to the `ConditionsMet?` procedure. The capability of employing registers in ATNs is used for this task. For example, consider the implementation of the following DPE:

```
A* & B
```

This is defined by the following ATN:

```
(define ATN
 (MakeATN
  (STATE 'S0 (CAT (MakeCategory 'A)
                  (SETR 'REG1 AddTo) (TO 'S0))
             (CAT (MakeCategory 'B)
                  (SETR 'REG2 AddTo) (TO 'S0))
             (POP #t ConditionsMet?))))
```

`AddTo` is a procedure that adds the contents of the current symbol to the named register. (The `AddTo` procedures were left out of the previous example to simply it, though in practice they are necessary.) The procedure `ConditionsMet?` examines these registers and uses them as input to the respective ATN operands. Only if each of the ATNs reach a final state is the boolean value `true` returned by `ConditionsMet?`. Only then is the concurrent ATN's `POP` state executed, otherwise the boolean value `false` is returned.

Consider `A B A` as input to the above ATN. `REG1` will contain `A A` as input to the ATN `A*`. `REG2` will contain `B` as input to the ATN `B`. Each of these will return `true`. Therefore the final state of the concurrent ATN will have been reached, and it will also return `true`.

Consider `A B A B` as input to the same ATN. `REG1` will still contain `A A` as input to the ATN `A*` and will still return `true`. `REG2` will now contain `B B` as input to the ATN `B` and will return `false`. The final state of the concurrent ATN will not have been reached. Therefore the ATN will return `false` for the input `A B A B`.

As a final example consider the following more complicated DPE:

```
((A + B)* & (X + Y + Z) & (O ; P ; Q))
```

The ATN for this DPE follows.

```
(define ATN
```

```
(MakeATN
 (STATE 'S0 (CAT (MakeCategory 'A 'B)
                 (SETR 'REG1 AddTo) (TO 'S0))
            (CAT (MakeCategory 'X 'Y 'Z)
                 (SETR 'REG2 AddTo) (TO 'S0))
            (CAT (MakeCategory 'O 'P 'Q)
                 (SETR 'REG3 AddTo) (TO 'S0))
            (POP #t ConditionsMet?))))
```

When given `O A P Z Q` as input to this ATN, `REG1` will contain `A` as input to the ATN `(A + B)*`, `REG2` will contain `Z` as input to the ATN `(X + Y + Z)`, and `REG3` will contain `O P Q` as input to the ATN `(O ; P; Q)`. Each of these will return `true`, the final state of the concurrent ATN will have been reached, and it will return `true`.

V. Overview of the Implementation

The author has implemented a concurrent event recognizer with ATNs using the techniques presented in the previous sections. It was implemented in the language Scheme and designed to be portable across many platforms. It will run on any Revised^4 Report compliant version of Scheme. [Clinger and Rees, 91] (There are many Scheme implementations available without charge that run on numerous systems. See the Scheme "Frequently Asked Questions" list that is posted monthly to the Usenet newsgroup `comp.lang.scheme` for details.)

When the program begins, the user is presented with an introductory screen that shows some example DPEs. After the introduction, the program prompts the user to type a DPE using semicolons (;) for sequencing, plus signs (+) for selection, asterisks (*) for Kleene closure and ampersands (&) for concurrency. Double bars (||) also signify concurrency and no operator between variables defaults to sequencing. Asterisks have the highest precedence, followed by sequencing, selection and concurrency, respectively.

Since the operator precedence may not be obvious to users, the expression is converted from infix to prefix notation, making apparent what operator is applied to specific operands. The operators are converted to text

abbreviations of their function: the sequencing operator is abbreviated `seq`, the selection operator is `slt`, asterisks are converted to `rep` and the concurrency operator to `cncur`. (The prefix expression is also the expression used internally.) The user is given the chance to abandon this expression before an acceptor is generated for it, if it is not the expression the user expected. An example will help clarify this. The user's responses are in bold type.

```
Expression> open; (read + write)*; close

The infix expression:

     open; (read + write)*; close

converted to prefix form looks like this:

      (seq open (rep (slt read write)) close)

Is this the expression you expected?  [yes]
```

The most common answers to questions posed to the user are presented in brackets as defaults. Pressing the enter key accepts the default answer. If the default answer is not desired, the user should type 'n' or 'no' or some synonym for negative (many are recognized).

When the user is satisfied that the expression is the desired one, the recognizer is generated from the prefix expression. If there are no problems generating the recognizer, then the user is asked to enter strings to determine if they are accepted by the language defined by the expression entered previously. The user's string is echoed to the console with a message that informs the user that the string is or is not contained in the given expression. The user continues to enter strings or the word 'quit' when the user is finished with the current expression. (The word 'quit' is tested before the user's string is passed to the recognizer; therefore, if 'quit' is contained in the language, it will not be passed to the recognizer). A continuation of the previous example will help clarify this:

```
Generating Parser for open; (read + write)*; close

The Parser has been generated.

You may enter strings in the language:

     open; (read + write)*; close

DPE> open close

     (open close) is accepted!!

DPE> open write read close

     (open write read close) is accepted!!

DPE> read close

     The expression (read close) is NOT accepted by
     the expression:

          open; (read + write)*; close


DPE> quit
```

Part of the system's response (such as advice on how to exit the system) was edited for brevity.

Future Improvements

Optimizing ATNs to remove `PUSH` states improves efficiency by decreasing the amount of backtracking. The current implementation does not attempt to do this. There are certain transformations that can be applied to ATNs to improve their efficiency and decrease the number of states required even further. Also there are certain transformations that could be applied to DPEs: `read* & write*` is syntactically equivalent to `(read`

`+ write)*`. The latter expression requires fewer states. This type of expression could be recognized and replace with the simpler expression without affecting the result.

VI. Conclusions

The author has implemented a concurrent event recognizer with ATNs using the techniques outlined in this paper. It was implemented in the language Scheme and designed to be portable across many platforms.

Event recognition is usually a part of a larger system that generates or manipulates program events. This implementation will fit readily in such a system, but it can also stand own its own. It has a rudimentary, but forgiving user interface that prompts the user for a DPE and generates the recognizer that will accept strings in the language defined by DPE. The user tries various strings and the system displays a message informing the user whether the string is contained in the language defined by the DPE. Since DPEs contain the set of regular expressions, the system performs equally well for regular languages.

Many approaches were tried before arriving at the method presented in this paper. At first, a matrix method similar to PAs but using ATNs was attempted. This method works well for simple DPEs that contain only one concurrency operator, but becomes much more complex with even simple DPEs that use two or more concurrency operators, such as `A & B & C & D`. Using the PA method, this expression requires 16 states and even the analytic representation is difficult to follow. When represented as a PA, the transition graph of this expression is useful only to topologists. As the number of terms in the expressions increases, the complexity of the PA method becomes unmanageable.

The method discussed in this paper was arrived at by noticing the similarity of expressions of the type `A & B & C & ...` to expressions of the type `(A + B + C + ...)*`, with the repetition possibilities restricted by the conditional transitions of ATNs, as mentioned previously.

This project has shown how that ATNs can be used to implement a recognizer for concurrent events. But how does this method compare with previous methods of concurrent event recognition? The most widely documented method used previously is Hseush and Kaiser's predecessor

automata. Their examples show that implementing a concurrent DPE using PAs with n concurrent operands requires 2^n states. Using the method described in this paper, the number of states increases linearly with the number of concurrent operands. In general, this method will require only two states in addition to the number of states required for each of the individual operands. While there are other factors to consider in evaluating the two approaches, for DPEs with a large number of concurrent operands this factor will become the most critical.

References

Bates, P. Distributed Debugging Tools for Heterogeneous Distributed Systems. In *8th International Conference on Distributed Computing Systems,* pp. 308-315. Computer Society Press. San Jose CA, June, 1988.

Campbell, R. H., and Habermann, A. N. The Specification of Process Synchronization by Path Expressions. In *Operating Systems,* Goos, G., and Hartmanins, J., eds., pp. 89-102. Springer-Verlag, Berlin, 1974.

Clinger, W., and Rees, J. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, Vol IV, No. 3, July-September 1991.

Hseush, W., and Kaiser, G. E. Modeling Concurrency in Parallel Debugging. *ACM SIGPlan Notices,* March 1990, pp. 11-20.

Kreutzer, W., and McKenzie, B. *Programming for Artificial Intelligence: Methods, Tools and Applications*. Addison-Wesley Publishing Company, Inc. Reading, MA, 1990.

Ponamgi, M. K., Hseush, W., and Kaiser, G. E. Debugging Multithreaded Programs with MPD. *IEEE Software,* May 1991, pp. 37-43.

Woods, W. A. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM,* 13, 1970, pp. 591-606.