Modeling Concurrency in Parallel Debugging

Wenwey Hseush*
Gail E. Kaiser+

Columbia University
Department of Computer Science
New York, NY 10027

(212) 854-8123
hseush@cs.columbia.edu

21 September 1989

CUCS-460-89

Abstract

We propose a description language, *Data Path Expressions* (DPEs), for modeling the behavior of parallel programs. We have designed DPEs as a high-level debugging language, where the debugging paradigm is for the programmer to describe the expected program behavior and for the debugger to compare the actual program behavior during execution to detect program errors. We classify DPEs into five subclasses according to syntactic criteria, and characterize their semantics in terms of a hierarchy of extended Petri Net models. The characterization demonstrates the power of DPEs for modeling (true) concurrency. We also present *predecessor automata* as a mechanism for implementing the third subclass qf DPEs, which expresses bounded parallelism. Predecessor automata extend finite state automata to recognize or generate partial ordering graphs as well as strings, and provide efficient event recognizers for parallel debugging. We briefly describe the application of DPEs to race conditions, deadlock and starvation.

*Hseush is supported in part by the Center for Telecommunications Research. Part of this research was conducted while Hseush was a summer employee of the IBM TJ. Watson Research Center. +Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from IBM, AT&T, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

keywords: debugging, formal models, Petri nets, path expressions, synchronization

1. Introduction

We are concerned with debugging parallel programs. One approach to locating the causes of program misbehavior is for the programmer to provide a high-level description of the expected behavior and for the debugger to compare the expected and actual behavior during execution. Expected behavior is specified abstractly in terms of control flow, data flow and/or synchronization events. In this approach, defining an appropriate notation for modeling program behaviors is a crucial prerequisite to developing a debugger. The conventional debugging approach, exemplified by dbx [Linton 81], also models program behavior but at a lower-level, in terms of source code entities such as subroutine names and line numbers; the programmer is responsible for comparing expected with actual behavior during execution. We refer to the first approach as *problem-oriented*, and the second as *program-oriented.* Both are necessary in practical debugging, just as both specification-based (blackbox) testing and program-based (whitebox) testing are required for practical testing [Howden 87].

In this paper, we are concerned primarily with the problem-oriented aspect of debugging. We have developed a style of problem-oriented debugging for parallel programs called *data path debugging*. Parallel program behaviors are described in a formal notation called *Data Path Expressions* (DPEs), an extension of Bruegge and Hibbard's generalized path expressions for debugging sequential programs [Bruegge 85, Bruegge 83]. This work is in turn an application to debugging of Campbell and Habermann's classical work on path expressions for describing process behavior in operating systems [Campbell 74]. Other researchers also advocate a problem-oriented approach to parallel debugging (e.g., Bates [Bates 88a], Miller and Choi [Miller 88]). The primary advantage of our DPEs is that they model *true concurrency*, distinguished from interleaving, and are concerned with data as well as control flow.

Subclass	Semantic Model
Sequential DPEs	**Finite State Automata**
Multiple DPEs	**K-Safe Nets (subset)**
Safe DPEs	**K-Safe Nets**
General DPEs	**Petri Nets**
Extended DPEs	**Extended Petri Nets (subset)**

Figure 1-1: DPE Hierarchy

In our previous paper [Hseush 88a], we informally described preliminary work on DPEs and discussed how they could be used in debugging parallel programs. The goal of this paper is to formally define several subclasses of DPEs in terms of their syntax and semantics. We define five subclasses according to syntactic criteria, and characterize the semantics of each subclass using a hierarchy of extended Petri net models [Peterson 81] (see Fig. 1-1). Extended Petri nets are equivalent to Turing machines [Hack 75, Thomas 76].

In our DPE taxonomy, the first subclass expresses only sequential behavior. The second subclass also expresses limited concurrency, where no program branching (*e.g.*, if-then-else) is allowed in conjunction with process splitting (*e.g.*, fork or para-do). The third subclass expresses general bounded parallelism. The fourth permits unbounded parallelism, but without the ability to join an unknown number of threads. The fifth describes general concurrency.

We propose *predecessor automata* as an implementation vehicle for the third subclass, safe DPEs, which subsumes the first and second subclasses. Predecessor automata extend finite state automata to represent predecessor events, and thus can recognize or generate partial ordering graphs (or pomsets) [Lamport 78] as well as strings. The concurrent composition [Milner 80] of two predecessor automata preserves causal independence (*i.e.*, true concurrency), while the concurrent composition of two finite state automata loses this information.

The expected program behavior described by a programmer as DPEs is translated into a predecessor automaton for efficiently recognizing concurrent events during execution. The DPE debugger will be useful for parallel applications where race conditions, deadlocks and starvation are concerns, and some small examples are given in this paper. We are in the process of implementing safe DPEs as a high-level debugging language for both a concurrent extension of C and for the Meld concurrent object-oriented programming language [Kaiser 89].

Section 2 introduces DPEs and explains other background material necessary to understand the remainder of the paper. In section 3, we show the power of the five subclasses of DPEs in terms of extended Petri Net models. Section 4 presents the predecessor automata model for efficient implementation of safe DPEs in a debugging system. Section 5 discusses the practicality of DPEs for parallel debugging. We conclude by summarizing the contributions of our work.

2. Background

A DPE consists of up to three components: one or more events, zero or more relations among events and zero or more actions. Events and the relations among them specify the behavior of program execution, while actions are performed by the debugger on program or debugger variables (or input/output) when the particular behavior is recognized during execution. A set of operators, sequencing (*), exclusive selection (+), repetition (*), concurrency (&) and concurrency closure (@), express the basic relationships among events. Other relationships such as permutation, partial concurrency and total concurrency can be derived from the basic relationships [Hseush 88a].

2.1. Events and Actions

There are four kinds of events: control, data, conditional and compound. Control events represent control activities, such as procedure entry and exit. Since there is a simple mapping from program execution to source code, control events can be specified using the corresponding identifiers in the program's source code, notably procedure names. **Function.enter** is the entering to **Function**, and **Function.exit** is the exiting from **Function**.

Data events occur when the specified program states become true. Data events are denoted as " [**condition**] ", where the **condition** is an expression in the target programming language, augmented with the ability to express the history of program states and activities associated with data such as read and write. For example, " [X = 0] " is the event that variable X becomes equal to zero, " [X = X′ + 1] " is the event that X is incremented (X′ refers to the previous value of X), " [X] " means that X has been referenced, and " [X′] " describes the case where X has been updated.

Data events are not associated with any particular control thread when defined, even though they are eventually caused by specific control events during execution. The programmer need only specify the effects on program entities without the knowledge of which control activities cause them; the debugger detects when the effects occur and reports which control activities cause them. It is difficult to efficiently recognize data events without either hardware support or significant modifications to the compiler and/or run-time support of the programming language, but we do not address this here.[1]

Conditional events are control or data events with predicates attached. The format is "**event** [**condition**]", where **condition** is a predicate. Conditional events are recognized when the event occurs in a context where the condition is already satisfied. For example, "`READ [lock = 1]`" is the condition where the READ procedure is called while `lock` is equal to one.

Compound events are defined by associating identifiers with DPEs, and permitting new DPEs to be defined in terms of such identifiers in the style of context-free grammars, except that recursive and empty event definitions are not allowed. The format is "**event-id** = **dpe**", where **dpe** is a data path expression as defined in the next section.

Actions may be attached to events. The format is "**event** { **statements** }", where **statements** is treated as a single action. The action is evaluated when the event is recognized. For example, "`[ X' ] { counter = counter + 1; }`" means that every time X is updated, `counter` is incremented by one. Statements may involve program variables and/or debugger variables or functions, such as input/output and `break`.

2.2. Safe Concurrency

The term *safe concurrency* refers to the case of bounded parallelism and *unsafe concurrency* to unbounded parallelism. Language constructs designed for expressing concurrency (*e.g.*, fork-join) often permit unsafe concurrency. Examples of safe and unsafe programs are shown in Figs. 2-1 and 2-2.

```
parallel-do i = 0 to 5
begin
  ...
  philosopher(i);
  ...
end
```

Figure 2-1: A safe program of parallel-do

```
Loop:
 if(fork() != 0) {
   parent();
   goto Loop;
 }
 else child();
```

Figure 2-2: An unsafe program of forks

The semantics of safe concurrency is characterized as a subclass of Petri nets, *k*-safe nets [Peterson 81], where the maximum number of tokens in a place is bounded by *k*. A *k*-safe net assures bounded parallelism. Every program with safe concurrency can be represented by a k-safe net, and every k-safe net is equivalent to a program with safe concurrency. The corresponding **k-safe** and unsafe nets for the safe and unsafe programs are shown in Figs. 2-3 and 2-4.

[1] The IBM 8CE group [Garcia 89] is working on hardware support — a debugging monitor — that can be parameterized by the symbol tables generated by the compiler; when it detects that interesting variables are accessed, it generates an interrupt to return control back to the debugger. Our implementation for C will take advantage of this hardware support. Our implementation for Meld on a network of Sun 3 workstations involves extensive debugging facilities within the Meld run-time support.

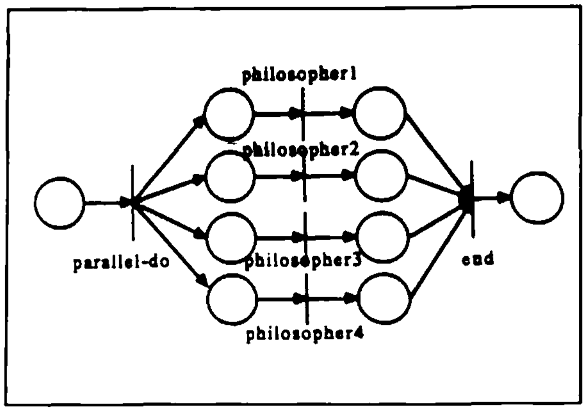


Figure 2-3: A safe net for the parallel-do program

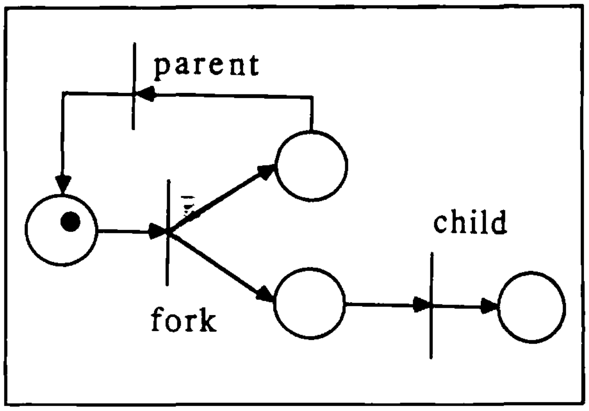


Figure 2-4: An unsafe net for the fork program

3. DPE Hierarchy

DPEs are classified into five subclasses by the operators employed and some other syntactic restrictions. Each subclass is also defined in terms of a semantic model and the corresponding programming domain. The first subclass is *sequential DPEs*, which expresses only sequential behavior. The second is *multiple DPEs*, which subsumes the first and expresses limited safe concurrency, where no program branching (*e.g.*, if-then-else) is allowed in conjunction with process splitting (*e.g.*, fork or para-do). The third subclass is *safe DPEs*, which expresses safe concurrency. The fourth subclass is *general DPEs*, which expresses limited unsafe concurrency, where unbounded parallel threads never join. The fifth subclass is *extended DPEs*, which subsumes the fourth one and expresses unsafe concurrency.

DPEs employ five operators: sequencing (;), selection (+), repetition(*), concurrency (&) and concurrent closure (@). Examples are shown in Table 3-1. Table 3-2 summarizes the DPE hierarchy, including equivalence proofs and related work.

Expression	Description
A;B	Event A causally precedes event B.
A+B	Either event A or event B occurs, but not both. + means exclusive-or.
A*	ε + A + A;A + A;A;A + ...
A&B	Event A and event B occur causally independently.
A@	ε + A + A&A + A&A&A + ...

Table 3-1: Operators

The first subclass is well known as regular expressions or path expressions. The path expression `"open ; (write + read)* ; close"` states that a file has to be opened, before an arbitrary sequence of reads and writes is performed, and then closed. To describe sequential program behavior like recursive procedure calls, control variables are used. `"E *n ; X *n"` states that the number of enterings is equal to the number of exiting, where `E` is the event of entering, `X` is the event of exiting, and the number of enterings is the same as that of exitings. The control variable `n` relates the number of occurrences of `E` to the number of occurrences of `X` with equality (=). One alternative is to simulate this behavior by actions attached to the events. In

Subclass	Sequential DPEs	Multiple DPEs	Safe DPEs	General DPEs	Extended DPEs
Expresses	sequential behavior	limited safe concurrency	safe concurrency	limited unsafe concurrency	unsafe concurrency
Syntax	**dpe$_1$: EVENT** **| (dpe$_1$)** **| dpe$_1$ + dpe$_1$** **| dpe$_1$; dpe$_1$** **| dpe$_1$ ***	**dpe$_2$: dpe$_1$** **| dpe$_2$ & dpe$_1$**	**dpe$_3$: EVENT** **| (dpe$_3$)** **| dpe$_3$ + dpe$_3$** **| dpe$_3$; dpe$_3$** **| dpe$_3$ *** **| dpe$_3$ & dpe$_3$**	**dpe$_4$: dpe$_3$** **| dpe$_4$ + dpe$_4$** **| dpe$_4$ & dpe$_4$** **| dpe$_4$ @**	**dpe$_5$: EVENT** **| (dpe$_5$)** **| dpe$_5$ + dpe$_5$** **| dpe$_5$; dpe$_5$** **| dpe$_5$ *** **| dpe$_5$ & dpe$_5$** **| dpe$_5$ @**
Semantic Model	finite state automata	*k*-safe nets subset	*k*-safe nets (see appendix I)	Petri nets [Garg 88]	extended Petri nets subset
Proof	Hopcroft & Ullman [Hopcroft 79]	Lauer & Campbell [Lauer 75]	see Appendix I	Garg [Garg 88]	see Appendix II
Example	**open;(read+write)*; close**	**(a;s;b) & (c;s;d)**	**(enq;deq)+(enq&deq)**	**(fork;parent)* & (fork;child)@**	**(enq;update)@;deq; display**
Limitations	no concurrency	no program branching preceding process splitting	no unbounded parallelism	no joining for unbounded parallelism	open question
Related Work	Generalized path expressions [Bruegge 85]	COSY [Lauer 81]	EBBA [Bates 83]	Concurrent regular expressions [Garg 88]	

Table 3-2: DPE Hierarchy

{ n = 0 } ; E { n = n + 1 } * ; X { n = n - 1 } * ; { if(n != 0) error }

the first action sets n to zero at the beginning of program execution. Then E occurs zero or more times and each time n is incremented by one. The sequence is then followed by the occurrences of X, where X might occur zero or more times and each time n is decremented by one. At the end of program execution, n is compared to zero.

The second subclass, multiple DPEs, expresses only global-level concurrency, where no nested concurrency (&) is allowed. Three examples are shown in Table 3-3. When the same event name appears in multiple subparts of the DPE, it is treated as a synchronization event and renaming is necessary to avoid this synchronization convention. We use (^) to distinguish two distinct events with the same name (see the third example). In concurrent programming, a synchronization event usually involves two events in different threads, as explained in subsection 3.1.

The third subclass, safe DPEs, allows multi-level concurrency. One example that can be expressed by safe DPEs but not multiple DPEs is "`enq ; deq + (enq & deq)`", which states that if the queue size is equal to zero, then enqueuing must precede dequeuing, otherwise, enqueuing and dequeuing can operate concurrently. Safe DPEs are equivalent to *k*-safe nets (the proof is given in appendix I).

The fourth subclass, general DPEs, expresses unbounded parallelism by using the concurrent closure operator (@), but disallows an event causally succeeding an unknown (unbounded) number of concurrent events. The general DPE "`(fork;parent)* & (fork;child)@`" models the program mentioned in Fig. 2-2 and the Petri net in

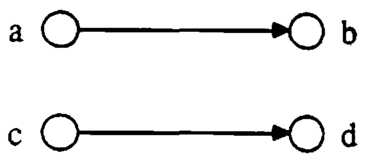
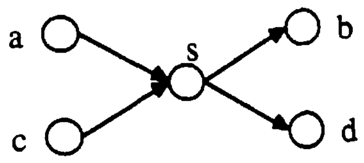
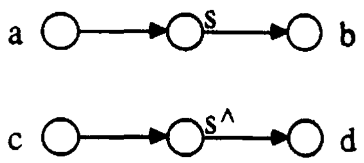
Multiple DPE	Description	Partial Order
(a;b)&(c;d)	**a;b** and **c;d** occur causally independently.	
(a;s;b)&(c;s;d)	Two causally independent paths **a;s;b** and **c;s;d** synchronize with each other at s.	
(a;s;b) & (c;s^;d)	**a;s;b** and **c;s^;d** occur causally independently.	

Table 3-3: Examples for multiple DPEs

Fig. 2-4. Some programming examples are: (1) an unbounded number of messages selecting the same method may arrive at an object and each message activates a control thread for executing the method without waiting for the prior activations to finish; and (2) an unknown number of signals arise and each signal invokes an unmasked signal handling routine.

General DPEs are also known as *concurrent regular expressions*. Concurrent regular expressions have been proven equivalent to Petri nets by Garg [Garg 88]. The limitations on general DPEs are the same as those on Petri nets: no zero testing [Keller 72]. Zero testing is the ability to test for zero tokens in an unbounded place of a Petri net. For example, "(A;B)@;C" is not expressible in general DPEs. It states that an unknown (unbounded) number of threads A;B are created, and when all B events in the concurrent threads are complete, then C occurs; note that C can occur while the number of non-processed B's is tested equal to zero, because of the concurrent closure operator. This expression cannot be described by a Petri net, but is expressible by an extended Petri net [Peterson 81].

The fifth subclass of DPEs, extended DPEs, allows an event causally succeeding an unknown (unbounded) number of concurrent events, as modeled by extended Petri nets. For example, "enq ; update) ; deq ; display" represents the case where an unbounded number of signals arise and each signal invokes a signal handling routine without disabling further signal invocations. The handling routine puts one character into a global queue and updates some information (assume the enqueue operation is atomic). After the control eventually returns to the main program, further signal invocation is disabled and all characters are dequeued and displayed. Extended DPEs express extended Petri nets, but whether extended DPEs are equivalent to extended Petri nets is an open question.

3.1. Synchronization Events

The behavior of language-specific synchronization primitives can be described using DPEs. Systems programmers or debugger users instruct the debugger to recognize the event patterns that constitute synchronizations among threads. For example, the pattern of sending a message X followed by receiving a message X constitutes a synchronization between the sender and the receiver. The description is

```
    SYNC(M): send(M); receive(M) { sync_event($1, $2); }
```

which instructs the debugger that send is causally related to receive by message M. Then a synchronization event from the send event ($1) to the receive event ($2) can be established by the debugger based on the information from the sender and the receiver, once both are recognized during execution. Otherwise, the debugger would have no knowledge that send and receive matched as a synchronization event.

Another example,

```
    SYNC(X): V(X).exit; P(X).exit { sync_event($1, $2); }
```

instructs the debugger that V.exit is causally related to P.exit by the shared datum X and constitutes a synchronization event. Say the set of events is P_1.enter, P_1.exit, P_2.enter, V_1.enter, V_1.exit, P_2,exit, V_2.enter, V_2.exit; the debugger uses the synchronization directive to establish the synchronization event (V_1.exit, P_2.exit).

This approach requires the same knowledge as in other approaches, but it provides the flexibility that users can easily invent and debug new synchronization primitives. In contrast, other debugging systems (e.g., [Goldszmidt 89]) retrieve such information through either source-to-source program transformation or augmenting the compiler with particular knowledge about synchronization primitives as related to parallel debugging.

4. Predecessor Automata

The problem-oriented debugging paradigm assumes that the programmer provides a description of expected program behavior, and the debugger compares this description to actual behavior at run-time to detect discrepancies. In our case, the debugger must be able to recognize sets of concurrent events matching DPEs. The debugger itself consists of support added (in hardware or software) to each executing thread or processor that submits messages representing primitive events to a centralized DPE recognition process. The sequence of events it receives are treated as tokens and are parsed them into a partial ordering graph according to the currently enabled set of DPEs. We are concerned here with the central recognition process, and do not address how the primitive events are generated.[2]

One key issue is the tradeoff between the efficiency of recognition and the memory space needed to represent the DPEs in a suitable internal form. In the case where minimizing memory space is most important, Petri nets are

[2] See previous footnote.

probably the best choice. Petri nets can represent sequential and concurrent behavior in a compact form, but they are relatively inefficient for recognizing events at runtime. In contrast, finite state automata (FSAs) are efficient recognizers for sequential behavior, but they cannot represent concurrent events that are causally independent. FSAs express interleaving semantics, but not true concurrency. The concurrent composition of two FSAs involves combining two FSAs into one such that all possible states and all possible interleavings of two sets of transitions are preserved [Milner 80]. This process loses the information regarding which events occur causally independently and there is no way to reverse the process to recover the original two FSAs. With or without concurrent composition, FSAs cannot distinguish two causally independent events interleaved with each other from two sequential events.

We present an implementation model, *predecessor automata* (PAs), that has the clean and efficient structure of finite state automata, but also the capability of representing true concurrency as in safe Petri nets. PAs can recognize behavior with safe concurrency and detect the situation of unsafe concurrency, and thus implement our third subclass, safe DPEs.

4.1. Definition of Predecessor Automata

A predecessor automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$.

- Q is a finite set of *states*.
- Σ is a finite set of *events*.
- δ is the *transition function* mapping $Q \times \Sigma \times P$ to Q, where P is the *predecessor set*, $P \subseteq \{\varepsilon\} \cup \Sigma \cup \Sigma \times \Sigma \cup \ldots$
- q_0 is the *initial* state, $q_0 \in Q$.
- F is the set of *final* states, $F \subseteq Q$.

The definition of a PA is the same as an FSA except for the transition function, which not only carries the information about the expected events, but also the information about its predecessors.

The predecessor p ($\in P$) of an event e is a set of events $\{w_1, w_2, \ldots, w_n\}$, where (1) n is a non-negative integer, (2) w_i causally precedes e and (3) w_i and w_j occur causally independently, $1 \le i, j \le n$, $i \ne j$. If $p = \varepsilon$, e is an *original* event (ε is also represented as '.') . The occurrence of event e implies that all its predecessor events $e_i \in p$, $1 \le i \le n$, have occurred.

The input to a PA is not a string of events, but a string of event-predecessor pairs, $(e_0\, p_0)$, $(e_1\, p_1)$, ..., $(e_n\, p_n)$, where $e_i \in \Sigma$ and $p_i \in P$, $0 \le i \le n$. The string of event-predecessor pairs can be considered as a partial ordering graph (or a directed acyclic graph), if for every event $w \in p_i$, $0 \le i \le n$, there exists an event e_j, $0 \le j \le n$, such that $e_j) = w$ and $j \ne i$. That is, every event mentioned as a predecessor event must occur. We usually assume $j < i$, which means the receiving order preserves the occurrence (partial) order. This is discussed in more detail later in this section.

A PA moves from one state q to another state r on an input $(e\, p)$, according to the transition function $\delta(\mathrm{q}, (e\, p)) = r$. That is, a move is made by examining the incoming event and its predecessors. Given a predecessor automaton PA, a sequence of transitions, $\delta(q_0, (e_0\, p_0)) = q_1$, $\delta(q_1, (e_1\, p_1)) = q_2$, ..., $\delta(q_{n-1}, (e_{n-1}\, p_{n-1})) = q_n$, where q_0 is the initial state, we need to construct a partial ordering graph. For each $(e_i\, p_i)$, $0 \le i \le n$,

1. Create a vertex labeled with e_i.
2. For each event $w \in p_i$, create a directed edge >from w to e_i.

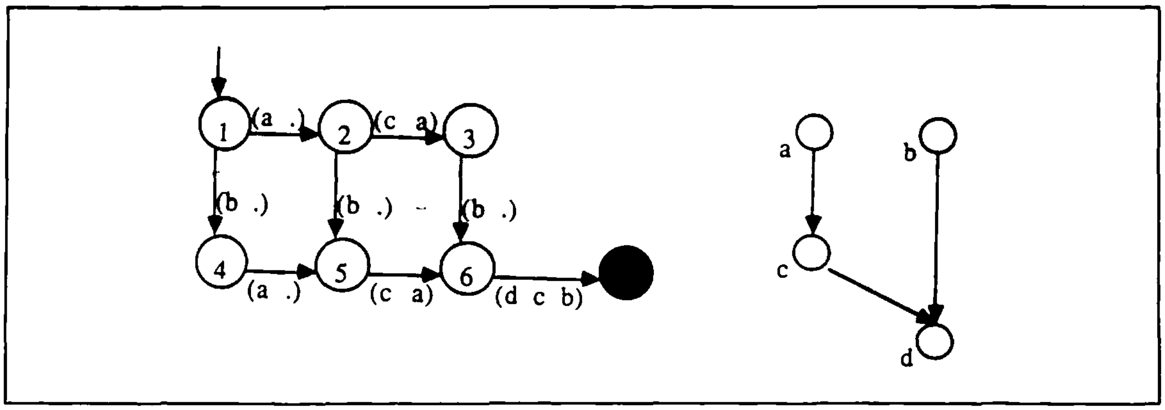


Figure 4-1: A PA and its partial ordering graph

Fig. 4-1 shows how to construct a partial ordering graph from a PA. Transitions (a ε) and then (b ε) create a graph with two vertices, a and b, and no edge. Then transition (c a) adds another vertex c and a directed edge a→c to the graph. Finally, transition (d (c b)) (also represented as (d c b)) adds vertex d and two directed edges c→d, b→d.

Two problems arise when constructing partial ordering graphs: (1) ambiguity and (2) instability. A PA is *ambiguous* if and only if there exists a sequence of transitions, $\delta(q_0, (e_0\, p_0)) = q_1$, $\delta(q_1, (e_1\, p_1)) = q_2$, ..., $\delta(q_{n-1}, (e_{n-1}\, p_{n-1})) = q_n$, where q_0 is the initial state, such that more than one partial ordering graph can be constructed. Fig. 4-2 illustrates an ambiguous situation.

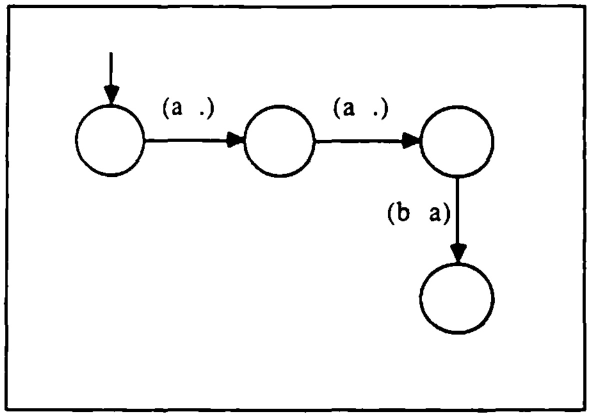


Figure 4-2: An ambiguous situation

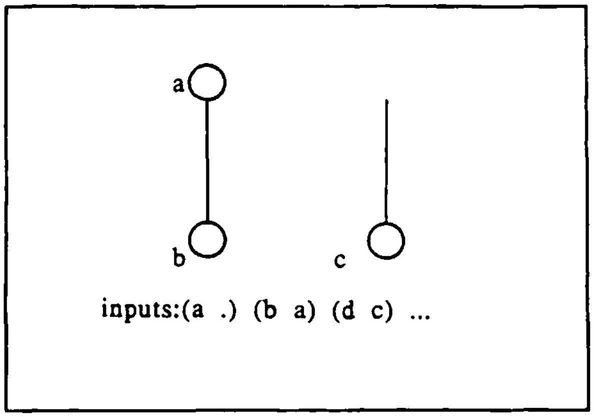


Figure 4-3: An unstable situation

The first step to eliminate the ambiguous situation is to rename some events e_i in the ambiguous PA, such that there does not exist an event e_j, $i \neq j$ and $e_i = e_j$. The standard approach is to rename every event e appearing in the PA to $q/e/r$, where q is the state when the transition is made and r the state that the transition leads to. A PA with a cycle is ambiguous, even after renaming. In this case, there is a second step that modifies the graph construction procedure. When a directed edge $w \rightarrow e$ is constructed, the vertex labeled with w is the one that was added to the graph most recently and labeled with w. These two steps eliminate all possible ambiguous situations.

Given a PA and a sequence of inputs, $(e_0\, p_0)$, $(e_1\, p_1)$, ..., $(e_n\, p_n)$, a situation is *unstable* at input $(e_i\, p_i)$ if and only if there exists a predecessor $w \in p_j$, $0 \leq j \leq i$, such that $e_k \neq w$ for all k, $0 \leq k \leq i\text{-}1$. Informally, a situation is unstable if an event mentioned as a predecessor has not arrived so far or the event is missing in the constructed graph. See Fig. 4-3.

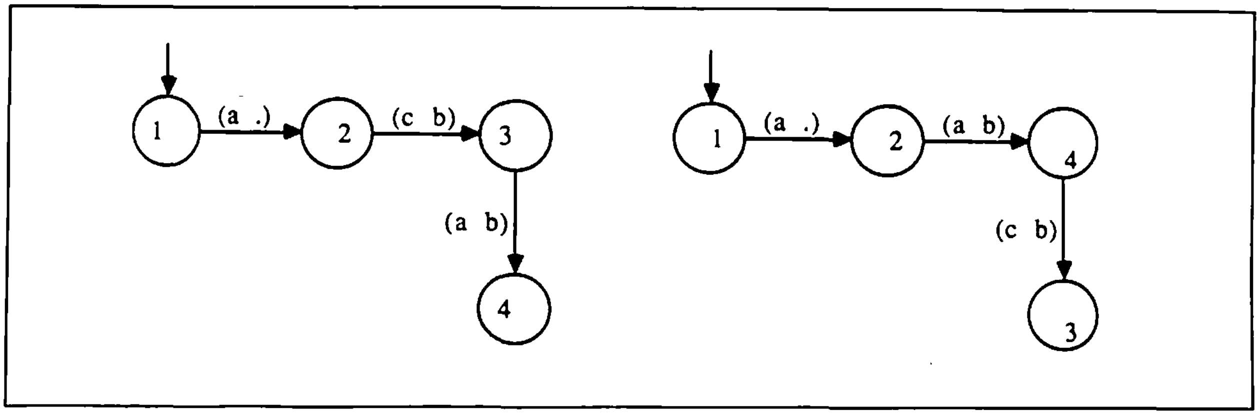


Figure 4-4: An unstable PA and the stabilized PA

A PA is unstable if and only if there exists a sequence of transitions $\delta(q_0, (e_0\, p_0)) = q_1$, $\delta(q_1, (e_1\, p_1)) = q_2$, ..., $\delta(q_{n-1}, (e_{n-1}\, p_{n-1})) = q_n$, where q_0 is the initial state, such that an unstable situation happens at $(e_i\, p_i)$, $0 \leq i \leq n$. An unstable PA can be stabilized by restricting the automaton, as shown in Fig. 4-4. An unstable situation may still arise in a stable PA, when the given sequence of inputs does not preserve the partial ordering in which the events occur. The way to avoid the unstable situation is to rearrange the ordering of the given sequence $(e_0\, p_0)$, $(e_1\, p_1)$, ..., $(e_n\, p_n)$, such that for every $w \in p_i$, there exists e_j, $j < i$, $w = e_j$. The receiving ordering preserves the occurrence partial ordering.

4.2. Event Recognition

As discussed above, recognition involves two components: (1) a target system that reflects the actual program behavior and provides the information about primitive events and their predecessors, and (2) a recognizer that represents the expected program behavior in some internal form and collects and processes the information. The target system is a concurrent system, messages representing primitive events and their predecessors are generated from different processors and sent to the centralized recognizer.

The recognizer is a sequential machine that receives messages representing primitive events from different threads one message at a time, compares them with the expected ones, and eventually reports the results. The message receiving order for primitive events and their predecessors is assumed independent from the order of the event occurrences, since the sending order may be different from the receiving order. The recognizer has two parts, a stabilizer and a PA. The stabilizer has two functions: (1) establishing synchronization events according to the descriptions of synchronization behavior, provided by system programmers or users, and (2) regulating the incoming event/predecessors messages such that the ordering of event messages that go into the automaton preserves the partial ordering of event occurrences in the target system. For example, if the input messages to the stabilizer are $(e_1\ \varepsilon)$ $(e_2\ e_3)$ $(e_3\ e_1)$ $(e_4\ e_1)$ $(e_5\ e_4)$, where e_2 is a send event and e_4 is a receive event, and the debugger has been instructed that a send event followed by a receive event is a synchronization event; the output of the stabilizer is $(e_1\ \varepsilon)$ $(e_3\ e_1)$ $(e_2\ e_3)$ $(e_4\ e_1)$ $(e_5\ \{e_4,\ e_2\})$. The output messages of the stabilizer are the input messages of the PA, which will compare the input messages (the actual behavior) with the DPEs (the expected behavior) provided by the users as represented by a PA. A general structure for such a debugging system is shown in Fig. 4-5.

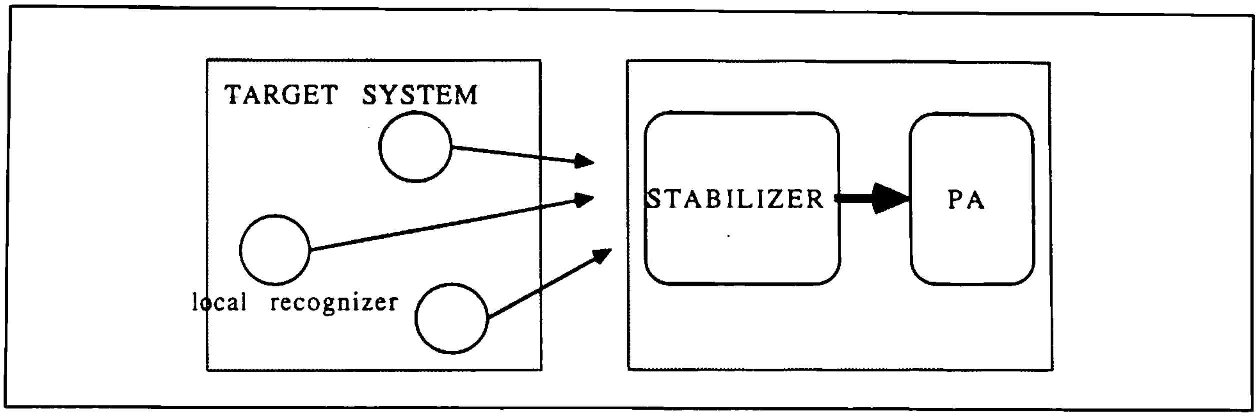


Figure 4-5: A general structure for event recognizers

The PA is in the initial state before receiving any messages. Every time a message describing an event and its predecessors arrives from the stabilizer, the PA compares the received information with the transitions directed from the current state. If both the event and its predecessors match one of the transitions, the automaton makes a move to the next state according to the matched transition. In the meantime, a partial ordering graph can be constructed. An example is illustrated in Figs. 4-6 and 4-7. One important assumption in our event recognition framework is that the target system (eventually) has full knowledge about every event that occurs and its predecessor events, where these events appear in some DPE used to construct the PA and/or reflect synchronization events.

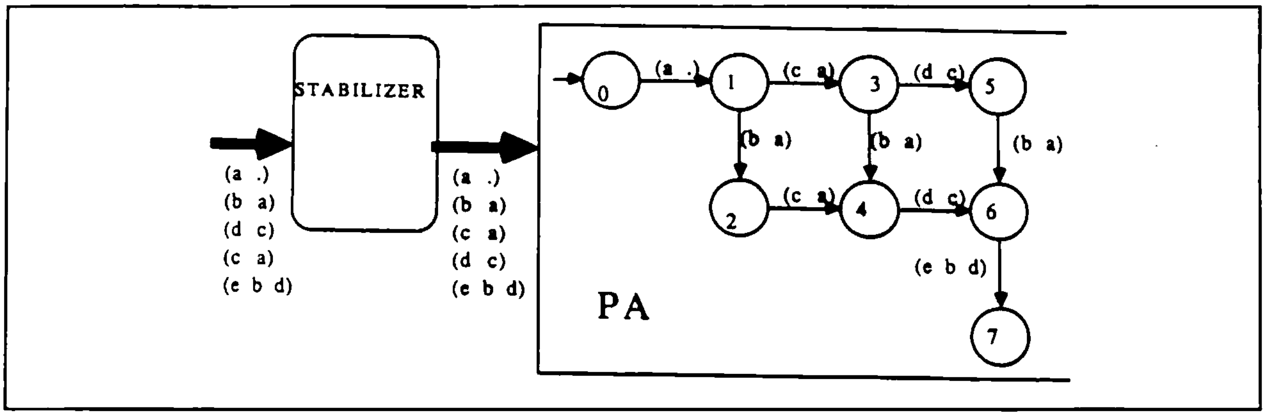


Figure 4-6: An event recognizer for a;(b&(c;d));e

Targeted system events	stabilizer events	predecessor automata state transitions
(a .)	(a .)	0 --> 1
(b a)	(b a)	1 --> 2
(d c)		
(c a)	(c a)	2 --> 4
	(d c)	4 --> 6
(e b d)	(e b d)	6 --> 7

Figure 4-7: PA Description

4.3. Constructing Predecessor Automata From Safe DPEs

Given a safe DPE, a predecessor automaton can be constructed. There are two steps, involving transformations of subexpressions and translation using an attribute grammar [Knuth 68]. The first step is to transform each expression into a new expression where there are no subexpressions R^*, such that $\varepsilon \in R$. For example, $(e^*)^*$ can be transformed into e^*. This guarantees that the constructed automaton has no transition cycles $\delta(q_1, (e_1\ p_1)) = q_2$, $\delta(q_2, (e_2\ p_2)) = q_3, ..., \delta(q_n, (e_n\ p_n)) = q_1$, such that $e_i = \varepsilon$, for all i, $0 \leq i \leq n$. The transformation is based on an extension to Foster's conversion theorem [Foster 86], which states that for any regular expression **R**, there is a regular expression **N(R)** such that

- N(R) does not contain the empty string,
- R* = (N(R))*
- N(R) is no longer than R.

If $\varepsilon \neq$ **R**, **N(R)** = **R**. Otherwise, there are three cases.

1. If R = P*, N(R) = N(P)
2. If R = P+Q*, N(R) = N(P) + N(Q)
3. If R = P;Q*, N(R) = N(P) + N(Q)

The theorem was originally defined for regular expressions, which only have three operators (; * +). In safe DPEs, a new operator (&) is employed. We augmented the theorem to apply to safe DPEs. Thus there is a fourth case.

1. If R = P&Q*, N(R) = (N(Q) & N(P)) + N(Q) + N(P)

But the third condition in Foster's conversion theorem (**N(R)** is no longer than **R**) is no longer true.

The second step applies an attribute grammar that specifies how to construct a PA. A DPE is first parsed into an abstract syntax tree, where three attributes are attached to each node of the tree, **AUTO**, **PRED** and **LAST**. The **AUTO** attribute of a node n will contain an automaton that represents the subtree (subexpression) rooted at node n. A subtree can be considered as a subexpression or a PA. The **PRED** attribute of n represents its predecessors, the events that might precede any event occurring in the subtree rooted with n. The **LAST** attribute of n refers to the events without successors in the subtree. The values of **PRED** and **LAST** have the form $(e_{0,0} \wedge e_{0,1} \cdots \wedge e_{0,m}) \vee (e_{1,0} \wedge e_{1,1} \cdots \wedge e_{0,k}) \vee \cdots \vee (e_{n,0} \wedge e_{n,1} \cdots \wedge e_{n,m})$, where the events related with ($\wedge$) occur concurrently and the events related with ($\vee$) occur exclusively. The semantic rules associated with the grammar are shown in Fig. 4-8.

This is not a syntax-directed translation system like YACC [Johnson 78]. Instead, the semantic rules describe the relations between a node in the abstract syntax tree and its parent node, and between the node and its children nodes. A semantic rule is evaluated only when its dependent attribute(s) is changed,[3] instead of at the time of parsing. For example, the first semantic rule, "`dpe.AUTO = new_automaton(EVENT, dpe.PRED)`", which is associated with a leaf node, is evaluated when its **PRED** attribute is changed.

The function `last_events`, with a PA as an input parameter, obtains the last events that might occur in the PA. The return value has the same form as **LAST** and **PRED**. The function `new_automaton` creates a new automaton with two input parameters, an event e and its predecessors p. The new automaton has one start state p,

[3] Using, for example, Reps's incremental evaluation algorithm [Reps 84].

```
dpe : EVENT
  {
   dpe.AUTO = new_automaton(EVENT, dpe.PRED);
   dpe.LAST = last_events(dpe.AUTO); /* EVENT */
  }
  | '(' dpe1 ')'
  {
   dpe1.PRED = dpe.PRED;
   dpe.AUTO = dpe1.AUTO;
   dpe.LAST = dpe1.LAST;
  }
  | dpe1 ';' dpe 2
  {
   dpe1.PRED = dpe.PRED;
   dpe2.PRED = dpe1.LAST;
   dpe.AUTO = concat(dpe1.AUTO, dpe2.AUTO);
   dpe.LAST = dpe2.LAST;
  }
  | dpe1 '+' dpe 2
  {
   dpe1.PRED = dpe.PRED;
   dpe2.PRED = dpe.PRED;
   dpe.AUTO = union(dpe1.AUTO, dpe2.AUTO);
   dpe.LAST = dpe1.LAST ∨ dpe2.LAST;
  }
  | dpe1 '*'
  {
   dpe1.PRED = dpe.PRED;
   dpe.AUTO = repeat(dpe1.AUTO);
   dpe.LAST = dpe1.LAST ∨ ε;
  }
  | dpe1 '&' dpe 2
  {
   dpe1.PRED = dpe.PRED;
   dpe2.PRED = dpe.PRED;
   dpe.AUTO = compose(dpe1.AUTO, dpe2.AUTO);
   dpe.LAST = last_events(dpe.AUTO);
  }
```

Figure 4-8: Attribute Grammar for DPEs

one final state q and one transition $\delta(p\ (e\ \mathbf{PRED}(e))) = q$.

The attribute grammar evaluation is started by setting the **PRED** attribute of the root to ε; every node will eventually be visited a few times, as changes are propagated around the tree. The root is the first node visited, since its **PRED** is changed. For each node e visited, if e is a leaf, **AUTO** is assigned a new PA and **LAST** is set to e. Since the values of **PRED** and **AUTO** are changed, its parent node will be visited again according to the semantic rules associated with the parent. If the node is not a leaf, it propagates the value of **PRED** down to its child nodes, and when the node is eventually visited again, it constructs a new PA from its children's PAs according to the operators and properly sets the value of its **LAST** attribute. The functions `concat`, `union` and `repeat` are depicted in Fig. 4-9. The function `compose` concurrently composes two PAs into one, as explained in the next section. When the evaluation is complete, the **AUTO** attribute of the root contains the PA for the given DPE.

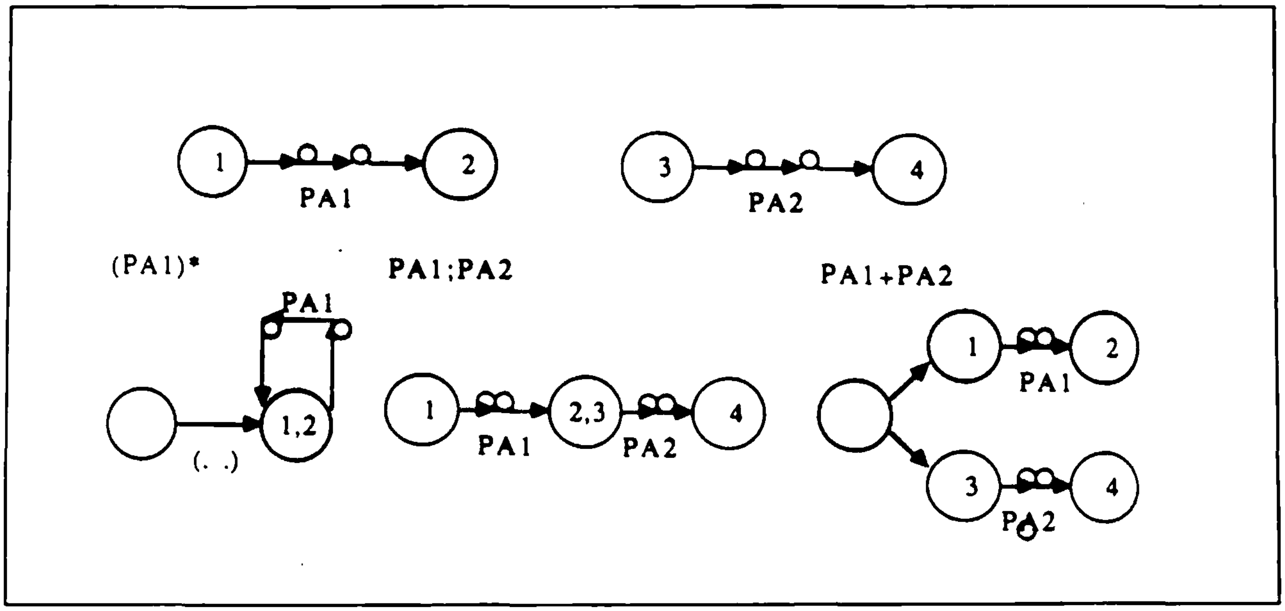


Figure 4-9: Functions for constructing PAs

4.4. Concurrent Composition

The concurrent composition of two PAs creates a new PA that preserves all possible states and all possible transitions as if the two original automata operate concurrently. As explained above, the concurrent composition of two finite state automata will lose the concurrency information, while the concurrent composition of two PAs will not. An example is shown in Fig. 4-10.

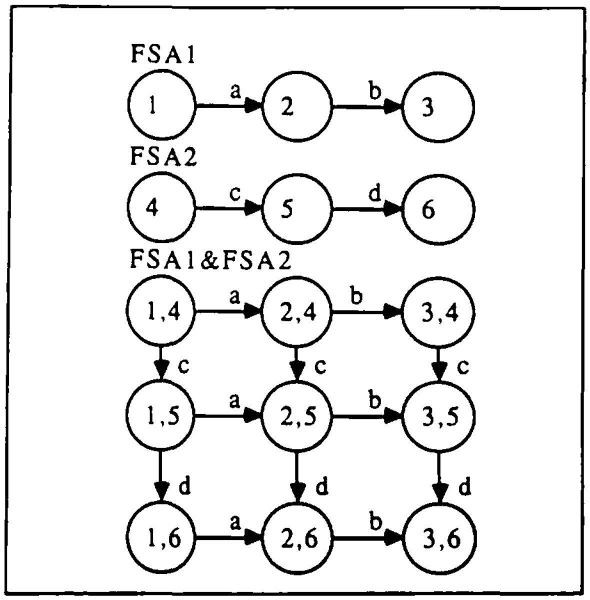


Figure 4-10: Concurrent composition of two FSAs

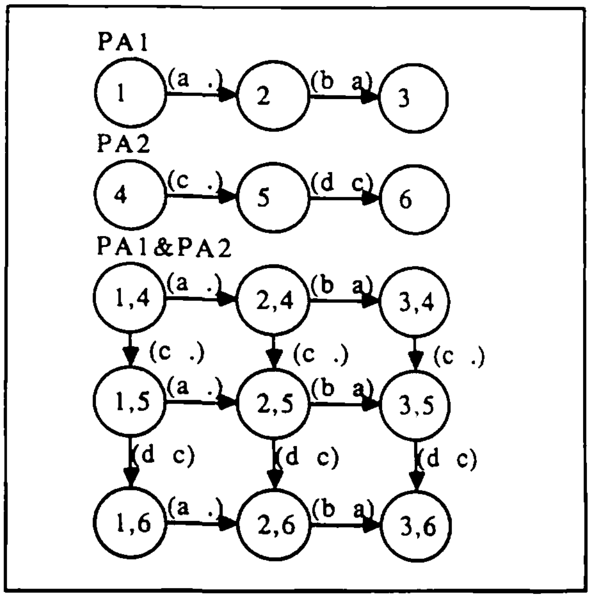


Figure 4-11: Concurrent composition of two PAs

Composition of two PAs can be divided into two cases, those that do and do not involve synchronization. Synchronization occurs when two automata have common events (or reflect components of synchronization events, as explained previously). Assume the first automaton has n states, $s_1, s_2, ..., s_n$, s_0 is the initial state, and the second automaton has m states, $z_1, z_2, ..., z_m$, z_0 is the initial state. In the case that two PAs have no synchronization, the composed automaton will have $n \times m$ states, $q_{1,1}, q_{1,2}, ..., q_{n,m-1}, q_{n,m}$. The state $q_{i,j}$ is the combined state of the state s_i in the first automaton and the state z_j in the second automaton. The transitions from $q_{i,j}$ to $q_{k,j}$ in the composed automaton are the transitions from s_i to s_k in the first automaton, and the transitions from $q_{i,j}$ to $q_{i,k}$ in the composed automaton are the transitions from z_j to z_k in the second automaton. No transitions hold between $q_{i,j}$ and $q_{h,k}$, $i \neq h$ and $j \neq k$.

In the case where two PAs do have synchronization, the concurrent composition consists of two steps. The first is to transform two PAs into one Petri net, where the places in the Petri net are the states in the automata, the transitions in the Petri net are the transitions in the automata, and multiple transitions with a common event are combined into one transition with multiple inputs and outputs. For example, a transition $(s_i\ e\ p_i\ s_j)$ and a transition $(s_k\ e\ p_k\ s_h)$ will be combined into a transition $(s_{i,k}\ e\ p_i \wedge p_k\ s_{j,h})$.

The second step is to find all possible states and transitions for the composed automaton by the following procedure. An example is shown in Fig. 4-11.

1. Let $Q = \{ (s_0\ z_0) \}$, $R = \emptyset$, $T = \emptyset$.
2. If $Q = \emptyset$, stop.
3. For every $v \in Q$, set $Q = Q - \{v\}$, $R = R \cup \{v\}$.

4. Let V be a set of all possible states succeeding v, for every $u \in V$, $u \notin R$, set $Q = Q \cup \{u\}$,
5. Set $T = T \cup \{$ transitions from v to u $\}$, goto step 2.

4.5. Related Work

EBBA [Bates 88b] employs *shuffle automata* [Bates 87] as a formal model for event recognition in distributed systems. Shuffle automata recognize concurrent events based on the interleaving semantics. That is, shuffle automata cannot distinguish two causally independent events interleaving with each other from two causally dependent events.

Shuffle automata are an FSA-like formalism that consist of a set of states and a finite state control that effects transitions from an initial state to some final state. An important difference between the shuffle automaton and an FSA is that in order to make transitions in the shuffle automaton, the finite state control examines sets of input symbols, rather than individual symbols. At run-time, the recognizer will accumulate the incoming events in a set. Whenever a subset of the accumulated event set becomes sufficient to make a transition, the finite control then goes >from the current state to another state. In the example of Fig. 4-12, if events **a** and **b** occur causally dependently due to some synchronization events in the target system, when **a** and **b** arrive, the shuffle automaton will recognize these two events as concurrent.

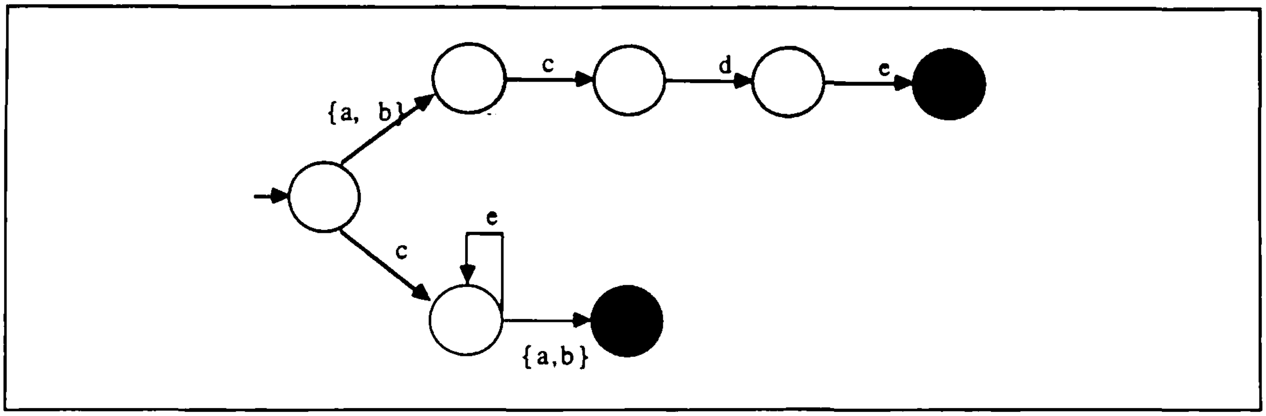


Figure 4-12: The shuffle automaton for ((a&b);(c;d;e))+(c;e*;(a&b))

5. Debugging Concurrent Programs

Most concurrency-related bugs involve problems with synchronization among multiple threads, which may share information in a number of different ways, including shared memory, message passing, files and devices, and human interaction. In this section, we demonstrate that DPEs are useful for aiding detection and correction of three typical kinds of synchronization errors: race conditions, deadlocks and starvations.

A race condition happens when two or more concurrent threads interact with some common resources without properly constraining the ordering of interactions, resulting in a computation that is nondeterministic and incorrect. To eliminate the race conditions, appropriate synchronization must be added to the program so that the crucial interactions are properly ordered and thus lead to a correct computation. Two types of synchronization mechanisms are frequently adopted: (1) wait-resume and (2) rollback-retry. Wait-resume constrains the ordering of interactions by blocking threads from competing for resources, but may lead to a deadlock situation when two or more threads wait for each other indefinitely due to lack of knowledge of the global situation. In the rollback-retry type of synchronization, a thread constrains the ordering of interactions by expecting other threads to complete their crucial interactions while temporarily releasing its resources. This may lead to a starvation situation where one or more threads repeats the rollback-retry cycle indefinitely. In the standard dining philosophers example, there is a deadlock when every philosopher has a fork in his right hand and is waiting for the fork on his left-hand side; there is starvation when a philosopher repeatedly picks up the forks on his right-hand side and then puts down the fork because the fork on his left-hand side is always unavailable.

It is difficult to debug programs with race conditions, deadlocks or starvations, where bugs may be embedded in (1) the synchronization primitives and/or (2) the program units that apply the synchronization primitives. It is also difficult for programmers to detect, by observing the external program behavior, whether the error is caused by buggy synchronization primitives or buggy program units. We assume in this paper that synchronization primitives are always correct, and are thus concerned only with (2). One concern in debugging is reproducibility, since it is desirable for the identical program behavior to be replayed by re-execution or simulation over and over again until the bugs are located. We assume this is possible, but do not address the mechanism here.[4]

5.1. Debugging Race Conditions

There are two necessary conditions for race conditions: (1) concurrent threads share common resources, and (2) the particular events within these threads that compete for the common resources are causally independent. Therefore, debugging a program with race conditions can be treated as a process of establishing relations of causal dependence and detecting whether the critical events that access the common resources occur causally independently.

```
program producer_consumer;
var
  s: semaphore := 1;
  deposited: semaphore := 0;
procedure producer;
  var next: integer;
  begin
   while true do
     next = calculate();
     P(s); --------------------> (1)
     enqueue(next); ---------> (2)
     V(s);
     V(deposited);
   end;
  end;
```

```
procedure consumer;
  var next: integer;
  begin
   while true do
     P(deposited);
     P(s);
     next = dequeue();
     V(s);
     print(next);
   end;
  end;
begin
  para-do
   producer();
   consumer();
  para-end
end
```

Figure 5-1: Producer-Consumer Program

For example, Fig. 5-1 shows a simple producer-consumer program, where the producer thread puts numbers in a queue, and the consumer thread gets and prints the numbers from the queue when the queue is not empty. A semaphore `s` and its operations `P(s)` and `V(s)` are used for synchronization. Assume the P(s) at point (1) is

[4] In another paper [Hseush 88b], we describe a form of pseudo-replay where it is possible to force reproducibility in many cases by stepping through the partial ordering graph generated by DPE recognition with respect to a preparatory execution.

missing from the program. During execution, the queue data structure may become inconsistent. In order to debug the program, the first step is to define, using DPEs, the synchronization events in the program (see section 3.1).

Then, in the case where a race condition between producer and consumer is suspected, the second step is to describe, in DPEs, the expected misbehavior that enqueue and dequeue occur concurrently.

```
enqueue() & dequeue() { print(s); break; }
```

instructs the debugger to print the value of semaphore s and stop the execution when enqueue and dequeue occur concurrently.

The third step is to replay the program execution. The program execution will stop at (2) and the value of s is printed out. The debugger will detect the true concurrency of enqueue and dequeue, no matter how the event messages interleave with each other. Some interleavings might accidently produce correct results and others produce the wrong results; in both cases, the debugger will detect the race condition.

5.2. Debugging Deadlocks

There are four necessary conditions for deadlock [Coffman 71]:

1. Threads claim exclusive control of the resources they require (mutual exclusion condition).
2. Threads hold resources already allocated to them while waiting for additional resources (wait for condition).
3. Resources cannot be removed from the threads holding them until completion (no preemption condition).
4. A circular chain of threads exists in which each holds one or more resources that are requested by the next thread in the chain (circular wait condition).

Debugging a program with deadlock requires the same description of synchronization events as in debugging a program with race conditions, but has a more complicated expected program behavior.

One example is that LOCK and UNLOCK are used to allocate resources before reference to the data. The first three conditions are determined by the synchronization primitives, and the fourth condition can be established by constructing a wait-for graph during debugging. The synchronization events can be described as follows.

```
SYNC(X): UNLOCK(X).exit; LOCK(X).exit { sync_event($1, $2); }
```

The expected program behavior can then be described as

```
LOCK(X).exit{hold($1.pid, X)};UNLOCK(X).exit{unhold($2.pid, X)}
LOCK(X).enter;wait(){wait_for($1.pid,X);check_deadlock()};resume();LOCK(X).exit{release($4.pid,X)}
```

where (1) the hold() function informs the debugger that the thread of the event ($1.pid) holds the resource X, (2) unhold() tells the debugger that the associated thread ($3.pid) does not hold the resource X any more, (3) wait_for() means that the associated thread ($1.pid) waits for resource X, (4) release () that the associated thread no longer waits for the resource X, and (5) check_deadlock asks the debugger to check whether a deadlock exists according to the information provided by the first four functions.

5.3. Debugging Starvations

Starvation is a special type of race condition where a set of causally independent events might repeat indefinitely. There is no easy way to detect this. In the example of dining philosophers, every philosopher might repeat picking up the fork on his left-hand side and putting it down. One possibility for detecting this is to store the program state every time a philosopher picks up his right fork and compare it with the previous states. If there exists an identical previous state and between them no progress has been made, there may (or may not) be an error. Detecting starvation is probably more amenable to program verification than debugging, but DPEs can check the correctness of verification assertions during execution.

6. Conclusions

We have defined a formal notation, DPEs, for modeling concurrent behavior in the context of debugging parallel programs. There are five subclasses of DPEs, four equivalent in power to a member of a hierarchy of Petri net models and the fifth a subset of extended Petri nets. We have developed an efficient implementation vehicle for the third subclass of DPEs, which models safe concurrency. We have briefly described the application of DPEs to practical concurrent debugging problems, from a viewpoint of problem-oriented behavior. DPEs must be combined with conventional debugging mechanisms to observe program-oriented behavior, for example, to support single-stepping among statements and modification of the program state at a breakpoint.

Acknowledgments

Some of the ideas presented here originated in discussions with Timothy Balraj, who has been working on Petri net models [Balraj 86] and path expressions in the context of silicon compilers. We would also like to thank Janice Stone, Bowen Alpern, Felix Wu and Dannie Durand for their helpful discussion, and Colin Harrison for his support of DPE debugging. Krish Ponamgi is working with us on an implementation of DPEs for C on the IBM 8CE multiprocessor running the Mach operating system [Rashid 87]. Krish is a co-op MS student at IBM under the supervision of Colin Harrison. Yi-wun Lu and Taka Ishizuka have previously worked with us on the Meld Debugger (MD) implementation of DPEs on Sun 3 workstations.

References

[Balraj 86] T.S. Balraj and M.J. Foster.
Miss Manners: A Specialized Silicon Compiler for Synchronizers.
In *Proceedings of the Fourth MIT Conference*, pages 3-20. The MIT Press, April, 1986.

[Bates 83] Peter Bates and Jack C. Wileden.
An Approach to High-Level Debugging of Distributed System.
In *ACM SIGSoft/SIGPlan Software Engineering Symposium on High-Level Debugging*, pages 107-111. Pacific Grove, CA, March, 1983.
Special issue of *Software Engineering Notes*, 8(4), August 1983.

[Bates 87] Peter C. Bates.
Shuffle Automata: A Formal Model for Behavior Recognition in Distributed Systems.
Technical Report COINS 87-27, University of Massachusetts at Amherst, January, 1987.

[Bates 88a] Peter Bates.
Distributed Debugging Tools for Heterogeneous Distributed Systems.
In *8th International Conference on Distributed Computing Systems*, pages 308-315. Computer Society Press, San Jose CA, June, 1988.

[Bates 88b] Peter Bates.
Distributed Debugging Tools for Heterogeneous Distributed Systems.
In *ACM SIGPLAN/SIGOps Workshop on Parallel and Distributed Debugging*, pages 11-22. Madison WI, May, 1988.
Special issue of *SIGPLAN Notices*, 24(1), January 1989.

[Bruegge 83] Bernd Bruegge and Peter Hibbard.
Generalized Path Expressions: A High-Level Debugging Mechanism.
The Journal of Systems and Software 2(3):265-276, 1983.

[Bruegge 85] Bernd Bruegge.
Adaptability and Portability of Symbolic Debuggers.
PhD thesis, Carnegie Mellon University, 1985.
CMU-CS-85-174.

[Campbell 74] R.H. Campbell and A.N. Habermann.
The Specification of Process Synchronization by Path Expressions.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science.* Volume 16: *Operating Systems*, pages 89-102. Springer-Verlag, Berlin, 1974.

[Coffman 71] E. G. Coffman, Jr., M. Elphick; and A. Shoshani.
Systems Deadlocks.
Computing Surveys 3(2):71-76, June, 1971.

[Foster 86] M.J. Foster.
Avoiding Latch Formation in Regular Language Recognizers.
In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, pages 740-748. University of Illinois, Urbana-Champaign IL, October, 1986.
This paper also appears in the *IEEE VLSI Technical Bulletin*, 1(2), September 1986.

[Garcia 89] Armando Garcia, David J. Foster and Richard F. Freitas.
The Advanced Computing Environment Multiprocessor Workstation.
Technical Report RC14491, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, N.Y. 10598, February, 1989.

[Garg 88] Vijay Kumar Garg.
Specification and Analysis of Distributed Systems With a Large Number of Processes.
PhD thesis, University of California at Berkeley, 1988.

[Goldszmidt 89] German S. Goldszmidt, Shmuel Katz and Shaula Yemini.
High Level Language Debugging for Concurrent Programs.
Technical Report RC14341, IBM Research Division, T.J.Watson Research Center, Yorktown Heights, N.Y. 10598, January, 1989.

[Hack 75] M. Hack.
Decidability Questions for Petri Nets.
PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1975.
Technical Report 161.

[Hopcroft 79] John E. Hopcroft and Jeffrey D. Ullman.
Introduction to Automata Theory, Languages and Computation.
Addison-Wesley Publishing Company, 1979, pages 28-35.

[Howden 87] William E. Howden.
Software Engineering and Technology: Functional Program Testing & Analysis.
McGraw-Hill Book Co., New York, 1987.

[Hseush 88a] Wenwey Hseush and Gail E. Kaiser.
Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language.
In *ACM SIGPLAN/SIGOps Workshop on Parallel and Distributed Debugging*, pages 236-246. Madison WI, May, 1988.
Special issue of *SIGPLAN Notices*, 24(1), January 1989.

[Hseush 88b] Wenwey Hseush and Gail E. Kaiser.
Concurrent Breakpointing.
Technical Report CUCS-402-88, Columbia University Department of Computer Science, October, 1988.

[Johnson 78] S.C. Johnson and M.E. Lesk.
Language Development Tools.
The Bell System Technical Journal 57(6):2155-2175, July-August, 1978.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun Felix Wu.
Melding Multiple Granularities of Parallelism.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 147-166. Cambridge University Press, Nottingham, UK, July, 1989.

[Keller 72] R. Keller.
Vector Replacement Systems: A Formalism for Modeling Asynchronous Systems.
Technical Report 117, Computer Science Laboratory, Princeton University, December, 1972.

[Knuth 68] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[Lamport 78] Leslie Lamport.
Time, Clocks and the Ordering of Events in a Distributed System.
CACM 21(7):558-564, July, 1978.

[Lauer 75] P. E. Lauer and R. H. Campbell.
Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.
Acta Informatica 5(4):297-332, 1975.

[Lauer 81] P. E. Lauer and M. W. Shields.
Formal behavioural specification of concurrent systems without globality assumptions.
In J. Diaz and I.Ramos (editor), *Lecture Notes in Computer Science.* Number 107: *Proceedings of Internation Colloquium on Formalization of Programming Concepts*, pages 115-151. Springer-Verlag, Berlin, 1981.

[Linton 81] M. Linton.
A Debugger for the Berkeley Pascal System.
Master's thesis, University of California at Berkeley, June, 1981.

[Miller 88] Barton P. Miller and Jong-Deok Choi.
Breakpoints and Halting in Distributed Programs.
In *8th International Conference on Distributed Computing Systems*, pages 316-323. Computer Society Press, San Jose CA, June, 1988.

[Milner 80] Robin Milner.
A Calculus of Communicating Systems.
In G. Goos and J. Hartmanis (editors), *Lecture Notes in Computer Science (92)*. Springer-Verlag, Berlin, 1980.

[Peterson 81] James L. Peterson.
Petri Net Theory and The Modeling of Systems.
Prentice-Hall, Inc., Englewood Cliffs, NJ 07632, 1981.

[Rashid 87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky and Jonathan Chew.
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.
In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 31-39. Palo Alto CA, October, 1987.
Special issue of *SIGPlan Notices*, 22(10), October 1987.

[Reps 84] Thomas Reps.
Generating Language-Based Environments.
The MIT Press, Cambridge MA, 1984.

[Thomas 76] P. Thomas.
The Petri Net: A Modeling Tool for the Coordination of Asynchronous processes.
Master's thesis, University of Tennessee, 1976.

I. Safe DPEs = k-safe nets

Safe DPEs are equivalent to k-safe nets. That is, four operators (; * + &) are necessary and sufficient to express safe concurrency. In order to prove this, we have to show (1) every expression constructed with (; * + &) can be translated to a k-safe net, and (2) every k-safe net can be expressed by a safe DPE.

I.1. Proof: safe DPEs --> k-safe nets

By construction: Given an expression including only (; * + &), the corresponding k-safe net can be constructed by translating every operator with the associated events into a net graph (see Fig. I-1).

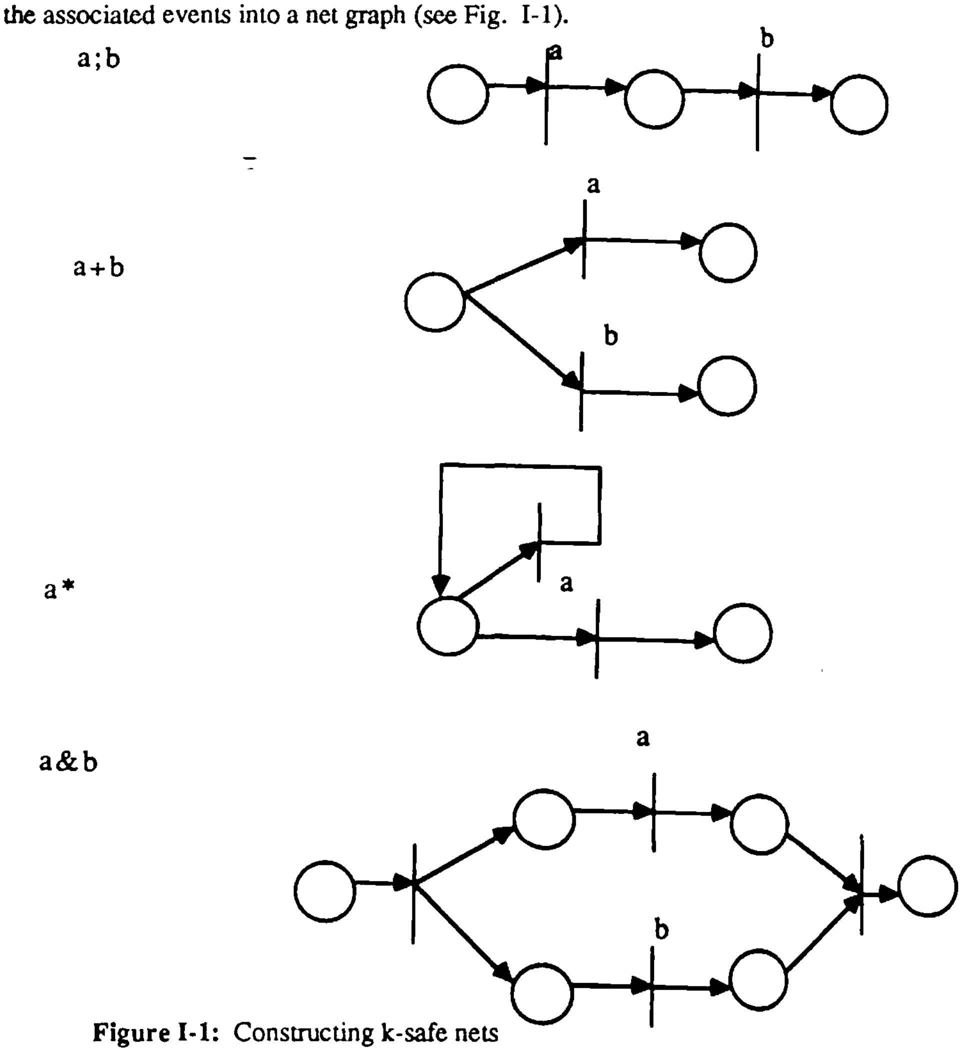


Figure I-1: Constructing k-safe nets

Since the number of tokens in each component is conserved, the number of tokens in the constructed k-safe net is conserved.

I.2. Proof: k-safe nets --> safe DPE

To show that every k-safe net can be expressed by a DPE constructed with (; * + &) is more difficult than the first part. Some definitions are given. Let *split transitions* be the transitions with multiple outputs, and *join transitions* be the transitions with multiple inputs; thus the degree of parallelism is increased by firing split transitions and decreased by firing join transitions. Here, *split* refers to the firing action of a split transition and *branch* refers to the selection of a multiple-output place. A standard k-safe net is a k-safe net with five conditions: (1) every split transition has exactly two outputs, (2) every join transition has exactly two inputs, (3) the initial marking is such that there is only one token in a start place and zero tokens elsewhere, and (4) there is no path in the net graph that starts at one output of a split transition and ends at the same transition without joining with any path that starts at the other output of the same transition.

We show this in two steps:

1. Every k-safe net can be translated to a standard k-safe net.
2. Every standard k-safe net can be expressed by an expression constructed with the four operators (; * + &).

It is easy to show the first three conditions (see Fig. I-2). A split transition with *n* outputs can be translated to *n-1* split transitions with two outputs, and a join transition with *n* inputs can be translated to *n-1* join transitions with two inputs. A k-safe net with the initial marking of *n* tokens in the start places *S* can be translated to a k-safe net by adding a new place *p* and a new *n*-output split transition *t*, where (1) *p* is the new start place with one token, (2) *p* is the only input place of *t*, (3) the original start places *S* are the output places of *t*, and (4) *n* tokens in *S* are removed.

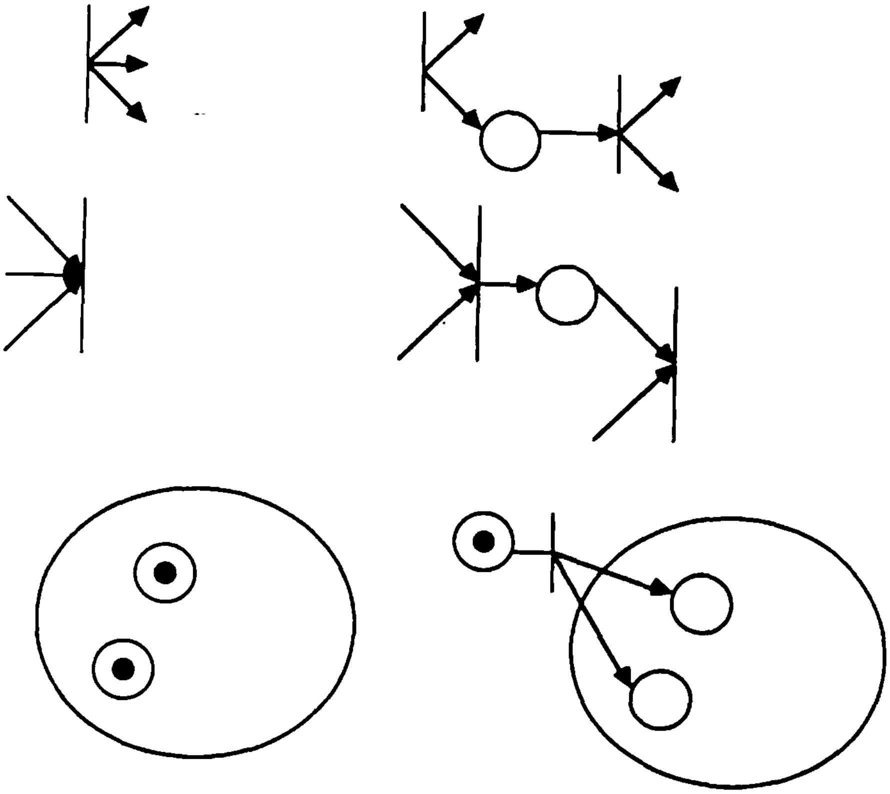


Figure I-2: Standard k-safe nets

For the fourth condition of standard k-safe nets, all decisions can be made at the first place. All possible cases of k-safe nets have to be considered. Split transitions in k-safe nets can be categorized into three possible cases 1-3:

- No cycle: There is no path that starts at one split transition and ends at the same transition.
- Conserved cycle: A path that starts at one output of a split transition and ends at the same transition joins with a path that starts at the second output of the same transition. The number of token is conserved.
- K-bounded cycle: The maximum number of firings of a split transition with an unconserved cycle is bounded by k.

One case left out from Petri nets is that the maximum number of firings of a split transition is unbounded. A Petri net with this condition is unsafe. A split transition with no cycle or conserved cycles satisfies the fourth condition of standard k-safe nets. A split transition with a k-bounded cycle can be translated into k split transitions with no cycles. The first step shows that a k-safe net can be translated into a standard k-safe net.

The second step will show that every standard k-safe net can be expressed by a safe DPE. In order to simply the proof, we assume the given standard k-safe net is an 1-safe net, which satisfies two conditions:

- 1-safe net: The maximum number of tokens in a place is bounded by one.

This assumption is reasonable because every k-safe net can be translated to a 1-safe net, and every 1-safe net can be translated to an 1-safe nets.

A standard 1-safe net can be expressed by a safe DPE. Split transitions in standard 1-safe net can only be in one of two cases: (1) no cycles and (2) conserved cycles.

In the case of no cycles, a DPE can be constructed by parsing the 1-safe net from the start place to the ends of the net according to the rules: (1) translate two sequential transitions a, b into expression a;b, (2)translate transitions a, b, c,.. starting from a multi-output place into expression a+b+c+..., and (3) translating the outputs (only two) of a split transition into two expressions related with (&). The first expressionn is constructed by parsing the sub-net starting from the first output of the split transition to the end according to the rules. The second expression is constructed by parsing the sub-net starting from the second output of the split transition but stop at the events that already appears in the first expression.

In the case of conserved cycles (*), the proof is the same as that a finite state machine can be denoted by a regular expression.

II. Extended DPEs: a Subset of Extended Petri Nets

This is obvious, since (1) the subclass of general DPEs is quivalent to Petri nets (2) a general DPE is a extended DPE, and (3) every extended DPE can be represented by an extended Petri net.