Modeling Regular and Safe Concurrent Processes
(Technical Summary)

Wenwey Hseush^{+}
Timothy S. Balraj^{++}
Gail E. Kaiser^{+++}

Columbia University
Department of Computer Science
New York, NY 10027
(212) 854-8123 hseush@cs.columbia.edu
Fax: (212) 666-0140

22 August 1990

CUCS-039-90

Abstract

The authors have previously described the use of data path expressions and predecessor automata in debugging concurrent systems. In this paper we examine the relationship of these models to two traditional models of concurrent processes: pomset languages and *k*-safe Petri net systems. We explore the regularity and safety of the concurrent languages described by each of the four models. Our main result is the equivalence of regular safe pomset languages and the languages described by safe data path expressions, safe predecessor automata and *k*-safe Petri net systems.

keywords: automata, debugging, formal languages, path expressions, Petri nets, synchronization

$^{+}$Hseush is supported in part by the Center for Telecommunications Research. $^{++}$Balraj is supported by DARPA project grant F29601-87-C-0074 and by grants from Citicorp and IBM. $^{+++}$Kaiser is supported by National Science Foundation grants CDA-8920080, CCR-8858029 and CCR-8802741, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

1. Introduction and Motivation

We are motivated by our application domain, parallel debugging, in which the programmer describes in a debugging language some anticipated partial orderings among run-time events, and then the debugger recognizes whether or not these partial orderings actually occur during execution. Thus we are concerned with languages to characterize both concurrency and computation, and automata to recognize these behaviors. We model regular, safe concurrency by employing pomsets [Pratt 86, Pratt 84] to formally define the regular and safe class of pomset languages. We contrast to regular, unsafe pomset languages, non-regular, unsafe pomset languages, and non-regular, safe pomset languages. We present data path expressions (DPEs) and predecessor automata (PAs) as the formal language and automata characterizing the debugging language and recognizer, respectively, and distinguish between safe and unsafe DPEs and between safe and unsafe PAs.

We previously introduced DPEs [Hseush 88] and PAs [Hseush 90]. The first paper demonstrated the utility of DPEs for defining interesting behaviors, such as race conditions, that often imply program errors. The second presented our approach to implementing a DPE debugger by using PAs generated from the DPEs to recognize run-time behaviors. We also characterized the power of a hierarchy of DPE languages in terms of Petri net models [Peterson 81, Reisig 85], and singled out the safe DPE languages, characterized by k-safe Petri net systems, as the class of debugging languages that could be implemented using PAs. The important contribution of this paper is to prove these four kinds of concurrency models — regular safe pomset languages, safe data path expressions, safe predecessor automata and k-safe Petri net systems — to be equivalent[1]. The relations among these four models are illustrated in Fig. 2-1, and Fig. 2-2 indicates what we prove.

2. Four Models

2.1. Regular and Safe Pomset Languages

2.1.1. Pomset Definitions

The following definitions are adapted from [Pratt 86].

Definition 1: A labeled partial order (*lpo*) is a 4-tuple $(V, \Sigma, <, \mu)$, where V is a set of event occurrences, Σ is a set of event types, $<$ (the causal precedence relation) is an irreflexive transitive binary relation, and $\mu: V \rightarrow \Sigma$ labels the event occurrences of V with event types from Σ.

Definition 2: A pomset $[V, \Sigma, <, \mu]$ is the isomorphism class of an lpo $(V, \Sigma, <, \mu)$.

Definition 3: A tomset $[V, \Sigma, <, \mu]$ (also called a string) is a pomset with a total order. That is, $\forall p,q \in V, p \neq q$, such that either $p < q$ or $q < p$.

Definition 4: A pomset language (also called a process) is a set of pomsets. A tomset language, or just language as the term is traditionally employed, is a set of strings.

[1]We restrict the proofs to the unambiguous set of regular safe pomset languages, but this does not affect the modeling power.

Figure 2-1: Relations Among four Models

Figure 2-2: What We Will Prove

2.1.2. Some Pomset Operations

In the following definitions, P, Q and R are pomset languages, and $p=[V_p,\Sigma_p,<_p,\mu_p]$, $q=[V_q,\Sigma_q,<_q,\mu_q]$ and $r=[V_r,\Sigma_r,<_r,\mu_r]$ are pomsets. The empty pomset $\varepsilon=[\emptyset,\emptyset,\emptyset,\emptyset]$. Five operators, *choice* (+), *concatenation* (;), *Kleene closure* (*), *concurrence* (||) and *concurrent closure* (@) are taken from Pratt [Pratt 86]. Our concurrence operator (&) allows synchronization of events, which Pratt's concurrence operator (||) does not. Pratt does not have a renaming operator. Some definitions are given.

Pratt's concurrence (||) : $P\|Q=\{[V,\Sigma,<,\mu] | \exists p\in P, q\in Q,$ such that $V=V_p\cup V_q, \Sigma=\Sigma_p\cup\Sigma_q, <\,=\,<_p\cup <_q, \mu=\mu_p\cup\mu_q\}$.

Concurrence (&) : An event occurrence homomorphism $h_{p,r}$ is a mapping from V_p to V_r such that $\mu_r h_{p,r}=\mu_p$.

$P\&Q=\{\, r \mid \exists p\in P \wedge q\in Q$
$\wedge\ \exists h_{p,r}, h_{q,r}$ such that $h_{p,r}(<_p)\subseteq <_r \wedge h_{q,r}(<_q)\subseteq <_r\}$.

Renaming Λ : $\Lambda_{a\rightarrow b}(p)=[V_p,\Sigma'_p,<_p,\mu'_p]$, where,
$\Sigma'_p=\Sigma_p\backslash\{a\}\cup\{b\}$,
$\mu'_p=\{(v,\sigma)|(\sigma\neq b\wedge(v,\sigma)\in\mu_p)\vee(\sigma=b\wedge(v,a)\in\mu_p)\}]$.
$\Lambda_{a\rightarrow b}(P)=\{\Lambda_{a\rightarrow b}(p) \mid p\in P\}$.

Concurrent Closure (@) : $P@=\{\varepsilon\}+P+P\|P+\ldots$

2.1.3. Regular Pomset Languages

Definition 5: A temporal snapshot (or just snapshot) of a pomset $p=[V,\Sigma,<,\mu]$ is a 5-tuple $s=[V,\Sigma,<_c,<_t,\mu]$, where $<_c$ is $<$, the causal relation, and $<_t$, called the temporal relation, is a total order over V, such that $< \subseteq <_t$. Let **Pomset**(s) = p, the pomset of the snapshot s.

Definition 6: The temporalization of a pomset p, denoted $\alpha(p)$, is the set of all possible snapshots of p. The temporalization of a pomset language L, $\alpha(L)$, is the union of the temporalizations of all pomsets in L.

The definition of temporalization is similar to Pratt's linearization [Pratt 86] except that, when a particular linear order is enforced, the former preserves the causal relations and the latter does not. One example in parallel debugging is that the debugger receives, in a linear order, a set of events, which occur in a partial order. To avoid losing the information of the partial order, the debugging system can be designed so that each event arrival carries the information about the occurrence of the event and the events that causally precede it.

A snapshot $s=[V,\Sigma,<_c,<_t,\mu]$ of a pomset p can be represented by a sequence of event-predecessor pairs, *EP(s)*= $(\mu(e_0)\ \mu(P_0))$, $(\mu(e_1)\ \mu(P_1))$, . . . ,$(\mu(e_n)\ \mu(P_n))$, where (i) e_0, e_1, . . . e_n is the total order described by $<_t$ and (ii) $\forall w \in P_i, w$ immediately precedes e_i in the causal order ($<_c$), $0 \le i \le n$. An event-predecessor pair $(\mu(e)\ \mu(P))$ carrys the information about the event type of the occurrence and the event types of its immediate and causal predecessors. For example, given a pomset that $V=\{v_1,v_2,v_3\}$, $\Sigma=\{a,b,c\}$, $<=\{(v_1 v_2),(v_1 v_3)\}$, and $\mu=\{(v_1, a), (v_2, b), (v_3, c)\}$, a snapshot s is $[V,\Sigma, <, <_t, \mu]$ that $<_t=\{(v_1\ v_2), (v_1\ v_3), (v_2, v_3)\}$. *EP*(s) is $(a\ \emptyset)\ (b\ \{a\})\ (c\ \{a\})$, which says that event occurrence *a* without any predecessor is received first, event *b* that has a predecessor *a* and event *c* that also has a predecessor *a* are received successively.

Definition 7: The event-predecessor language (EP language) of a pomset p, denoted *EP*(p), is $\{EP(s) \mid s \in \alpha(p)\}$, which is a string language over the symbol set $\Sigma_{EP} = \{(\mu(e)\ \mu(P)) \mid e \in V \wedge P \in 2^{\Sigma}\}$.

The EP language of a pomset language L, denoted *EP(L)*, is the union of *EP*(p) for all $p \in L$.

For example, p is *(a;b & c)*. *EP(p)* is $\{(a\ \emptyset)\ (b\ \{a\})\ (c\ \emptyset),\ (a\ \emptyset)\ (c\ \emptyset)\ (b\ \{a\}),\ (c\ \emptyset)\ (a\ \emptyset)\ (b\ \{a\})\}$. We can simply treat each distinct event-predecessor pair as a distinct symbol (*A*, *B* and *C*). *EP(p)*={*ABC*, *ACB*, *CAB*}, where $A=(a\ \emptyset)$, $B=(b\ \{a\})$ and $C=(c\ \emptyset)$.

There are two problems for *EP(s)*, where *s* is a snapshot of *p*. (1) *EP(s)* might ambiguously describe more than one pomset, since only event types are coded in the predecessor set *P*. The disambiguity rule *Most Recent Input* (MRI) is introduced to unambiguously describe a unique pomset. Under the MRI rule, a predecessor event $w \in P_i$ of e_i indicates the most recent event with the same label (event type) in the input sequence preceding e_i. (2) Even though a representation can uniquely describe a pomset using MRI rule, *EP(s)* doesn't always imply *p*. For example, *p* is $(a;b \parallel a;c)$. *EP(p)* describes $\{p, q\}$, where q is $((a;b \ \&\ a;c) \parallel a)$

Definition 8: A refinement of a pomset $p=[V,\Sigma,<,\mu]$ is a pomset $q=[V,\Sigma',<,\mu']$, such that the size of Σ' is finite and $\forall a,b \in V, a \neq b, \mu'(a)=\mu'(b)$ implies $\mu(a)=\mu(b)$.

A refinement of a pomset language L is the union of refinements for all p in L.

For example, a pomset p is $(a;b \parallel a;c)$. A refinement p' is $(a;b \parallel a';c)$, such that *EP(p')* describes only p'. For the pomset language $L=(a;b)@$, there doesn't exist a finite refinement such that *EP(L)* implies L.

Definition 9: A pomset language L is regular iff there exists a refinement L' of L, such that (i) *EP(L')* implies L' and (ii) *EP(L')* is regular.

Informally, we say a pomset language L is regular iff the temporalization of L is regular. For example, the temporalization of pomset language $(a\&b)*$ is

$((a\ \emptyset);(b\ \emptyset) + (b\ \emptyset);(a\ \emptyset));((a\ \{a\ b\});(b\ \{a\ b\}) + (b\ \{a\ b\});(a\ \{a\ b\}))*$. If we replace $(a\ \emptyset)$, $(b\ \emptyset)$, $(a\ \{a\ b\})$ and $(b\ \{a\ b\})$ with A, B, C and D, respectively, we have a language $(A+B);(C+D)*$, which is regular. The temporalization of pomset language $(a\&b)@$ is a language with the same number of $(a\ \emptyset)$ and $(b\ \emptyset)$, which is non-regular.

Definition 10: A pomset language L is unambiguous iff *EP(L)* implies L.

We focus on the unambiguous regular pomset languages. The refinement process that maps an ambiguous pomset language into an unambiguous pomset language can be done by a pre-processing machine. The details are outside the scope of the paper, since we are only interested in the modeling powers of the formalisms.

2.1.4. Safe Pomset Languages

Definition 11: Given a pomset $p = [V, \Sigma, <, \mu]$, a cut $\mathbf{C}$ is a subset of V, such that $(\forall s,t \in \mathbf{C}, s \neq t)\ (s,t), (t,s) \notin <$. $|\mathbf{C}|$ is the size of $\mathbf{C}$.

Definition 12: A pomset language L is *safe*, iff $\exists k \in \mathbf{N}, \forall p \in L$, for every cut $\mathbf{C}$ of p, such that $|\mathbf{C}| \leq k$.

2.2. Predecessor Automata

A *predecessor automaton* (PA) is a model of a concurrent system, whose input is a sequence of events which are related by a causal partial ordering. Each event carries information about the events immediately preceding it in the causal ordering (its *predecessor events*). The system can be in one of a finite number of internal states and move from one state to another by examining the input event along with the information of its predecessor events; thus the same input event with different predecessor events would result in different state transitions.

Definition 13: A PA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*.
- Σ is a finite set of *events*.
- δ is the *transition function* mapping $Q\times(\{\varepsilon\}\cup(\Sigma\times 2^{\Sigma}))$ to Q, where ε is a distinguished event representing an internal (empty) transition. ε has no predecessors. δ can be viewed as the set of edges between states, each labeled by an event and its predecessor set (or by ε).
- 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 a finite state automaton (FSA) except for the transition function δ, which takes into consideration both the events and their predecessors. The predecessor set is finite, and thus the PA is a finite system. An important distinction between PAs and FSAs is that concurrent composition of two PAs preserves causal independence, while the concurrent composition of two FSAs loses this information [Hseush 90].

A PA is represented graphically by a directed labeled graph. The nodes represent the states of the PA, and the edges are labeled with event-predecessor pairs. The labeled edges represent state transitions. If $q_1, q_2 \in Q$, $e \in \Sigma$, $P \in 2^{\Sigma}$, and $\delta(q_1, (e\ P)) = q_2$, then there is an edge labeled $(e\ P)$ from the node representing q_1 to that representing q_2. If P is the empty set $\emptyset$, we write the label as $(e\ .)$. $(e\ .)$ indicates event e has no causal predecessors, while ε represents a transition taken without an input event. Fig. 2-3 gives an example.

Figure 2-3: A safe PA and an unsafe PA

An input to a PA is a sequence of event-predecessor pairs, which is a snapshot of a pomset. A PA moves from one state q_1 to another state q_2 on an input $(e\ p)$, according to the transition function $\delta(q_1, (e\ p)) = q_2$.

Definition 14: A snapshot s is accepted by a PA iff *EP(s)* causes the PA to move from the initial state to a final state. A pomset p is accepted by a PA iff the PA accepts *EP(p)*. A pomset language L is accepted by a PA iff the PA accepts only *EP(L)*.

Definition 15: A PA, M, is complete iff for any snapshot s accepted by M, every other snapshot t of **Pomset**(s) is accepted by M.

We consider only complete PAs.

Definition 16: A PA is *safe* iff the pomset language it accepts is safe.

Lemma 17: A PA is safe iff for every cyclic path of transitions, $q_1, q_2, \ldots, q_n$, such that $\delta(q_1, (e_1\ P_1)) = q_2$, $\delta(q_2, (e_2\ P_2)) = q_3$, ..., $\delta(q_n, (e_n\ P_n)) = q_1$, in the PA, the following conditions hold.

- $\forall i,\ 1 \le i \le n,\ P_i \ne \emptyset$, and $P_i \subseteq \{e_1, e_2, \ldots, e_n\}$.
- For any sub-path of the cycle, $q_i, q_{i+1}, q_{i+2}, \ldots, q_{j-1}, q_j$, such that $\delta(q_i, (e_i\ P_i)) = q_{i+1}$, $\delta(q_{i+1}, (e_{i+1}\ P_{i+1})) = q_{i+2}$, ..., $\delta(q_{j-1}, (e_{j-1}\ P_{j-1})) = q_j$, where $e_i = e_{j-1}$, $e_i \in P_{i+1} \cup P_{i+2} \cup \ldots \cup P_{j-1}$.

Proof: Since the sets of states of a PA is finite, the only way that unbounded parallelism can be produced is by a cycle. The first condition says that the events in the cycle (except those in the first iteration) have predecessors that are events occurring in the cycle. This ensures that no event outside the cycle can be the predecessor of more that one occurrence of an event inside the cycle, and hence the cycle does not produce unbounded parallelism at a node outside the cycle. The second condition says that the events (except those in the last iteration) must be the predecessors of events in the cycle. This ensures that every concurrent fork in the cycle is matched by a join, and so there is no unbounded parallelism at a node within the cycle.

Fig. 2-3 shows a safe PA and an unsafe PA.

2.3. Data Path Expressions

Path expressions were first used as a formalism for describing concurrent processes by Campbell and Habermann [Campbell 74] and for debugging by Bruegge [Bruegge 83]. Data path expressions (DPEs) extend the formalism by allowing concurrency and introducing a renaming operator.

2.3.1. Definition

A DPE is an expression over a set of events $\Sigma \cup \{\varepsilon\}$ using six operators: concatenation (;), choice (+), Kleene closure(*), concurrence (&), concurrent closure (@) and renaming Λ.

Definition 18: A data path expression is defined recursively as follows (by a finite number of applications of the recursive step):

i) ε is a DPE and $\forall e \in \Sigma$, e is a DPE.

ii) If $\mathbf{r}$ and $\mathbf{s}$ are DPEs, then so are $\mathbf{r}+\mathbf{s}$, $\mathbf{r};\mathbf{s}$, $\mathbf{r}^*$, $\mathbf{r}\&\mathbf{s}$, $\mathbf{r}@$ and $\Lambda_{e \to e'}(\mathbf{r})$.

Each DPE represents a pomset language. The DPE operators (+ ; * & @ Λ) are analogous to the pomset operators described above. The DPE ε represents the empty pomset language $\{\varepsilon\}$, and the atomic DPE consisting of a single event e corresponds to the atomic pomset language $\{e\}$. If $\boldsymbol{P}(\mathbf{r})$ is the pomset language corresponding to the DPE $\mathbf{r}$, then the DPEs $\mathbf{r}+\mathbf{s}$, $\mathbf{r};\mathbf{s}$, $\mathbf{r}^*$, $\mathbf{r}\&\mathbf{s}$, $\mathbf{r}@$ and $\Lambda_{e \to e'}(\mathbf{r})$ correspond to the pomset languages $\boldsymbol{P}(\mathbf{r})+\boldsymbol{P}(\mathbf{s})$, $\boldsymbol{P}(\mathbf{r});\boldsymbol{P}(\mathbf{s})$, $\boldsymbol{P}(\mathbf{r})^*$, $\boldsymbol{P}(\mathbf{r})\&\boldsymbol{P}(\mathbf{s})$, $\boldsymbol{P}(\mathbf{r})@$ and $\Lambda_{e \to e'}(\boldsymbol{P}(\mathbf{r}))$, respectively.

The pomset languages that DPEs can express are not necessarily regular. For example, a DPE, *(a&b)@*, is equivalent to $a^{[n]}\&b^{[n]}$, which is not regular. $a^{[n]}$ is *a&a&...&a* for *n* times.

Definition 19: A DPE is safe iff it describes a safe pomset.

Lemma 20: A DPE is *safe* iff (;), (+), (*), (&) and (Λ) are the only operators used in the DPE.

Proof: Since a DPE is finite, the only source of unsafe concurrency is the @ operator. Eliminating it ensures safety. From theorems 23 and 25, we can show that every safe DPE can be built up by using only (;), (+), (*), (&) and (Λ).

The DPE *a;b & c;d* expresses a safe pomset language, and the DPE *a@* expresses an unsafe pomset language.

2.4. *K*-safe Petri Net Systems

Definition 21: A labeled Petri net system is a 6-tuple [S, T; F, M_0, Σ, μ], where

- S is the set of *places*,
- T is the set of *transitions*,
- F is the *flow function*, $\subseteq S\times T \cup T\times S$,
- M_0 is the *initial marking*, $M_0 \subseteq \mathbf{N}^S$,
- Σ is a set of *labels*, and
- μ is a mapping from T to Σ.

A marking of a Petri net corresponds to a distribution of *tokens* among the places of the net. When a transition *fires*, corresponding to an occurrence of the event by which the transition is labeled, one token is removed from each place at the start of an edge leading to the transition, and one token is added to each place at the end of an edge leading from the transition. A transition can fire only if there is at least one token in each of the places with edges leading to it.

The behavior of a Petri net system can be modeled by a set of *occurrence nets* [Best85 85]. Each occurrence net corresponds to a pomset; the behavior of a Petri net system corresponds to a pomset language. Since the number of tokens in a place is not limited, the pomset language described by a Petri-net system is not necessarily regular.

Definition 22: A k-safe Petri net system is a Petri net system such that for all markings reachable from the initial marking, there are at most k tokens in any one place.

Intuitively, a k-safe Petri net system corresponds to a safe, regular pomset language, since the number of concurrent events and the number of possible states are both bounded by $k^{|S|}$.

3. Equivalence Proofs

Theorem 23: Given a safe and regular (unambiguous) pomset language L, these exists a safe PA that accepts the language.

Proof: Since L is regular, there exists an FSA that accepts *EP(L)* (from definition 9). Every edge of the FSA is labeled with an event-predecessor pair, which is a symbol in the symbol set of *EP(L)*. The FSA actually is a PA, which accepts L. This automatically ensures that the PA is safe, if L is safe.

Definition 24: Given a pomset $p = [V, \Sigma, <, \mu]$, a line is a subset V' of V, such that (i) V' is totally ordered by $<$ and (ii) there does not exist a different subset V'' of V, which is totally ordered by $<$ and $V' \subset V''$.

Theorem 25: Given a safe PA, M, there exists a safe DPE that expresses the same pomset language described by M.

Proof: The proof is very long. Here we only briefly sketch it. Let the pomset language accepted by M be L. First, we construct an isomorphic FSA $M'=(Q, \Sigma_{EP}, \delta_{EP}, q_0, F)$ from $M=(Q, \Sigma, \delta, q_0, F)$ by treating each distinct event-predecessor (an edge label) as a distinct symbol. M' accepts *EP(L)*. Second, we transform the FSA M into a regular expression R that describes *EP(L)* (see [Hopcroft 79]). R is an expression over the symbol set Σ_{EP}. Remember there is also a PA, M_R, which is 1-1 corresponding to R. The next step is to convert R to a DPE. To do this, we need to consider the regular expression over Σ_{EP} in two cases: (1) If the expression R does not contain the (*) operator, we can generate a finite set of pomsets, $\{p_1, p_2, \ldots, p_n\}$, from M_R. The DPE corresponding to R is $(exp(p_1) + \ldots + exp(p_n))$. $exp(p_i)$, $1 \le i \le n$, is an expression $(exp'(l_1) \;\&\; \ldots \;\&\; exp'(l_k))$, where $l_j = \{e_1, \ldots, e_m\}$ $(1 \le j \le k)$, is a line of p_i. The $exp'(l_j)$ is $(e_1; \ldots; e_m)$, where $e_1 < \ldots < e_m$. (2) If the expression R has a sub-expression $E*$. From the safe conditions in lemma 17, we know the events inside the loop except those in the first iteration are isolated from the events outside the loop. E can be converted into a DPE, exp_E, recursively according to the cases (1) and (2). From E, we also know the predecessors, P_E, for the whole sub-expression $E*$. We replace $E*$ with an event-predecessor pair $(S\ P_E)$, where S is a distinct symbol. The expression now can be converted into a DPE, exp_R, according to case (1), since it has no (*). Finally, we replace any S in exp_R with $(exp_E)*$. The DPE expresses the language described by the PA. The DPE is safe, since we only use (;)(*)(+)(&) in the process.

Theorem 26: Given a safe DPE, there exists a k-safe Petri net system that describes the same pomset language expressed by the DPE.

Proof: We show this by constructing a k-safe Petri net for each step in (i) and (ii) of definition 18 with only (+) (;) (*) (&) and (Λ) (Lemma 20). See figure 3-1. A k-safe net system that describes the language expressed by the given DPE can then be constructed recursively, following the similar steps of constructing the DPE. All constructing steps are straightforward except the construction for the concurrence operator (&). Assume Net_p and Net_q are the net structures for p and q, respectively, and p and q synchronize at the events a, b and others (they both have the same label). For each homomorphism $h_{p,r}$ and $h_{q,r}$, we can construct $Net_{p\&q}$ by combining transition a in p and a in q into one (see figure).

Theorem 27: Finite k-safe net systems describe only safe and regular pomset languages

Proof: Given k-safe Petri net system, M. Since the number of tokens of a place is bounded by k and the number of places is finite, the number of transitions ready to fire is finite, which corresponds to the concurrent event occurrences in pomsets. Thus, M describes a safe pomset language. Since the set of possible markings (possible configurations of tokens) in M is finite, the system is a finite system. Thus it describes a regular pomset language.

Figure 3-1: Constructing *k*-safe Petri nets

References

[Best85 85] E. Best.
Concurrent Behaviour : Sequences, Processes and Axioms.
In S. D. Brookes and A. W. Roscoe and G. Winskel (editor), *LNCS 197 : Seminar on Concurrency*, pages 221-245. Springer-Verlag, 1985.

[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.

[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.

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

[Hseush 88] 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 90] Wenwey Hseush and Gail E. Kaiser.
Modelling Concurrency in Parallel Debugging.
In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Seattle, Washington, March, 1990.

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

[Pratt 84] V. Pratt.
The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial.
In S. D. Brookes and A. W. Roscoe and G. Winskel (editor), *LNCS 197: Seminar on Concurrency*, pages 180-196. Springer-Verlag, Berlin, 1984.

[Pratt 86] Vaughan Pratt.
Modeling Concurrency with Partial Orders.
International Journal of Parallel Programming 15(1):33-71, Febuary, 1986.

[Reisig 85] W. Reisig.
Petri Nets.
Springer-Verlag, 1985.