Semantics for Safe Concurrency

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

Krish Ponamgi*

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, NY 10598

6 February 1990

CUCS-002-90

Abstract

We present two systems for modelling safe concurrency. The first is an automata-based system, called *predecessor automata* (PAs). The second is a language-based system, called *Data Path Expressions* (DPEs). In a previous paper [Hseush 90], we presented a hierarchy of DPEs and described PAs as a mechanism for implementing the safe DPEs from the viewpoint of parallel debugging. In this paper, we formalize these two models and investigate the power of PAs and DPEs.

keywords: formal models, path expressions, Petri nets, synchronization, predecessor automata

+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 IBM and Citicorp. +++Kaiser is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from IBM, AT&T, Citicorp, DEC, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research. *Ponamgi is a co-op MS thesis student at Columbia University.

1. Introduction

Safe concurrency refers to the case of bounded parallelism and *unsafe concurrency* to the case of unbounded parallelism. The semantics of safe concurrency is characterized as a subclass of Petri nets, *k*-safe nets (also referred to in the literature as *k*-safe P/T-nets) [Reisig 85], in which for all markings reachable from the initial marking, there are at most *k* tokens in any one place. In Fig. 1-1, a 1-safe net models a system where a finite number of processes are created and run concurrently. In Fig. 1-2, an unsafe net models a computer program where processes are created in an infinite loop.

Figure 1-1: A safe net

Figure 1-2: An unsafe net

Safe concurrency is of fundamental importance since any real machine has only finite resources, even though programming systems often assume infinite availability of resources. Some programming language constructs designed for expressing concurrency allegedly permit unsafe concurrency, but actual parallelism is still bounded by the underlying computer systems due to resource limitations.

In our application of debugging parallel programs [Hseush 88, Hseush 90], the major concern is to model program behavior with safe concurrency and detect the situation of unsafe concurrency, to avoid the danger of running out of resources on a machine due to an unsafe program. A debugging language that can both express true concurrency and is easy to use must be provided by the system in order for users to describe the expected concurrent program behavior, which the debugger then compares to the actual program behavior during execution to detect program errors. Consequently, we need a high-level language for expressing concurrent behavior and an efficient recognizer for concurrent events. Most concurrent models, whether or not intended for application to debugging, fall into one of two categories: (1) able to express true concurrency, but difficult to use, or (2) easy to use, but expressing only interleaving semantics.

We present two systems for modelling safe concurrency. The first is an automata-based system, called *predecessor automata* (PAs). Predecessor automata extend finite state automata (FSAs) to recognize or generate partially ordered events. PAs are advantageous in three ways: (1) PAs express true concurrency. That is, the concurrent composition of two predecessor automata preserves the information of causal independence, while the concurrent composition of two finite state automata loses this information. (2) The class of languages accepted by PAs is a superset of that accepted by FSAs. That is, for every FSA, there exists a PA that is semantically equivalent to it. (3) PAs are structurally isomorphic to FSAs. That is, PAs and FSAs are structurally and mechanically similar. The property of isomorphism is advantageous in two ways: (i) Several results of FSA theory (*e.g.* the equivalence of deterministic FSAs and non-deterministic FSAs) can be applied to PAs. (ii) PAs are easy to manipulate and implement using current programming technology. A subclass of PAs, called safe PAs, will be defined and proven equivalent to *k*-safe Petri nets.[1]

The second system for modelling safe concurrency is a language-based system, called *Data Path Expressions* (DPEs), which is 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]. We have classified DPEs into five subclasses according to syntactic criteria, and characterized the semantics of each subclass using a hierarchy of extended Petri net models [Peterson 81] (see Fig. 1-3). Extended Petri nets are equivalent to Turing machines [Hack 75, Thomas 76]. The characterization demonstrates the power of DPEs for modelling true concurrency. In this paper, the third subclass, safe DPEs, will be proved equivalent to *k*-safe Petri nets.

DPEs were originated for describing concurrent program behavior. In our DPE taxonomy, the

[1]The PA hierarchy is orthogonal to the Chomsky hierarchy [Chomsky 56, Chomsky 59], since the PA hierarchy classifies languages according to concurrent complexity, while the Chomsky hierarchy classifies them according to sequential complexity.

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 1-3: DPE Hierarchy

first subclass expresses only sequential behavior. The second subclass expresses limited concurrency, in which process splitting (*e.g*, fork or para-do) is not permitted following a program branch (*e.g*, if-then-else), while program branching is permitted following a process split. 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.

In a previous paper [Hseush 90], we presented this hierarchy of DPEs and described predecessor automata as a mechanism for implementing the safe DPEs from the viewpoint of parallel debugging. In this paper, we formalize these two models and investigate the power of PAs and DPEs.

2. Predecessor Automata

A *predecessor automaton* (PA) is a mathematical model of a concurrent system, with sequential inputs of partially ordered events. The partial ordering is described by a sequence of events along with information about the events causally preceding them (called their *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.

2.1. Definition of PAs

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 $Q\times(\{\varepsilon\}\cup\Sigma\times\mathbf{N}^{\Sigma})$ to Q, where ε is a distinguished event representing an internal transition.

- 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 δ, which not only carries the information about the expected events, but also the information about their predecessors.

The *predecessor set* P_e ($\in \mathbf{N}^{\Sigma}$) of an event e is a multiset of events $\{w_1\ \ w_2\ \ \ldots\ \ w_n\}$, $n \geq 0$, where (1) $w_i \in \Sigma$ and (2) w_i is an immediate causal predecessor of e, $1 \leq i \leq n$. When P_e is an empty multiset (*i.e.*, $P_e = 0^{\Sigma}$), e is an *original* event.

A PA is represented graphically by a directed labelled graph. The nodes represent the states of the PA, and the edges are labelled with event-predecessor pairs. The labelled edges represent state transitions. If $q_1, q_2 \in Q$, $e \in \Sigma$, $P \in \Sigma\times\mathbf{N}^{\Sigma}$, and $\delta(q_1, (e\ P)) = q_2$, then there is an edge labelled $(e\ P)$ from the node representing q_1 to that representing q_2. If P is the empty multiset (0^{Σ}), we write the label as $(e\ .)$. Note that $(e\ .)$ indicates event e has no causal predecessors, while ε represents a transition taken without an input event. If $\delta(q_1, \varepsilon) = q_2$, then there is an edge labelled ε from the node representing q_1 to that representing q_2. This is known as an ε-transition. One example is shown in Fig. 2-1.

Figure 2-1: An example of PA

The input to a PA is not a sequence of events, but a sequence of event-predecessor pairs, $(e_0\ P_0)$, $(e_1\ \ P_1)$, ..., $(e_n\ \ P_n)$, where $e_i \in \Sigma$, $P_i \in \mathbf{N}^{\Sigma}$, $0 \leq i \leq n$.

A PA moves from one state q_1 to another state q_2 on an input (e p), according to the transition function $\delta(q_1, (e\ p)) = q_2$. That is, a move is made by examining the incoming event along with its predecessor set. When $\delta(q_1, (e\ p))$ is unspecified, the PA moves from q_1 to an error state. In the example shown in Fig. 2-1, one possible input sequence accepted by the automaton is `(a .) (b .) (c {a}) (d {b, c})`. The state of the automaton will change from state 1 to 2 with `(a .)`, from 2 to 5 with `(b .)`, from 5 to 6 with `(c {a})`, and from 6 to the final state with `(d {b, c})`. The input sequence `(a .) (b .) (c {a}) (d {b})` will lead to an error state, since the transition δ(6, `(d {b})`) is unspecified.

2.2. Semantics of PAs

Although PAs and FSAs are structurally and mechanically similar, their semantics are very different. The difference is best described in terms of the semantics of inputs. Since the term *event* sometimes refers to the type of an event and sometimes refers to the occurrence of an event, we use the terms *event type* and *event occurrence*. An input sequence to an FSA is a total order of event occurrences, where the input order is the occurrence order. The FSA moves from one state to another state by examining the type of the input event.

Conceptually, an input sequence to a PA is a *pomset* $[V, \Sigma, <, \mu]$ [Pratt 84][2], where V is a set of event occurrences, Σ is a set of event types, $<$ (called the relation of *causality*), is an irreflexive transitive binary relation, and $\mu: V \rightarrow \Sigma$ labels the event occurrences of V with event types from Σ. A pomset can be represented by a sequence of pairs $(e_0\ P_0)$, ..., $(e_n\ P_n)$, where e_i is an event occurrence and P_i is the predecessor set of e_i, $0 \le i \le n$. An input pair $(e_i\ P_i)$ indicates the occurrence of event e_i and for every $w \in P_i$, an event of type w causally precedes e_i.

Formally, an input sequence to a PA is a sequence of pairs $(\mu(e_0)\ \mu(P_0))$, ..., $(\mu(e_n)\ \mu(P_n))$, where the sequence $(e_0\ P_0)$, ..., $(e_n\ P_n)$ represents a pomset, and $\mu(e_i)$ is the event type of e_i and $\mu(P_i)$ is the set of event types for e_i's immediate causal predecessor events.

An input sequence is *legal*, if and only if for all $w \in P_i$, $0 \le i \le n$, there exists e_j, $j \neq i$, such

[2] A *partial order* is a set $(V, <)$ where $<$ is an irreflexive transitive binary relation on a vertex set V. A *labelled partial order* or lpo is a 4-tuple $(V, \Sigma, \mu, <)$ where $(V, <)$ is a partial order and $\mu: V \rightarrow \Sigma$ labels the vertices of V with symbols from an alphabet Σ.

Two lpo's $(V, \Sigma, <, \mu)$ and $(V', \Sigma', <', \mu')$ are *isomorphic* when there exists a bijection $\tau: V \rightarrow V'$ such that for all u in V, $\mu(u) = \mu'(\tau(u))$, and for all $u, v \in V$, $u < v$ just when $\tau(u) <' \tau(v)$.

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

that $e_j = w$. That is, an event can occur only when all its predecessor events have occurred. A legal input sequence is *stable*, if and only if for all $w \in P_i$, $0 \le i \le n$, there exists e_j, $j < i$, such that $e_j = w$. That is, the (total) ordering of the inputs preserves the partial ordering of the event occurrence. Unfortunately, a stable input sequence might ambiguously describe more than one pomset. For example, the sequence (a .) (b {a}) (a {b}) (c {a}) specifies two pomsets. (Fig. 2-2).

Figure 2-2: Two possible pomsets

The disambiguity semantics *Most Recent Input* (MRI) is introduced in order for an input sequence to unambiguously describe a unique pomset. Under the semantics of MRI, a stable input sequence, where $\forall w \in P_i$, $0 \le i \le n$, $\exists e_j$, $j < i$, $e_j = w$, describes the partial order that there exists no such k, $j < k < i$, such that $e_k = w$. That is, a predecessor event w in the predecessor set P_i of e_i indicates the most recent event with the same label (event type) in the input sequence preceding e_i. In the above example (Fig. 2-2), the input sequence describes the second pomset, where c causally precedes the second a, since the second a is the most recent one in the input sequence. If the first pomset, where c causally precedes the first a, is expected, the approach is to distinguish the second a from the first by renaming. The sequence can be (a .) (b {a}) (a' {b}) (c {a}) by renaming the second a to a'. For the reason of simplicity, in the rest of the paper, we assume all events in an input sequence are distinguishable.

A sequence $(e_0\ P_0)$, $(e_1\ P_1)$, ..., $(e_n\ P_n)$ is said to be accepted by a PA, $U = (Q, \Sigma, \delta, q_0, F)$, if $\delta(q_0, (e_0\ P_0)) = q_1$, $\delta(q_1, (e_1\ P_1)) = q_2$, ..., $\delta(q_{n-1}, (e_{n-1}\ P_{n-1})) = q_n$, where q_0 is the initial state and $q_n \in F$. The set of pomsets that can be accepted by the PA is represented by $L(U)$, the language accepted by U.

A PA is *legal* if and only if it accepts only legal input sequences. A PA is *stable* if and only if it accepts only stable input sequences. A PA is *unambiguous* if and only if it accepts only unambiguous input sequences.

In the rest of the paper, all cases we are going to discuss are assumed to be legal, stable and unambiguous.

2.3. Safe and Unsafe PAs

PAs can express safe concurrency as well as unsafe concurrency. The example in Fig. 2-3 shows how a PA describes a set of pomsets, which are of unbounded parallelism.

Figure 2-3: An unsafe PA and its possible inputs

Definition 1: A PA is *safe* if and only if for every cyclic path of transitions, $\delta(q_1, (e_1\ P_1)) = q_2$, $\delta(q_2, (e_2\ P_2)) = q_3$, ..., $\delta(q_n, (e_n\ P_n)) = q_1$, in the PA, the following conditions hold.

1. $P_1 \cup P_2 \cup \ldots \cup P_n \subseteq \{e_1, e_2, \ldots, e_n\}$, and
2. $P_i \neq \mathbf{0}^{\Sigma}$, $\forall i$, $1 \leq i \leq n$.

A PA describes an unsafe situation when one predecessor event w of an event e that occurs in a cyclic path does not occur in the cycle. The repeated occurrences of e will be bound to a single occurrence of w as their predecessor; thus unbounded parallelism can be created when the cycle of transitions is repeated. Cycling in PAs can mean ‘‘repeating concurrently’’, while it means only ‘‘repeating sequentially’’ in FSAs.

Theorem 2: Safe PAs describe only bounded parallelism.

Proof: Since the sets of states of a PA is finite, the only way that unbounded parallelism can be produced is by a cycle. Due to the conditions in definition 1, and under MRI semantics, any event occurrence in an iteration of a cycle can only be the predecessor of event occurrences in one iteration succeeding it, and any event occurrence outside the cycle can only be the predecessor of event occurrences in the first iteration of the cycle. Therefore, no event occurrence can be the predecessor of an unbounded number of event occurrences, and hence there cannot be unbounded parallelism.

2.4. Concurrent Composition

The concurrent composition of two predecessor automata preserves causal independence (*i.e.*, true concurrency), while the concurrent composition of two finite state automata loses this information.

The concurrent composition of two PAs, $(Q^1, \Sigma^1, \delta^1, q_0^1, F^1)$ and $(Q^2, \Sigma^2, \delta^2, q_0^2, F^2)$, is a function that creates a new PA $(Q, \Sigma, \delta, q_0, F)$, where

- $Q = Q^1 \times Q^2$,
- $\Sigma = \Sigma^1 \cup \Sigma^2$,
- $\delta([q_i\ q_j], (e\ P)) = [q_k\ q_l]$, if
  1. $j = l$ and $\delta^1(q_i, (e\ P)) = q_k$ or
  2. $i = k$ and $\delta^2(q_j, (e\ P)) = q_l$.
- $q_0 = (q_0^1, q_0^2)$
- $F = F^1 \times F^2$.

The composed PA preserves all possible states and all possible transitions as if the two original automata operate concurrently. An example is shown in Fig. 2-5. The concurrent composition of two finite state automata will lose the concurrency information, if the similar process is applied to FSAs. An example is shown in Fig. 2-4. The composed FSA preserves all possible states and all possible transitions by interleaving the first FSA with the second, but it loses the information about causal independency among events (*e.g.*, the information that event a is causally independent from event c is lost).

Figure 2-4: Concurrent composition of FSAs **Figure 2-5:** Concurrent composition of PAs

2.5. Semantic Relationship Between FSAs and Safe PAs

For every FSA, there exists a safe PA that is semantically equivalent to the FSA. Here, we say ‘‘semantically equivalent’’ instead of just ‘‘equivalent’’ because the FSA and the PA accept the same language but with different forms. FSAs accept sequences of totally ordered events, while PAs accept sequences of partially ordered events. A sequence of totally ordered events e_0, e_1, ...,e_n can be rewritten as a sequence (e_0 .), (e_1, {e_0}), ..., (e_n, {e_{n-1}}) without losing any information. Let ψ be the rewriting function. Using the same form of inputs, we can prove that the class of languages accepted by PAs is a superset of the class of languages accepted by FSAs. The proof hinges on showing that PAs can simulate FSAs. That is, for every FSA we can construct an equivalent safe PA.

Theorem 3: Let L be a set accepted by an FSA and ψ be the rewriting function. Then there exists a safe PA that accepts $\psi(L)$.

Proof: Given an FSA, $M = (Q, \Sigma, \delta, q_0, F)$, we can construct an equivalent PA by the following procedure:

1. Construct an equivalent FSA, $M' = (Q \times \Sigma, \Sigma, \delta', [q_0, e_0], F \times \Sigma)$, such that $\delta(q_i,e)=q_j$ implies $\forall e_k \in \Sigma$ ($\delta'([q_i,e_k],e)=[q_j,e]$) $\wedge$ ($\forall e' \neq e$ $\delta'([q_i,e_k],e') \neq [q_j,e]$). That is, all transitions terminating in a state of M' are labelled with the same event.

2. Construct an equivalent PA, $U = (Q \times \Sigma,\ \Sigma,\ \delta'',\ [q_0,\ e_0],\ F \times \Sigma)$, such that if $\delta'([q_i,e_k],e) = [q_j,e]$ then $\delta''([q_i,e_k],(e,\{e_k\})) = [q_j,e]$.

In the constructed PA, U, a transition from state $[q_1,\ e_1]$ to state $[q_2,\ e_2]$ is labelled $(e_2 \quad \{e_1\})$. Hence, every cycle is of the form $(e_2\ \ \{e_1\}),\ (e_3\ \ \{e_2\}),\ \ldots,\ (e_n\ \ \{e_{n-1}\}),\ (e_1\ \ \{e_n\})$, and the conditions of definition 1 hold. Therefore, U is a safe PA.

A PA is *sequential*, if and only if there exists an FSA that is semantically equivalent to the PA.

2.6. Structural Isomorphism Between FSAs and PAs

Every PA = $(Q,\ \Sigma,\ \delta,\ q_0,\ F)$, is isomorphic to an FSA $(Q,\ \Sigma',\ \delta',\ q_0,\ F)$, where there exists a bijection $\lambda{:}\{\varepsilon\} \cup \Sigma \times \mathbf{N}^{\Sigma} \rightarrow \{\varepsilon\} \cup \Sigma'$, such that $\lambda(\varepsilon) = \varepsilon$, $\delta'(q,\ \lambda((\sigma\ \ p_\sigma))) = \delta(q,\ (\sigma\ \ p_\sigma))$ and $\delta'(q,\ \varepsilon) = \delta(q,\ \varepsilon)$.

This structural relationship allows us to borrow some of the results of FSA theory to manipulate the structure of PAs. Since the relationship is only with regard to the structure of the automata, we are limited to using only the purely structural results of FSA theory.

The definition of a PA given in section is for a deterministic PA with ε-transitions. Analogous to FSAs, PAs also have both deterministic and non-deterministic variants, and may or may not allow ε-transitions. A non-deterministic PA (NPA) is a PA that allows zero, one or more transitions from a state on the same pair of input event and predecessor set (*i.e.*, $\delta : Q \times (\{\varepsilon\} \cup \Sigma \times \mathbf{N}^{\Sigma}) \rightarrow \mathbf{2}^{Q}$).

We can use the standard FSA algorithms [Hopcroft 79] to convert from a non-deterministic PA to a deterministic PA and to remove ε-transitions, via a three step process:

1. Given a PA, U, construct an isomorphism function λ, and the corresponding FSA, M, which is structurally isomorphic to U.

2. Use the standard algorithms on M to remove non-determinism and/or ε-transitions. The resultant FSA, M', accepts the same language as M.

3. Use the inverse of the isomorphism function λ^{-1}, to construct a PA, U', structurally isomorphic to M'. U' accepts the same language as U does.

Several other results of FSA theory can be applied to PAs, but are not discussed in this paper for lack of space. In particular, the authors are working on a hierarchy of PAs analogous to the Chomsky hierarchy [Chomsky 56, Chomsky 59]. We believe the PA hierarchy is orthogonal to the Chomsky hierarchy.

2.7. The Equivalence of Safe PAs and finite *k*-Safe Nets

In this section, we prove that the class of languages accepted by safe PAs is equivalent to the class described by finite *k*-safe nets. We do this in two steps, and we use C/E-systems as an intermediate stage in the first step.

The following definitions are adapted from Reisig [Reisig 85].

Definition 4: A quadruple $\Sigma = (B, E; F, C)$ is called a condition/event-system (C/E-system) iff

1. $(B, E; F)$ is a simple net without isolated elements, $B \cup E \neq \emptyset$.
2. $C \subseteq \mathbf{2}^B$ is an equivalence class of the reachability relation $R_\Sigma = (r_\Sigma \cup r_\Sigma{}^{-1})*$ where $r_\Sigma \subseteq \mathbf{2}^B \times \mathbf{2}^B$ is given by $c r_\Sigma c'$ iff there exist $c_0,...,c_n \in C$, $c = c_0$, $c' = c_n$, such that $c_i F c_{i+1}$ for $i = 0 \, . \, . \, . \, n-1$. C is called the *case class* of Σ.
3. $\forall e \in E, \exists c \in C$ such that ${}^\bullet e \subseteq c \wedge e^\bullet \cap c = \emptyset$.

Definition 5: Let $\Sigma = (B, E; F, C)$ be a C/E-system, let Γ be the set of all steps of Σ, and let $P = \{(c_1, G, c_2) \in C \times \Gamma \times C \mid c_1 r_\Sigma c_2\}$. Then the graph $\phi_\Sigma = (C, P)$ is called the *case graph* of Σ.

We now prove that safe PAs are at least as powerful as C/E-systems and finite *k*-safe nets.

Theorem 6: For every C/E-system there is a safe PA that accepts the same language.

Proof: Let $\Sigma = (B, E; F, C)$ be a C/E-system. Construct its case graph $\phi_\Sigma = (C, P)$. Now, if $(c_1, G, c_2) \in P$, and if $e_1, e_2 \in G$, then there exist c_3 and c_4 such that (c_1, G_1, c_3), (c_3, G_2, c_4), (c_1, G_2, c_4), $(c_4, G_1, c_2) \in P$ where $G_1 = G \setminus e_2$, $G_2 = G \setminus e_1$, $c_3 = c_1 \setminus {}^\bullet e_1 \cup e_1{}^\bullet = c_2 \setminus e_2{}^\bullet \cup {}^\bullet e_2$, $c_4 = c_1 \setminus {}^\bullet e_2 \cup e_2{}^\bullet = c_2 \setminus e_1{}^\bullet \cup {}^\bullet e_1$. (This is obvious from the definitions of cases and steps).

Remove arcs (c_1, G_1, c_3), (c_3, G_2, c_4), (c_1, G_2, c_4), (c_4, G_1, c_2) from P, and replace arc

(c_1, G, c_2) by $(c_1, G \setminus (e_1, e_2) \cup \lambda(\{e_1,e_2\}), c_2)$, where $\lambda : \mathbf{2}^E \rightarrow E'$ is a bijection and $E' \cap E = \emptyset$. Extend λ to $E \cup E'$ by defining $\lambda(\{e,e'\}) = \lambda(\{e',e\}) = \lambda(\{e\} \cup \lambda^{-1}(e'))$.

Repeat this process until all steps are singletons. The resultant case graph ϕ'_Σ is now isomorphic to an FSA.

Using the technique described in the proof of theorem 3, convert ϕ'_Σ to an equivalent FSA and then to a safe PA with transitions labelled in $E \cup E'$.

Now, λ^{-1} is used to produce a PA labelled with events and predecessors in E as follows. If $(e\ \{p\})$ is an event-predecessor pair and

1. $p \in E'$, replace the pair by $(e\ \ \lambda^{-1}(p))$.
2. $e \in E'$, add extra states to enumerate every possible sequence of $\lambda^{-1}(e)$, using the same predecessor set $\{p\}$ for each.

The resultant safe PA accepts the same language as the original C/E-system.

Theorem 7: For every finite k-safe net, there is a safe PA that accepts the same language.

Proof: A finite k-safe net has a finite set of possible markings and hence can be transformed into a finite C/E-system. The result follows from theorem 6.

Now we show that safe PAs cannot accept any language not described by a k-safe net.

Theorem 8: For every safe PA, there exists a finite k-safe net that describes the language it accepts.

Proof: The result follows directly from theorem 2.

3. Data Path Expressions

Data Path Expressions (DPEs) is a language-based formalism for modelling the behavior of concurrent systems. DPEs were originally designed as a high-level debugging language for users to describe the expected program behavior, where a style of problem-oriented debugging, called data path debugging, was advocated. For this historical reason, we use the term DPE even though there is nothing to do with data paths.

3.1. Definition of DPEs

A DPE is an expression over a set of events Σ. Five operators: sequencing (;), selection (+), repetition(*), concurrency (&) and concurrent closure (@) are employed (see Table 3-1).

A DPE, **r**, denotes a set of pomsets, **R**. The sets that DPEs denote are defined recursively by the following formation rules.[3]

1. ∅ is a DPE and denotes the empty set.
2. ε is a DPE and denotes the set {ε}.
3. For each *e* in Σ, *e* is a DPE and denotes the set {*a*}.
4. If **r** and **s** are DPEs denoting the sets **R** and **S**, respectively, then (**r+s**) (**r**;**s**), (**r***), (**r**&**s**), and (**r**@) are defined by using the operations of pomset algebra [Pratt 86].[4]
   - (**r+s**) denotes the union of of **R** and **S**, **R**∪**S**.
   - **r**;**s** denotes the concatenation of **R** and **S**, **R**;**S**.
   - **r*** denotes the Kleene closure of **R**, **R***.
   - **r**&**s** denotes the concurrence of **R** and **S**, **R||S**.
   - **r**@ denotes the concurrent closure of **R**, $\mathbf{R}^{+}$.

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, in which process splitting (*e.g*, fork or para-do) is not permitted following a

---

[3] The definition is similar to that of regular expressions by Hopcroft and Ullman [Hopcroft 79].

[4] $\mathbf{R}\|\mathbf{S} = \{\ a\|b \ \mid\ a \in \mathbf{R},\ b \in \mathbf{S}\}$, where $a\|b$, the concurrence of two pomsets, is defined as
$[V,\Sigma,<,\mu]\|[V',\Sigma',<',\mu'] = [V\cup V',\Sigma\cup\Sigma',\ <\cup<',\mu\cup\mu']$

$\mathbf{R};\mathbf{S} = \{\ a;b \ \mid\ a \in \mathbf{R},\ b \in \mathbf{S}\}$, where $a;b$, the concatenation of two pomsets, is defined as
$[V,\Sigma,<,\mu];[V',\Sigma',<',\mu'] = [V\cup V',\Sigma\cup\Sigma',\ <\cup<'\cup(V\times V'),\mu\cup\mu']$

$\mathbf{R}^{+} = \emptyset\cup\mathbf{R}\cup\mathbf{R}\|\mathbf{R}\cup \ldots$

<table>
<tr><th>Operator</th><th>Expression</th><th>Description</th><th>Pomsets</th></tr>
<tr><td>Sequencing</td><td>r;s</td><td>r causally precedes s.</td><td>R;S</td></tr>
<tr><td>Selection</td><td>r+s</td><td>Either r or s occurs, but not both.</td><td>$\mathbf{R}\cup\mathbf{S}$</td></tr>
<tr><td>Kleene closure</td><td>r*</td><td>ε + r + r;r + r;r;r + ...</td><td>R*</td></tr>
<tr><td>Concurrency</td><td>r&s</td><td>r and s occur causally independently.</td><td>$\mathbf{R \| S}$</td></tr>
<tr><td>Concurrency Closure</td><td>r@</td><td>ε + r + r&r + r&r&r + ...</td><td>$\mathbf{R}^+$</td></tr>
</table>

Table 3-1: Operators

program branch (*e.g*, if-then-else), while program branching is permitted following a process split. 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. Table 3-2 summarizes the DPE hierarchy, including equivalence proofs and related work.

<table>
<tr><th>Subclass</th><th>Sequential DPEs</th><th>Multiple DPEs</th><th>Safe DPEs</th><th>General DPEs</th><th>Extended DPEs</th></tr>
<tr><td>Expresses</td><td>sequential behavior</td><td>limited safe concurrency</td><td>safe concurrency</td><td>limited unsafe concurrency</td><td>unsafe concurrency</td></tr>
<tr><td>Syntax</td><td>dpe$_1$: EVENT
| (dpe$_1$)
| dpe$_1$ + dpe$_1$
| dpe$_1$; dpe$_1$
| dpe$_1$ *</td><td>dpe$_2$: dpe$_1$
| dpe$_2$ & dpe$_1$</td><td>dpe$_3$: EVENT
| (dpe$_3$)
| dpe$_3$ + dpe$_3$
| dpe$_3$; dpe$_3$
| dpe$_3$ *
| dpe$_3$ & dpe$_3$</td><td>dpe$_4$: dpe$_3$
| dpe$_4$ + dpe$_4$
| dpe$_4$ & dpe$_4$
| dpe$_4$ @</td><td>dpe$_5$: EVENT
| (dpe$_5$)
| dpe$_5$ + dpe$_5$
| dpe$_5$; dpe$_5$
| dpe$_5$ *
| dpe$_5$ & dpe$_5$
| dpe$_5$ @</td></tr>
<tr><td>Semantic Model</td><td>finite state automata</td><td>k-safe nets subset</td><td>k-safe nets</td><td>Petri nets [Garg 88]</td><td>extended Petri nets subset</td></tr>
<tr><td>Proof</td><td>Hopcroft & Ullman [Hopcroft 79]</td><td>Lauer & Campbell [Lauer 75]</td><td>in this paper</td><td>Garg [Garg 88]</td><td>Hseush & Kaiser [Hseush 89]</td></tr>
<tr><td>Example</td><td>open;(read+write)*; close</td><td>(a;s;b) & (c;s;d)</td><td>(enq;deq)+(enq&deq)</td><td>(fork;parent)* & (fork;child)@</td><td>(enq;update)@;deq; display</td></tr>
<tr><td>Limitations</td><td>no concurrency</td><td>no process splitting following program branching</td><td>no unbounded parallelism</td><td>no joining for unbounded parallelism</td><td>open question</td></tr>
<tr><td>Related Work</td><td>Generalized path expressions [Bruegge 85]</td><td>COSY [Lauer 81]</td><td>EBBA [Bates 83]</td><td>Concurrent regular expressions [Garg 88]</td><td></td></tr>
</table>

Table 3-2: DPE Hierarchy

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.

The second subclass, multiple DPEs, expresses only global-level concurrency, where no nested concurrency (&) is allowed. When the same event name appears in multiple subparts of the DPE, it is treated as a synchronization event.

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 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 unsafe program and the corresponding unsafe net mentioned in Fig. 1-2. Some programming examples are: (1) an unbounded number of messages may arrive at an object and each message activates a control thread for executing the same function without waiting for the prior activations to finish (*e.g*, process servers); 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 proved equivalent to Petri nets [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 or Petri nets. 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 (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.

3.2. Equivalence of Safe DPEs and Finite *k*-Safe Nets

Theorem 9: For every safe DPE, there exists an equivalent *k*-safe net.

Proof: By construction: Given a safe DPE, which uses only the four operators (; * + &), a *k*-safe net that recognizes the same language can be constructed by translating every operator with the associated events into a net graph according to the rules shown graphically in Fig. 3-1. Each sub-expression *r* is translated into a sub-net *Net(r)* along with an initial and a final place and an initial marking. The sub-nets compose as shown and the net corresponding to the whole expression is built up recursively. Since the number of tokens in each component is conserved, the number of tokens in the constructed net is conserved, and hence it is *k*-safe.

Theorem 10: For every C/E-system, there is a safe DPE that represents the language it accepts.

Proof: The proof is along the same lines as that of theorem 6. Let $\Sigma = (B, E; F, C)$ be a C/E-system. Construct its case graph $\phi_\Sigma=(C, P)$. Now, if $(c_1, G, c_2) \in P$, and if $e_1,e_2 \in G$, then there exist c_3 and c_4 such that (c_1, G_1, c_3), (c_3, G_2, c_4), (c_1, G_2, c_4), $(c_4, G_1, c_2) \in P$ where $G_1=G \setminus e_2$, $G_2=G \setminus e_1$, $c_3=c_1 \setminus {}^\bullet e_1 \cup e_1{}^\bullet = c_2 \setminus e_2{}^\bullet \cup {}^\bullet e_2$, $c_4=c_1 \setminus {}^\bullet e_2 \cup e_2{}^\bullet = c_2 \setminus e_1{}^\bullet \cup {}^\bullet e_1$. (This is obvious from the definitions of cases and steps).

Remove arcs (c_1, G_1, c_3), (c_3, G_2, c_4), (c_1, G_2, c_4), (c_4, G_1, c_2) from *P*, and replace arc (c_1, G, c_2) by $(c_1, G \setminus \{e_1, e_2\} \cup e_1\&e_2, c_2)$, where $e_1\&e_2$ is to be treated as if it were a single new event. By the semantics of the & operator, the resultant case graph accepts the same language as the original.

Repeat this process until all steps in the case graph are singletons.

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

Now, use the algorithm for reducing an FSA to the equivalent regular expression on the final case graph. The resultant expression uses the four operators (; * + &), i.e. a safe DPE which represents the same language accepted by the case graph, and thus by the original C/E-system.

Theorem 11: For every finite *k*-safe net, there exists a safe DPE that represents the language it accepts.

Proof: A finite *k*-safe net has a finite set of possible markings and hence can be transformed into a finite C/E-system. The result follows from theorem 10.

4. Conclusion

We have presented two systems for modelling safe concurrency, and proven them equivalent to finite *k*-safe nets (see Fig. 4-1). These two systems belong to a hierarchy of systems for describing concurrent behavior (see table 3-2), which we introduced in a previous paper [Hseush 90]. PAs are structurally isomorphic to FSAs, and can be manipulated using many of the same techniques used on FSAs. DPEs is a language-based formalism useful for describing concurrent behavior. We have employed DPEs as the description language in a parallel debugging system [Hseush 88].

Figure 4-1: Formalisms for Safe Concurrency

References

[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.

[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.

[Chomsky 56] N. Chomsky.
Three Models for the Description of Language.
IRE Transactions on Information Theory 2(3):113-124, 1956.

[Chomsky 59] N. Chomsky.
On Certain Formal Properties of Grammars.
Information and Control 2(2):137-167, 1959.

[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.

[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.

[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 89] Wenwey Hseush and Gail E. Kaiser.
Modeling Concurrency in Parallel Debugging.
Technical Report CUCS460, Computer Science Department, Columbia University, NY, NY, October, 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.

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

[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.

[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.
Technical Report STAN-CS-86-1113, Department of Computer Science, Stanford University, June, 1986.

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

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