Parallel Finite Automata for Modeling Concurrent Software Systems

P. David Stotts*
Department of Computer Science
CB 3175, Sitterson Hall
University of North Carolina
Chapel Hill, NC, 27599-3175

William Pugh
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Abstract

The class of parallel finite automata (PFA) is described that naturally expresses the interleaving parallelism inherent in Petri net notation without admitting the possibility of an infinite state space. The equivalence of this class to deterministic finite automata (DFA) is demonstrated using an algorithm for generating an equivalent nondeterministic finite automaton (NFA) from a PFA. A composition rule is given for constructing a PFA from a regular expression with the interleaving operator. Finally, the languages generated by this class are related to known classes of Petri net languages. Though the class PFA is equivalent in recognition power to the class DFA, the fact that DFAs are a structural subset of PFAs makes the PFA representation preferable for many applications requiring finite automata models. As an example, we discuss the usefulness of PFA notation as a structural model for hypertext, and non-linear interactive information networks in general.

1 Parallel Finite Automata

We present a class of finite automaton called *Parallel Finite Automata*, or *PFA*. Before getting to the formal definitions and application, it should be clearly stated that the contribution of this work is not primarily theoretical. Though the notation employed here does not appear to be generally known or used, the theory of finite state machines is among the best understood material in computing. The contribution of this work is in the utility of this particular notation and in the manner in which it fit into an existing software modeling development (see section 4). PFAs combine the modeling capabilities of Petri nets[1] without admitting the possibility of an infinite state space. The resulting automaton class is therefore a form of finite state machine, but one that is capable of directly expressing interleaving forms of parallelism without having it encoded into the meaning of states. The PFA notation leads to automata that are highly compact relative to the state space they encode (see section 3 under *notational succinctness*).

The notation we use to represent a PFA is similar to that commonly used for deterministic and nondeterministic finite automata (DFA and NFA respectively) but it is slightly modified for expressing parallel activity. The modifications essentially match the elements in a Petri net structure, but we have represented them with a notation that makes PFAs more recognizable as purely finite automata. In this section we explain the definition, representation, and behavior of a PFA. In section 2.1 we present a proof of equivalence between the DFA and PFA automaton classes. Section 2.2 demonstrates a technique for composing PFAs

*This work partially supported by the National Science Foundation under grants IRI–9007746 and IRI–9015439.

[1] We assume a familiarity with basic Petri net theory. An excellent introduction to the field can be found in the recent survey by Murata [9], and a detailed exposition of more advanced theory can be found in the book by Reisig [11].

directly from regular expressions with an interleaving operator. In section 3 we discuss several interesting modeling properties exhibited by PFAs and discuss their relationship to subclasses of Petri nets. In section 4 we conclude by discussing the use of PFAs in the software system architecture of a novel interactive information system called Trellis.

1.1 Formal descriptions

We begin with a formal definition of the members of class PFA, and follow that with a description of their representation and behavior. The notation and style is that commonly used in finite automata theory [7]. The formal development is straightforward, but presented in detail for completeness and clarity of understanding the execution of the transition diagrams in the ensuing examples.

Definition 1 Parallel finite automaton

A PFA M can be formally defined as a 7-tuple $M = (N, Q, \Sigma, \gamma, \delta, q_0, F)$ in which

N is a finite set of nodes
$Q \subseteq 2^N$ is a finite set of states
Σ is a finite input alphabet
γ: $2^N \times (\Sigma \cup \{\lambda\}) \to 2^{2^N}$ is the node transition function
δ: $Q \times (\Sigma \cup \{\lambda\}) \to 2^Q$ is the state transition function
$q_0 \in Q$ is the start state
$F \subseteq N$ is the set of final nodes

and where γ and δ are partial functions subject to the restrictions outlined in the following discussion.

In a PFA, there is no one-to-one correspondence between states and nodes in a transition diagram, as is common in other finite automata. Thus, the node transition function γ is used to generalize the notion of arc found in directed graphs. Specifically, if $((\{A, B\}, a), \{C, D, E\})$ is an element of γ, we say that a transition labeled a exists with source nodes A and B, and with target nodes C, D, and E.[2]

A state in a PFA is a set of nodes. During execution, the nodes composing the current state are termed *active*. The state transition function δ is then defined as follows. Initially, the set of active nodes for M is exactly q_0, the initial state. During execution of M, on seeing the input symbol[3] c in state q, the set of active nodes constituting the next state for M is any one of the sets in $\delta(q, c)$. More specifically, each state transition is defined based on the node transitions, as follows:

Definition 2 State transition rule (firing rule)

Given an state $q \in Q$ and an input symbol $c \in (\Sigma \cup \{\lambda\})$,

$$\delta(q, c) = \{ \ (q - m) \cup n \mid n \in \gamma(m, c) \ \text{ for } \ m \subseteq q \ \} \ .$$

We extend the operation of δ to strings from Σ^* using the function $\hat{\delta}$, as follows:

$$\begin{aligned} \hat{\delta}(q, \lambda) &= \delta(q, \lambda) \\ \hat{\delta}(q, \alpha a) &= \{ \ \delta(p, \lambda) \mid \text{ for some } r \in \hat{\delta}(q, \alpha), \ p \in \delta(r, a) \ \} \end{aligned}$$

Furthermore, the meaning of these functions applied to a *set* of states is simply the union of the results of application to the individual states in the set. In the ensuing discussion we use only the symbol δ and let the context of its use distinguish between the two functions.

[2] A normal directed graph expressed in this form would have singletons for both the source and target node sets.

[3] We will consistently use the term *symbol* to denote what is often called an input *token* in automata theory. This is to prevent confusion with the Petri net notion of token, which refers to an indicator of node activity.

The PFA M accepts a string ω if, for some state $q \in \delta(q_0, \omega)$, $q \subseteq F$. This intuitively says a string is accepted if, after it is fully consumed by *some* execution of M, *every* active node is a final nodes. Correspondingly, the language accepted by a PFA M is defined to be

$$L(M) = \{\ \omega \mid \delta(q_0, \omega) \text{ contains a subset of } F\ \}.$$

An error occurs during execution if no state transition is possible from the current state with the current input symbol.

1.2 Informal representation and execution

A PFA is best depicted as a form of directed graph with labeled transitions, as shown in figure 1. The nodeset N in the PFA is represented by the nodes in the graph. Each node transition in γ is represented by drawing a collective form of arc from its set of source nodes to its set of target nodes. We shall refer to such arcs as *tied arcs*. For any transitions with multiple sources or targets, a small circle is drawn as a collection point to emphasize the tying together of the various arcs composing the transition. The label is placed at the collection point. Collectively, the labels on the transitions leaving any particular node do not have to be unique.

A transition that enters more than one target node is termed a *parallel* transition; if it leaves more than one source node it is termed a *synchronizing* transition. The one or more nodes which collectively constitute the start state q_0 are denoted in the representation of a PFA with bold arrows. The final nodes in F are shown as double circles.

A node transition in γ is *possible*, or *enabled*, if all of its source nodes are active and the current input symbol matches the label on the node transition. At each step in the execution of a PFA, an enabled transition is nondeterministicly chosen and executed, producing a next state from the current state. Note that nondeterministic choice possibly arises in two ways: a single node can have several outgoing transitions bearing the same label, and two or more active nodes may each have outgoing transitions bearing the same label. When a transition is executed, all of its source nodes are made inactive, and then all of its target nodes are made active. This consumes the input symbol. There is no notion of a node being "doubly" active, as might be thought if a target node of an executed transition is already an active node.

If a transition is labeled with λ, it indicates that the transition is enabled whenever all of its source nodes are active, and that it does not consume any input when it executes.[4]

The most intuitive way to understand PFA execution is in Petri net terms. A PFA has basically the same structure as a Petri net, with the PFA node set N representing Petri net places, the tied PFA arcs in γ representing Petri net transitions, and the initial PFA state q_0 specifying an initial Petri net marking of one token per active place. PFA execution, then, proceeds as normal Petri net execution, *except* that the token count in each place is *normalized* (*i.e.*, any token count greater than 1 is set to 1) after each transition firing. Thus, the net never has more than one token in any place. The token distribution is part of the state information in a normal Petri net, and since multiple tokens can accumulate in places, a classical Petri net is a potentially infinite state automaton. The nets represented by PFAs, however, have 1-bounded token counts at each node in the graph, so PFAs are finite state automata. Section 2.1 below expands on this point by showing specifically the equivalence between PFAs and DFAs. First, we informally illustrate PFA execution with two examples.

1.3 Example 1

To illustrate the representation and execution of a PFA, consider the diagrams in figure 1. In the PFA, nodes 4 and 6 are the final nodes, and node 1 is the only start node. The language L accepted by this automaton is

$$a^*b \mid\mid (c^+d^+)^+$$

[4] Since our construction allows λ transitions, a PFA could have a single start node instead of a set of start nodes without loss of generality. We have chosen to define a set of start nodes for consistency with common Petri net notation.

Figure 1: Example of a PFA with an equivalent minimal DFA

where the || operator specifies an interleaving of the languages that are its operands. The DFA shown is equivalent to the PFA, in the sense that it accepts the same language L. It is also minimal for L. The execution sequence obtained while accepting the word "accbd$" (where "$" is the end marker) is as follows:

```
Active Nodes     Input Symbol       Action
  {1}                $\lambda$          control from 1 to {2,5}
  {2,5}              a           control from 5 to 5
  {2,5}              c           control from 2 to 3
  {3,5}              c           control from 3 to 3
  {3,5}              b           control from 5 to 6
  {3,6}              d           control from 3 to 4
  {4,6}              $           accept
```

In contrast, the execution sequence obtained for this PFA on the word "caba$" leads to an error, as follows:

```
Active Nodes     Input Symbol       Action
  {1}                \           control from 1 to {2,5}
  {2,5}              c           control from 2 to 3
  {3,5}              a           control from 5 to 5
  {3,5}              b           control from 5 to 6
  {3,6}              a           error (no possible moves)
```

1.4 Example 2

As further illustration, consider the PFA shown in figure 2. It recognizes the language

$$(a^*b \mid\mid c^*) \ m \ (ac^* \mid\mid (cd)^* \mid\mid b^*a)$$

Figure 2: Another PFA example

using two start nodes (1 and 3) and three final nodes (5, 6 and 9). An execution sequence for this PFA accepting the word "acbmaba$" is as follows:

Active Nodes	Input Symbol	Action
{1,3}	a	control from 1 to 1
{1,3}	c	control from 3 to 3
{1,3}	b	control from 1 to 2
{2,3}	m	control from {2,3} to {4,6,8}
{4,6,8}	a	control from 4 to 5
{5,6,8}	b	control from 8 to 8
{5,6,8}	a	control from 8 to 9
{5,6,9}	$	accept

Note that when control is at nodes $\{4, 6, 8\}$ and the input symbol is "a" the automaton can choose to move to have control in nodes $\{4, 6, 9\}$, but for the word in this example no accepting state can then be reached. Thus, the behavior of PFAs is somewhat similar to that of NFAs, as section 2.1 points out. By directly incorporating interleaving, though, the notation of PFAs is more convenient for expressing cooperating parallel activities.

Note also that the move from active nodes $\{2, 3\}$ to active nodes $\{4, 6, 8\}$ on input symbol "m" is a parallel synchronizing move. Because the transitions leaving nodes 2 and 3 are tied together, control must be at *both* nodes concurrently for the move to occur.

1.5 Related work

The language theory of regular expressions with interleaving is well known. Peterson [10, pp.169-171] restates (without proof) the result that type 0, 1, and 3 (regular) languages are closed under interleaving, and then (more to the point for this report) demonstrates a construction to support the fact that Petri net languages are closed under interleaving as well. We have adapted this construction, by adding λ transitions, for the generation of PFAs from augmented regular expressions, as described in section 2.2.

It was previously noted that PFAs are best thought of as a form of Petri net with a modified (normalizing) execution rule. In this light, they bear some resemblance to the class of automata recently introduced as *binary Petri Nets (BPN)* [2]. The definition and execution behavior of BPNs has been explained, but their language theory remains undeveloped. Several distinctions between PFAs and BPNs are evident, though, with the most obvious one being that, to emphasize the relationship with regular languages, PFAs are formulated more with a traditional finite state machine notation rather than the Petri net notation of BPNs. BPNs incorporate the notion of a net node either being active or not without counting the number of activations; thus they are also finite state automata, and as well they include the Petri net notions of parallel and synchronizing transitions that we have adopted for PFAs. However, BPNs have an unusual interpretation of an active node that is unlike other Petri net models and unlike PFAs. When a BPN node is active, all enabled transitions that leave that node must be executed at once. In PFA terms, this behavior would require several input symbols to be consumed simultaneously, since enabled transitions leaving a PFA node must be selected for execution one-at-a-time.

2 PFAs and regular languages

We now demonstrate the equivalence of the automaton class PFA with deterministic finite automata. After that, a translation procedure is presented to construct a PFA directly from regular expressions augmented with the interleaving operator.

2.1 Equivalence of the PFA and DFA classes

The equivalence of regular expressions and regular languages, and the corresponding equivalence of regular languages and DFAs, is well known. As stated in the previous section, it is also known that adding an interleaving operator to normal regular expressions does not change their expressive power. In the remainder of this paper, if not explicitly stated otherwise, we understand the term "regular expressions" to include an interleaving operator.

It is easy intuitively to see that the class PFA is equivalent to the class of finite automata. Every DFA is a PFA by degenerate case (there are no tied arcs in a DFA). In addition, since there are a finite number of nodes in a PFA, and since a PFA state is a subset of nodes, there are a finite number of states in every PFA. For completeness we give a formal constructive proof using the notation from our definitions, that is, we show how to obtain a DFA that accepts the same language as a given PFA.

Theorem 1 The set of languages generated by regular expressions is exactly the set

$$\{ \ L(P) \ \mid \ P \text{ is a } PFA \ \}.$$

Proof: by double subset.

We first establish the fact that every DFA is also a PFA (one without any parallel activity). This follows simply by letting the DFA states be the nodes of the PFA, and by altering the DFA transition function so that each state/symbol pair (s, a) in its domain becomes a set/symbol pair $(\{s\}, a)$ in the domain of the PFA transition function; similarly, each state s in the DFA range becomes a set $\{s\}$ in the PFA range. The start node set of the PFA is exactly the set containing the start state of the DFA, and the final node set of the PFA is the set containing the final states of the DFA. There are no tied arc groups in the resulting PFA.

Secondly, we argue that every PFA has a corresponding DFA that accepts the same language. The proof consists of an algorithm for generating the transition diagram of an equivalent NFA,

Figure 3: NFA equivalent to PFA in figure 2

along with a short argument that the construction terminates. The NFA can, of course, be converted into a DFA by the well-known "subset" algorithm (see [1, pages 117-119]).

Each state of the NFA will represent a set of active PFA nodes. The NFA can be generated by first creating an initial state that is exactly the set containing the start nodes of the PFA, marking the state "undone," letting this node be the "current" NFA state, and then generating successor NFA states from it. First construct the different sets of PFA nodes that can be active after a move is made from the active PFA nodes represented by the current NFA state. For each such PFA node set, if the NFA already has a state representing it, add an arc in the NFA from the current state to that next state. Label the arc with the input symbol that causes that move. For each set of PFA nodes that has not already been represented in the NFA, construct a new NFA state, mark the new state "undone," and add an arc from the current state to it. Label this arc with the input symbol that causes the move. After all possible successors of the current NFA state are generated, mark the current NFA state "done," pick an "undone" NFA state to be the new "current" state, and repeat the process of generating successors.

The NFA construction algorithm terminates because there are only a finite number of subsets of the nodes in the original PFA. Since each node generated in the NFA is either a new subset or a duplicate of a previously generated one, eventually no new nodes will be generated from each "undone" state and all NFA states will become "done."

□

The tied arcs in the PFA affect NFA generation by entering into the determination at each NFA state of the possible successors for that state. This procedure is very similar to the previously mentioned NFA-to-DFA construction algorithm; it is also essentially the same procedure used to generate the *coverability graph* of a Petri net [10], which, for bounded nets, can be thought of as a deterministic finite automaton. The NFA shown in figure 3 was generated by this construction and is equivalent to the PFA in figure 2. The states of the NFA have been named to correspond to the active PFA nodes each represents.

Figure 4: Interleaving two PFA languages

2.2 Composition of PFAs from regular expressions

For any regular expression r constructed from the normal operators (plus interleaving), a PFA P with λ transitions can be constructed such that

$$L(P) = L(r).$$

The procedure is an augmentation of the one given in Hopcroft and Ullman [7, pages 30-32] for constructing an NFA from a regular expression without the interleaving operator. We first note that any NFA is also a PFA, by argument analogous to the one made in the proof of Theorem 1 showing that any DFA is a PFA. Thus, the Hopcroft and Ullman algorithm already constructs a PFA from a normal regular expression. We augment the construction, then, by simply defining an additional component for the interleaving operator.

Given two PFAs $M(A)$ and $M(B)$ recognizing the languages A and B respectively, the language

$$(A \parallel B)$$

is recognized by a PFA M' constructed as follows. Create a new node for M' and make the initial state of M' be the set containing exactly this new node. Connect the node to all the start nodes of $M(A)$ and $M(B)$ with a parallel transition labeled λ. Create another new node for M' and let the set of final nodes for M' contain exactly this node. Connect all the final nodes of $M(A)$ and $M(B)$ to the new final node of M' with a synchronizing transition labeled λ. Figure 4 schematically depicts this construction.

3 Properties of PFAs

3.1 Safety and composition

Use of the composition algorithm from the previous section results in a special subclass of PFA having a structural property we refer to as *safety*. The notion of safety from Petri net theory[5] can be directly adapted to say that a PFA is safe if no node transition in γ, when executed, will attempt to activate an already active node. Note that self-loops (a transition that has some node as both a source and a target) are considered safe, since when a self-loop transition is executed, the source node can considered as becoming inactive before it is reactivated.

Theorem 2 A PFA generated using the given composition rules from regular expressions is safe.

Proof: by induction on the PFAs generated from each composition operator.

[5] A net is termed *k-bounded* if no place (node) can ever contain more than k tokens. A *1-bounded* net is termed *safe*. For details see Reisig [11].

For regular expressions A and B, if $M(A)$ and $M(B)$ are safe PFAs, then $M(AB)$, $M(A|B)$ and $M(A^*)$ (and so $M(A^+)$) are all safe because the composition rules create no new possible parallel activity. The PFA $M(A \parallel B)$ is safe because the concurrently active machines have no interconnecting arcs in the construction. The base cases of the induction (machines to accept single symbols) are obviously safe, because they are DFAs.

□

The safety property guarantees that the *structure* of a PFA is, in a sense, simple. There are no complex interactions among states, but only transitions from distinct subsets of nodes to distinct subsets of nodes. It is fairly easy to see that for any unsafe PFA, a safe PFA can be found that recognizes the same language. This is a consequence of the fact that safe PFAs are derived from regular expressions, which properly generate all regular languages.

3.2 Relation of PFAs to other Petri net models

We first relate PFAs to the Petri nets from which their parallel and synchronizing transition structures are borrowed. A Petri net can be viewed as a language generator/acceptor by mapping elements from a finite alphabet to the transitions in its structure. The recognition power of Petri nets is affected by the manner in which symbols are mapped to transitions (e.g., mapping a unique symbol to each transition, or allowing λ transitions), as well as how termination and acceptance is determined. Termination and acceptance in PFAs is a form of what are called $\mathrm{L}-type$ Petri net languages, in that a particular set of final states is identified. PFA acceptance occurs whenever a *reachable* subset of the final nodes is obtained. A full description of the Petri net languages can be found in Peterson [10, pp. 154-188].

Theorem 2 essentially shows how to construct PFAs that are structurally and behaviorally equivalent to safe Petri nets given the language to be recognized. A Petri net recognizing the same language can be created directly from a safe PFA structure by placing a Petri net transition bar on each PFA transition. Similarly, any safe Petri net is trivially a PFA. This direct *structural* equivalence with PFAs does not hold, however, for Petri nets that are bounded and not safe, that is, that have a finite state space but may sometimes contain two or more net tokens at some node. A PFA does not count the number of times that its nodes are activated, but simply records each as being "on" or 'off.'

For Petri nets that are bounded but not safe, though, a language equivalence with PFAs has already been proven. This can be seen from the fact that for every bounded Petri net, its previously mentioned *coverability graph* is, in fact, an NFA recognizing the same language as the net.

Obviously neither structural nor language equivalence with PFAs holds for general Petri nets since they admit possibly infinite state spaces.

3.3 Inhibitor arcs

Another interesting property of the PFA is that, unlike classical Petri nets, the addition of *inhibitor arcs* does not change the recognition power of the automata. An inhibitor arc is one that allows a transition to fire only if the node that is its source is empty (as opposed to *nonempty* for a normal arc). For Petri nets, this addition takes the class of automata into an equivalence with Turing machines. For PFAs, since they are finite state by construction, no change occurs in their power, but considerable flexibility is added as a modeling notation.

Other Petri net extensions also can be used with PFAs to extend their expressive flexibility while retaining the analyzability of a finite state space. For example, the recently introduced concept of *debit arcs* [15] (similar to *negative tokens* [8]) can be employed to represent systems allowing physical phenomena similar to economic debt, or deferred actions. Though we have not introduced any notation for such PFA extensions, the Petri net notation given in the indicated references will work well if the firing rule is altered to the normalizing one used with PFAs. In fact, Petri net notation in general is a viable alternative for the basic PFA notation of this paper. As mentioned earlier, we adopted a DFA-like notation for familiarity and to make plain the fact that a PFA has a finite state space.

Figure 5: PFA accepting (via forced safe firing) $(a|b)^*a(a|b)$

3.4 Forced safe firing

Another useful PFA technique adapted from Petri nets (specifically from *condition/event* systems [11, pp. 21-23]) is the notion of *forced safety*, or *forced safe firing*. A PFA can be forced to exhibit safety, whether or not it is structurally safe (from compositional construction, or otherwise), if the firing rule is altered to allow a state transition only when all input nodes are active *and* all output nodes are inactive (excluding self-loops). This alteration to the state transition function δ can be formally stated as follows:

Definition 3 Forced safe firing
Given an state $q \in Q$ and an input symbol $c \in (\Sigma \cup \{\lambda\})$,

$$\delta(q, c) = \{ \ (q - m) \cup n \mid n \in \gamma(m, c) \ \text{ for } \ m \subseteq q \text{ and } q \cap (n - m) = \emptyset \ \} \ .$$

Forced safe firing adds no recognition power to the class PFA. This can be shown by describing how to construct, for any PFA with forced safety, a normal PFA that recognizes the same language. Informally, this construction consists of creating an extra node (a *shadow*) for each node in the given PFA. For each original node, its shadow is tied as input to each transition into the node, and each transition out of the node is tied to the shadow as output. All shadow nodes are initially active. In effect, a shadow node provides mutual exclusion for its associated original node. Figure 6, in conjunction with the example in the next section, illustrates this construction.

Since a forced safe firing rule provides a notational convenience only, and no extra power, we will use it in subsequent examples where helpful without compromising the validity of the claims for normally executed PFAs.

3.5 Notational succinctness

We next note that PFA notation is highly succinct for expressing interleaved parallelism. For any regular language L, the minimal DFA recognizing L has at least as many nodes in its state transition diagram as the minimal PFA recognizing L. This is easily seen by noting first that the minimal DFA for L is also a PFA for L, so clearly no minimal PFA will be larger than its equivalent minimal DFA.

Figure 6: Node expansion to eliminate forced safe firing

However, the same exponential blowup experienced in coverting NFAs to DFAs can occur in converting PFAs to DFAs. In fact, for some PFAs, the equivalent minimal DFA suffers a doubly exponential blowup in the size of its state space. This is because the algorithm for creating the DFA may first, in an exponential number of steps, actually create an NFA; the NFA in turn will require an additional exponential number of steps to produce the minimal DFA.

The PFA in figure 5 provides an example of double-exponential blowup in converting a PFA to a DFA. This PFA accepts the string $(a|b)*a(a|b)^{31}$. Deciding whether or not to accept a string of this form requires keeping track of the last 32 characters, requiring 32 bits of memory with a state space of 2^{32}. More generally, we can write a PFA with $O(k)$ states that accepts strings of the form $(a|b)*a(a|b)^{2^k}$ and requires a state space of 2^{2^k}.

For conciseness, the PFA in figure 5 makes use of the forced safety firing rule: any transition with activated output sites that are not also input sites are unable to fire. Figure 6 shows the conversion of the boxed area of figure 5 into a PFA without the safe firing rule, using the construction previously described.

The PFA in figure 5 works similarly to the NFA for $(a|b)*a(a|b)^n$ normally used to show exponential blowup. The bottom part of the PFA encodes a binary counter than must receive either 0 or 32 activations. We can encode a counter that must receive either 0 or 2^k activations in $2k + 2$ states using the assumed firing rule (or 4k+4 without the forced safety firing rule).

4 PFA Use in Software Design and Modeling

This section illustrates the value of PFAs in design and construction of concurrent systems with several brief examples excerpted from previous reports on a multi-user information browsing system called Trellis.[6] The goal of this summary is not to fully explain the details of Trellis and its uses, but to give the reader a general idea of some practical applications for PFAs. In these examples, the inherent parallelism in the structure of a PFA is used for directly expressing important physical or logical aspects of the system being specified; the inherently finite state space of the PFA is exploited for exact analysis of the dynamic behavior of each system.

Our examples concentrate specifically on applications that have been studied for Trellis, and for the interpretation of nets with browsing semantics. Many other software engineering and system design researchers have developed applications for generic PT nets, though. For example, we refer the reader to the recent text by Ghezzi *et al.* [5] for an extended examples of how PT nets can be used as formal specifications in the lifecycle of concurrent and real-time software systems.

[6] The design, construction, and study of Trellis is the work of Richard Furuta along with the first author; the bibliography lists several detailed reports about this project.

Figure 7: Trellis client/server system architecture.

4.1 Trellis hyperprograms: PT nets with browsing semantics

The Trellis project [16, 4] has investigated for the past several years the structure and semantics of human computer interaction, in the context of hypertext/hypermedia systems, program browsers, visual programming notations, and software process models. In the ensuing discussion, we will refer to an information structure in Trellis as a *hyperprogram*. Due to the unique features combined in the Trellis model, a hyperprogram integrates user-manipulatable information (the hypertext) with user-directed execution behavior (the process). We say that a hyperprogram *integrates task with information.*

The Trellis model treats a hyperprogram as an annotated, timed PT net that is used both as a directed graph (for static information) and as a parallel automaton (for dynamic behavior). The places of the PT net are annotated with fragments of information (text, graphics, video, audio, executable code, other hyperprograms); these annotations are termed the *content* elements of the hyperprogram. A hierarchy is created indirectly by allowing the content element of a place to be itself an independent hyperprogram.

Visual interfaces provide users with a tangible interpretation of a net and its annotations. For example, annotations on net places might be Unix file names, with display names attached to transitions. When a token enters a place during net execution, the file for that place would be presented for viewing. The names of enabled transitions leading out of the place would be shown as a menu of selectable *buttons* next to the file. Selecting a button (with a mouse, usually) would cause the net to fire the associated transition, moving the tokens around and changing which files would then be visible.

In a Trellis implementation, this cooperative separation between net and interpretation is realized by a distributed *client/server* network, as shown in figure 7. Every Trellis model is an information server—an engine that accepts remote procedure call (RPC) requests for its services. The engine has no visible user interface, but does have an API that allows other remote processes to invoke its functions for building, editing, annotating and executing a PT net. Interface clients are separate processes that have visible user interfaces and communicate with one or more engines via RPC. Clients collectively provide the necessary views, interactions, and analyses of a net for some specific application domain. Simply put, Trellis clients are the syntax of an application, whereas Trellis engines are the dynamic semantics; clients and servers provide an application's look and feel respectively.

Figure 8: Concurrently displayed Neptune images classified by features.

As the following examples show, Trellis hyperprograms are especially useful expression and simulation of processes in which human direction is an important aspect of the control flow. An example is the software development process. Such computations are referred to as being *enacted*, rather than as being executed, to distinguish the major role human input and human decisions have in the unfolding of the actions described in the program. A version of Trellis is currently being developed specifically for expression and enactment of software process models, and PFAs are one form of engine being used to structure these models.

Example: image browsing index

Figure 8 shows a screen from an early Trellis prototype implemented for Unix platforms and the SunView window system. This example is an image browsing index constructed as part of a NASA experiment at CESDIS (Goddard Space Flight Center, MD). Images of Neptune and Phobos are linked and cross linked in the net structure according to common characteristics. The net concurrently displays all images that share some characteristic, as selected by a reader from the browsing interface.

Three interface clients are visible: a graphical editing client on the right, overlapping the windows of a text browsing client on the left, and a graphics image client on the bottom; all three are communicating via RPC with one Trellis engine. In the text browser are four text windows. When a net place is marked its content element is displayed in one of these text windows. In this example, the text showing in the top window of the browser is the content of the place "neptune" which is showing as marked in the graphical editor window. Each enabled transition is displayed as a selectable button in a menu to the left of the text window (for example, transitions "rings" and "clouds" in the net, among others). Selection of a button in a browser menu causes the associated transition to fire in the net, changing the net state and thereby the display as well.

The net editor client on the right presents a graphical view of the underlying model. With it, a user can build or alter the structure of the net, annotate the net with content element names, and also execute the net. The graphics client on the bottom is displaying under X Windows on a different monitor. It monitors the execution activity in a hyperprogram and renders graphics images on the X screen whenever a marked place has a bit-mapped image as its content. At the instant shown, the places "nepColor" and "nep3" are marked in the net, so two images show.

The graphics client does not allow a user to alter the net structure like the editor client does; it does not even allow a user to fire transitions like the text browser does; it just sits and listens, acting when necessary according to its purpose.

Example: Parallel program browsing

The Trellis application illustrated in this section shows both the usefulness of the model for representing parallel threads of activity, and the usefulness of our hypertextual interpretation of the PT net for supporting human reasoning through browsing. A CSP program browser [12] is shown in figure 9. This specific example uses a CSP program taken from Hoare's original paper [6].

We wrote a translator to parse CSP programs and generate as output the storage format of Trellis hyperprograms. The translation converted the control structures of CSP statements and the message buffers between CSP processes into PT net structures with the appropriate control behaviors. Each place in the PT net represents a statement from the source program. We annotated the places with CSP source code; each place is mapped to a copy of the CSP process that contains its statement, with that statement highlighted (this gives a reader some context for the statement). The top view of figure 9 shows the editor client full-screen to illustrate the graphical structure of this particular model. The bottom view shows the editor client with a closeup of the net, with the text browser displaying the CSP code segments that are active at this point in simulated execution.

The result is a browsing system for simulating the parallel execution of CSP programs. The simulation proceeds by selecting buttons in the text browser to "execute" statements one at a time. The simulation proceeds at user speed and at a user's discretion, following a user's train of thought as browsing progresses.

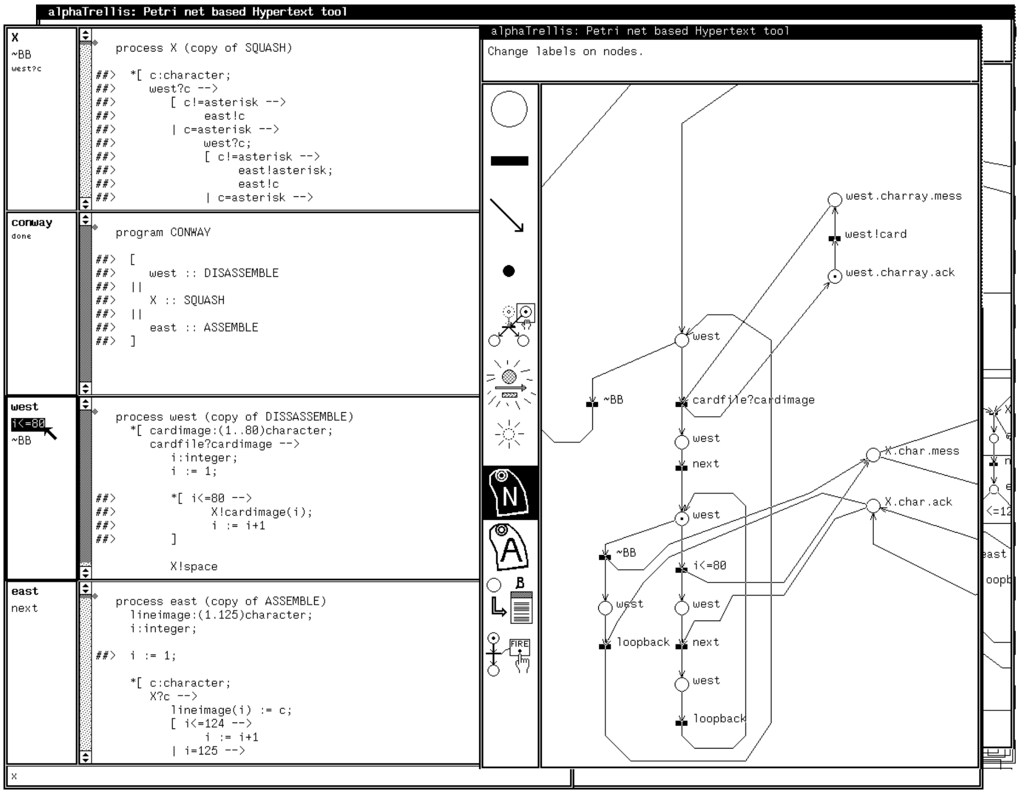

Figure 9: Two views of Trellis used for browsing a CSP parallel program.

Figure 10: Initial Dining Philosophers screen.

Example: Process simulation in Trellis

The previous example shows user-directed simulation of processes in Trellis. This next example illustrates a facility of the model that also allows non-user directed control in a process simulation. The method uses timed transitions in the PT net, a Lisp interpreter in the Trellis engine, and chunks of Lisp code (called *agents*) on net transitions [17, 13]. When a transition is fired, its Lisp agent (if present) is executed. In this form, Trellis resembles the concurrent programming language Linda, in that a sequential kernel language (Lisp) is separate from, but works with, a parallel control flow language (timed PT nets).

A transition in a Trellis net can be given a time-out value; the engine will automatically fire such a transition if it sits unfired for this amount of time after it is enabled. A time-out can be specified either as a constant, or as the value of some Lisp variable. Lisp agents can change transition time-outs while a hyperprogram is being browsed, thereby dynamically altering the behavior of a Trellis structure.

The traditional concurrency abstraction of Dining Philosophers illustrates the use of Lisp agents for process simulation in Trellis. An network of four philosophers is shown in figure 10. Two text browsing clients are executing along with the Trellis editor client; the leftmost browser shows place content elements (which in this example are inconsequential) and the middle browser shows the Lisp agents associated with enabled transitions. After the "init" transition is fired, the Lisp agents alter various Lisp variables so that fork events (grab, drop) fire in a disciplined fashion. The code shown implements a round-robin schedule, with fork transitions firing about every 6 seconds. Other scheduling policies can obviously be simulated by writing Lisp agents to trigger the transitions appropriately. A reader may also directly manipulate fork events by reacting faster than the time-outs and selecting buttons directly in the browser interface.

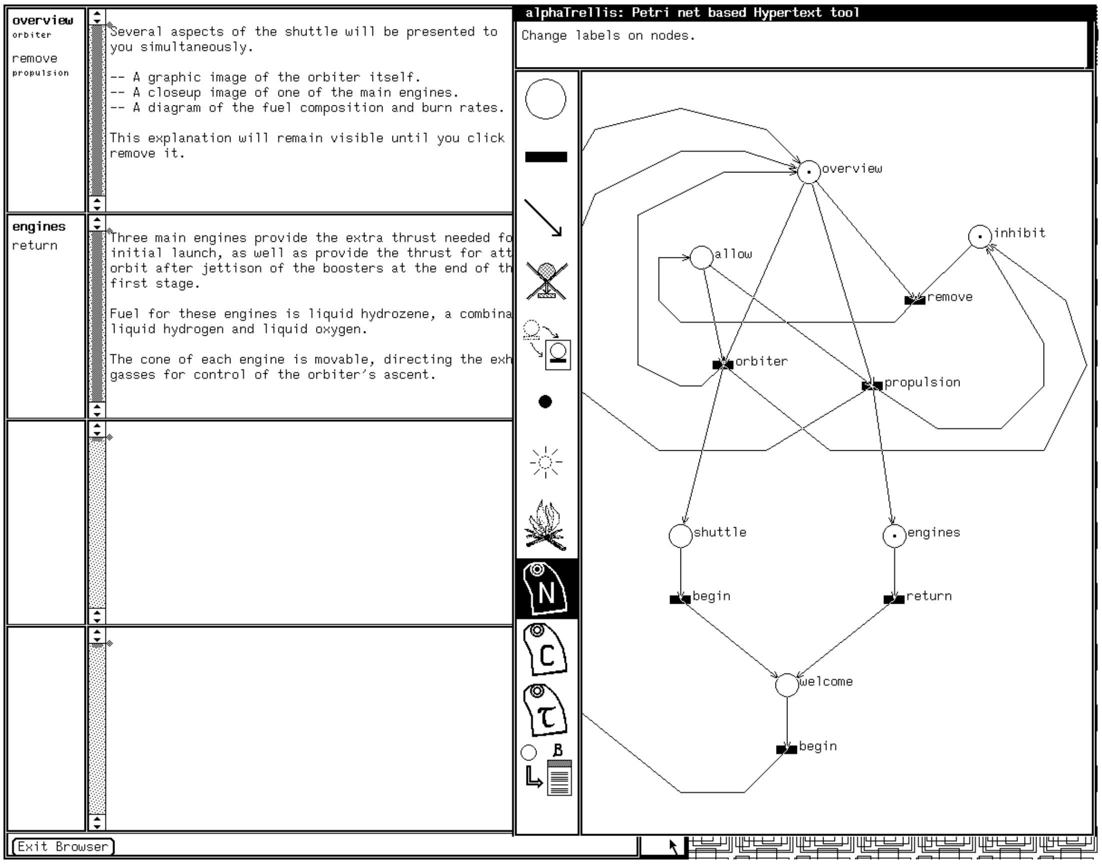

Figure 11: Small Trellis structure with programmed browsing behavior.

4.2 Process analysis: finite-state model checking

Trellis and its implementations provide a formal structure for hyperprograms, and net analysis techniques have been developed for exploiting this formalism. One very promising approach involves adaptation of automated verification techniques called *model checking* [3] from the domain of concurrent programs. This approach allows verification of browsing properties of Trellis hyperprograms expressed in a temporal logic notation called CTL. An author can state a property such as "no matter how a document is browsed, if Node X is visited, Node Y must have been visited within 10 steps in the past." The model checker efficiently verifies that the PT net structure maintains the validity of the formula denoting the property.

In model checking, a state machine (the model) is annotated with atomic properties that hold at each state (such as "content is visible" or "button is selectable"), and then search algorithms are applied to the graph of the state machine to see if the subcomponents of a formulae hold at each state. By composing the truth values of these subformulae, one obtains a truth value for the entire formula. For PT nets, we obtain a state machine from the *coverability graph*.

The details of our use of CTL are discussed elsewhere [14]. For this rationale, it is sufficient to give an idea of how the method is applied to net models. The Trellis document shown in figure 11 is a small net that expresses the browsing behavior found in some hypertext systems, namely that when a link is followed out of a node, the source content stays visible and the target content is added to the screen. The source must later be explicitly disposed of by clicking a "remove" button.

After computing the coverability graph and translating it into the input format required by the checking tool, the model can be queried for desired browsing properties. These examples use the syntax of Clarke's CTL model checker, and show its output:

- Is it impossible for both the "shuttle" text and the "engines" text to be concurrently visible?

```
|= AG( ~C_shuttle | ~C_engines ).
The formula is TRUE.
```

- Can both the "allow" access control and the "inhibit" access control ever be in force at the same time?

```
|= EF(C_inhibit & C_allow).
The formula is FALSE.
```

- Is it possible to select the "orbiter" button twice on some browsing path without selecting the "remove" button in between?

```
|= EF(B_orbiter & AX(A[B_remove U B_orbiter])).
The formula is FALSE.
```

This particular Trellis model is very small compared to those encountered in realistic applications. Our checker has also been tested on larger Trellis documents. For example, the one shown back in figure 9 produced a state machine with over six thousand states. Using a DECstation 5000/25, the performance of the model checker on formulae like those above is still on the order of a few seconds each.

5 Discussion and Conclusions

The specific firing semantics chosen for a PT net syntax determines the composition of the state space for any particular net structure. We have built and experimented with several different PT net firing rules in Trellis engines. Though the main portion of an engine is a C++ program with over 50 methods (comprising over three thousand lines of code), making a version of the engine with a new firing semantics requires changing a relatively few methods—specifically, those methods related to enabling and firing of transitions.

The first Trellis system was based on a Petri net, which is a PT net syntax with a firing rule allowing tokens to accumulate in places without bound. This admits potentially infinite state spaces, but also allows Petri nets to recognize languages more complex than the regular sets.

We have created another version of the engine to implement PFA semantics. This version allows use of the familiar PT net syntax when modeling concurrent systems, yet it keeps the state spaces of resulting structures finite. When a coverability graph is generated from a PFA, no information is lost and the state space is represented exactly. With PFAs, then, model checking verification methods give an exact analysis. With traditional Petri nets, computation of a coverability graph will discard some information in order to approximate an infinite state space in a finite graph. Due to this lost information, model checking on Petri nets cannot reliably answer all queries.

Other PT models for which Trellis engines can easily be created include the previously mentioned *debit nets* [15] and *binary Petri nets)* [2].

Other finite state models with PT net syntax exist. One such model, the *condition/event (C/E) system* mentioned in section 3.4, has an execution rule that prevents a transition from firing if any of its output places are marked. C/E systems, then, have implicit synchronization conditions that are not directly a part of the net structure. They are most applicable to physical systems that require this enforced synchronization behavior; for example, C/E systems are natural for modeling robot assembly lines, where the places represent physical resources that can handle one task at a time. If a robot is busy (its place has a token), then no transition may fire that would send more activity (another token) to it.

The C/E system firing rule does produce a finite state space, but the natural association of places with physical resources in not appropriate for more abstract systems like information browsing structures. We noted that in Trellis net places represent document content (text, graphics, sound, etc.). A token in a place means that the content is rendered for reader use. A transition represents a link between the content of its source place and the content of its destination place. If a link is followed (transition is fired), then the currently displayed content element is deactivated, and the content elements mapped to the output nodes are activated, in accordance with token movement. It makes no sense to always require, as with C/E systems, that a reader *cannot follow a link* to see, say, a new text block if that text block is already rendered for viewing. It is more natural to think of the reader following the link (firing the transition), but nothing changing in the visible state (i.e., the extra token in the output place is normalized away). Thus, our physical environment requires PT net semantics free of implicit synchronizations; PFA behavior meets this need.

We conclude by noting that, even though PFAs and DFAs are equivalent in language theoretic terms, the inherently parallel state representation in a PFA makes this automaton class more useful as a modeling tool. The fact that the DFA automaton class is a direct *structural* subset of the PFA class makes the PFA a preferred replacement for the traditional state machine representation in applications involving concurrency.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] H. Alayan and R. W. Newcomb. Binary petri-net relationships. *IEEE Transactions on Circuits and Systems*, CAS-34(5):565–568, May 1987.

[3] E. Clarke, E. A. Emerson, and S. Sistla. Automatic verification of concurrent systems. *ACM TOPLAS*, 8(2):244–263, April 1986.

[4] Richard Furuta and P. David Stotts. Programmable browsing semantics in Trellis. In *Hypertext '89 Proceedings*, pages 27–42. ACM, New York, November 1989.

[5] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 1991.

[6] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[7] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[8] T. Murata and H. Yamaguchi. A petri net model with negative tokens and its application to automated reasoning. In *Proceedings of the 33rd Midwest Symposium on Circuits and Systems*, pages 7–10. IEEE, August 12-15 1990.

[9] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[10] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., 1981.

[11] Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[12] P. D. Stotts and R. Furuta. Browsing parallel process networks,. *Journal of Parallel and Distributed Computing*, 9(2):224–235, 1990.

[13] P. D. Stotts and R. Furuta. Hypertextual concurrent control of a Lisp kernel. *Journal of Visual Languages and Computing*, 3(2):221–236, June 1992.

[14] P. D. Stotts, R. Furuta, and J. C. Ruiz. Hyperdocuments as automata: Trace-based browsing property verification. In *Proceedings of the 1992 European Conference on Hypertext (ECHT92: November 30–December 4, Milan, Italy)*, pages 272–281. ACM Press, New York, 1992.

[15] P. D. Stotts and P. Godfrey. Place/transition nets with debit arcs. *Information Processing Letters*, 41(1):25–33, January 1992.

[16] P. David Stotts and Richard Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, January 1989.

[17] P. David Stotts and Richard Furuta. Temporal hyperprogramming. *Journal of Visual Languages and Computing*, 1(3):237–253, 1990.