UC Irvine
ICS Technical Reports

Title
Modeling and analysis of concurrent systems

Permalink
https://escholarship.org/uc/item/536135nf

Author
Sidwell, Richard D.

Publication Date
1987

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material may be protected by Copyright Law (Title 17 U.S.C.)

Modeling and Analysis of Concurrent Systems

Richard D. Sidwell

Technical Report 87-03
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

Abstract

A survey of modeling and analysis techniques in common use for modeling and analyzing concurrent systems. The models surveyed are CSP (Communicating Sequential Processes), Path Expressions, CCS (Calculus of Communicating Systems), CIRCAL, Petri Nets, Coloured Petri Nets, Predicate-Action Nets, Numerical Petri Nets, Contour-Transition Nets, and several varieties of Timed Petri Nets. The analysis techniques are state-space analysis, temporal logic, structural analysis, and inductive analysis.

This work was supported in part by NSF grant DCR 84-06756

Contents

	Page
Introduction	1
Models Based on Synchronizing or Communicating Processes	3
Communicating Sequential Processes	4
Path Expressions	5
Calculus of Communicating Systems	9
CIRCAL	10
Net Based Models	11
Condition-Event Systems	12
Place-Transition Nets	14
Coloured Petri Nets	16
Predicate-Transition Nets	17
Predicate-Action Nets	17
Contour-Transition Nets	21
Numerical Petri Nets	21
Extended Petri Nets	22
Super Nets	23
Ramchandani's Timed Petri Nets	23
Merlin's Time Petri Nets	24
Razouk's Timed Petri Nets	25
Analysis Techniques	25
State-Space Analysis	26
Temporal Logic	28
Structural Analysis	29
Inductive Analysis	31
Language Theory	32
Reductions	32
Simulation	32
Other Analysis Techniques	32
Summary	32
References	35

Modeling and Analysis of Concurrent Systems

Introduction

As the field of computer science progresses, more and more computer systems are designed which contain more than one processor. Having more than one processor increases the power of a system since the processing tasks may be shared by the different processors as they run concurrently. At the same time, the complexity of the system is also increased since the processors must usually communicate and synchronize with each other in order to work together as a system. The complexity is further increased by the fact that the separate processors do not generally run at exactly the same speed—the exact order of computation differs from one run of the system to the next, even with exactly the same input data. The net result of this is that methods for analyzing and validating systems with only one processor will generally not work for concurrent systems; different analysis methods are needed. The complexity of concurrent systems can be reduced to make analysis feasible by making a model of the system. In this paper, some of the techniques used for modeling and analyzing concurrent systems will be described.

Before describing modeling and analysis techniques for concurrent systems, however, it will be useful to define some of these terms. A concurrent system is a system in which several parts may be executing in parallel. Such systems are generally implemented with several processors, but this is not required—many such systems may be implemented on a single-processor system using time-sharing techniques. The implementation of concurrent systems will not be addressed in this paper; what will be described is a number of techniques which may be used to model and analyze concurrent systems independently of how the concurrency is implemented.

A model of a system is an abstraction of that system. It accurately reflects the parts of the system which are important, but omits the fine details which complicate the analysis. Of course, if the omitted details were completely unimportant, there would be no reason to implement the full system—the model would work just as well. So the question in modeling a system is to decide which details are important for the analysis and which are not. The answer depends on what properties are to be analyzed. As a non-computer example, consider the design of a new airplane. If the designer wants to analyze its air-flow properties, he might build a small replica of the exterior of the airplane, with adjustable control surfaces, and use a wind tunnel to perform his analysis. However, such a model would not be very useful for analyzing how well the plane would survive a crash—a model to study this property would have to include internal structural details and how the passengers would be seated, but the control surfaces would probably not need to be modeled

very accurately. Similarly, a model of a concurrent system used to analyze safety and liveness properties may not be the best model to analyze the performance of the system.

System analysis can be defined as determining what properties the system has. There are several types of properties that we are interested in. One type includes general properties, such as boundedness and liveness. Another type includes system specific properties, such as whether a communication protocol delivers messages in the order in which they were sent. These are the main types of properties we will discuss here. There are many other types, such as performance properties, which are not within the scope of this paper; we will mention them briefly when they seem appropriate, but will not discuss them in detail.

Of the many different considerations which must be made when comparing modeling techniques, there are two important ones which we will discuss here: modeling capability and modeling power. Modeling capability is a way of comparing the systems which different modeling techniques are capable of modeling. For example, in the realm of non-concurrent models, a Turing machine has more capability than a finite-state automaton, since anything which can be modeled by a finite-state automaton may also be modeled by a Turing machine, but a Turing machine is capable of modeling many systems which a finite-state automaton can not. Many authors use the term modeling power when making this kind of comparison; we wish to use that term for something else, so we use modeling capability when comparing what modeling techniques can do when pushed to their limits. Modeling power is a way of comparing how efficient modeling techniques are at modeling systems. Given a system to model, a more powerful modeling technique can model it using a smaller model than a less powerful one. For example, a non-deterministic finite-state automaton is more powerful than a deterministic one, since in general, smaller models can be used if non-determinism is allowed. However, a deterministic finite-state automaton can simulate a non-deterministic one if required, so the two models have the same modeling capability.

The modeling techniques in common use for modeling concurrent systems can be divided into two categories. The first category contains models which consist of a number of concurrently executing sequential processes which use synchronization and/or communication operations to communicate with each other; these models will be discussed in section 2. The other category contains models which are based on net theory, exemplified by Petri nets; these models will be discussed in section 3. Although analysis techniques for the models are generally developed at the same time as the models, we will discuss them separately here, in section 4. The modeling and analysis techniques will be described only informally and with examples—references will be given to formal and detailed descriptions for those readers who need them. There are many modeling and analysis techniques which are not described here–the ones chosen for inclusion in this paper are those which seem most important, useful, and/or interesting. The intent of this paper is to

give the reader an overview of the important modeling and analysis techniques for concurrent systems so that he can choose which might be the most useful for his needs and consult the literature for details on how to use them.

Models Based on Synchronizing or Communicating Processes

Since many concurrent systems are implemented as a set of processes which execute in parallel, pausing occasionally to synchronize and/or communicate with each other, it is natural to use modeling techniques which do the same. Models, of course, can often omit needless details such as what a modeled process does between the times it interacts with other processes. There are many ways of expressing synchronization and communication; the models described in this section will illustrate the variety possible. However, all of these models have one trait in common: they are composed of several processes executing at the same time, and which synchronize or communicate with each other in order to model a complete system.

In order to express more clearly the relationships and differences between these models, we will use a metamodel based on that by Milne and Milner in [MILNE79]. Although this metamodel is used for some of the modeling techniques we will discuss, it is not used by all of them, and the notation will be a little bit different. After the description using this metamodel, the standard notation will be described. In our metamodel, processes are represented by *agents*, which are drawn graphically as boxes. Each agent has a number of *ports*, which are drawn graphically as dots on the boxes. Each agent and port may be labeled. For each modeling technique, we need to specify two things: how to describe the behavior of each agent, and how the agents interact.

As a common example for all of these modeling techniques, we will use an example from [MILNER80] of a scheduler. A set of agents $\{p_i \mid 1 \leq i \leq n\}$ desire to perform a certain task repeatedly; the scheduler must ensure that they perform it in rotation, starting with p_1. Although the p_i are to start their performance in rotation, their performances do not need to be mutually exclusive—more than one process may be performing the task at a given time. We also enforce the restriction that no p_i may initiate the task twice without completing the first initiation. A diagram of the structure of the scheduler system using our metamodel is shown in Figure 1. Note that the examples we will give include only the scheduler, not the processes p_i which it controls.

Although some of the modeling techniques use greek letters for labels, we will use roman letters here for their mnemonic value. Each task p_i requests initiation at label s_i and signals termination at label t_i. The scheduler must impose two constraints on any signal sequence $(s_i \cup t_i)^\omega$:

1) When all occurrences of t_i are deleted, it becomes $(s_1 s_2 \ldots s_n)^\omega$;

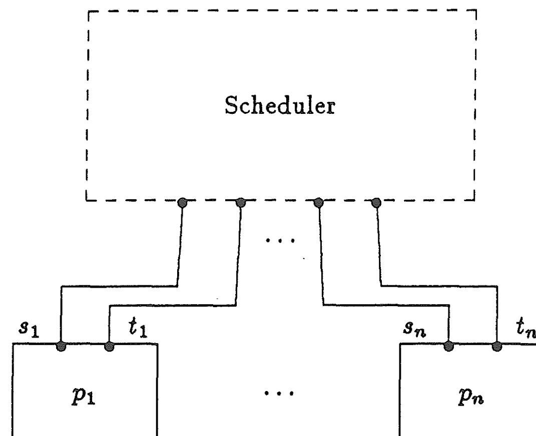

Figure 1
The Structure of the Example Scheduler

2) For each i, when all occurrences of $s_j, t_j (j \neq i)$ are deleted, it becomes $(s_i t_i)^\omega$.

Communicating Sequential Processes

Communicating Sequential Processes (CSP) were first described by Hoare in [HOAR78] as a concurrent programming method based on communication between concurrently executing sequential processes. The behavior of each process is specified as a program in a simple programming language which has the standard programming constructs of assignments ($x := x + 1$), conditional statements (e.g., $[x < 0 \rightarrow x := -x \,\square\, x \geq 0 \rightarrow x := x]$), and loops ($*[r \geq y \rightarrow r := r - y; q := q + 1]$). The conditional and loop statements differ slightly from standard programming languages; they use the notation of guarded commands. Each alternative consists of a guard and a statement list. The alternatives are seperated by a $\square$. Any alternative whose guard is true may be selected, and its statement list executed. If more than guard is true, one of the alternatives is selected non-deterministically. For loop statements, the statement is executed repeatedly until none of the guards are true; execution then proceeds to the next statement. In addition, each process has a name, and can communicate with other processes by referring to them by their name. Two types of communication are possible: input and output. The statement $A?x$ inputs a value from the process named A and assigns it to x. The statement $A!x$ sends the value of x to the process named A. If process A executes the statement $B!(x + y, z)$, process B must execute a statement such as $A?(w, x)$ in order for the communication to take place; processing is suspended until B does (or, if B happens to execute its input statement first, B will be suspended until A executes its output statement). The input statement may also be used as a guard in a conditional or loop statement; it is true if input is available (and is executed if that guard is selected) and false if not.

```
B ::
b : (0..9) data
i, o : integer; i := 0; o := 0
    *[i < o + 10; P?b(i mod 10) → i := i + 1
    □ o < i; C? more() → C!b(o mod 10); o := o + 1
    ]

P ::
d : data
    *[true → produce d; B!d]

C ::
d : data
    *[true → B!more(); B?d; consume d]
```

Figure 2
A Bounded Buffer System in CSP

To illustrate a simple CSP program, we show here an example of a bounded buffer system from [HOAR78]. The problem is to construct a buffering process B which smooths variations in the speed of output by a producer process and input by a consumer process. The consumer C executes the statement B!more() when it is ready to accept data from the buffer, followed by $B?d$ to get it. The producer executes the statement $B!d$ to send data to the buffer. The system is shown in Figure 2.

As another example, the scheduler described above is shown modeled in CSP in Figure 3.

For comparison with the other models described in this section, we now show how CSP works in terms of our metamodel. Each process is an agent with a label. Each agent has two ports for each process with which it communicates: one for input, labeled with the name of the process, and one for output, labeled with the name of the process with a bar over it. The interactions between the agents are fixed —each port is connected to the appropriate port on another agent such that the label of the port corresponds to the label of the agent connected to it and barred ports are connected to unbarred ports, as shown in Figure 4. When an agent activates a port, the other agent must activate the port connected to it in order for both to proceed.

Path Expressions

In path expressions, proposed by Campbell and Habermann in [CAMPB74], the behavior of each agent is described by an expression which essentially a regular

```
S1 ::
   *[true → p1!s();
            [p1?t() → S2!e()
             □ true → S2!e(); p1?t()];
             Sn?e()
   ]
S2 ::
   *[true → S1?e();
            p2!s();
            [p2?t() → S3!e()
             □ true → S3!e(); p2?t()];
   ]

            ⋮

Sn ::
   *[true → Sn-1?e();
            pn!s();
            [pn?t() → S1!e()
             □ true → S1!e(); pn?t()];
   ]
```

Figure 3
A CSP Model of a Scheduler

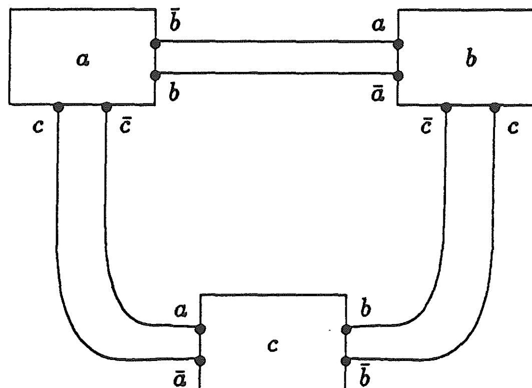

Figure 4
Interactions Between Agents in CSP

expression, except that the operators are not the standard ones. An identifier indicates that the port with that label may be active. A semi-colon is used to specify that two things happen in sequential order and a comma is used to specify that either of two things may happen. A star (*) is used to indicate repetition—something can occur zero or more times. Some examples will make this more clear. Suppose we wish to model a process which implements a simple buffer. It accepts a request to deposit data into it, then a request to remove data from it, then another deposit request, then another remove request, and so forth. If d represents

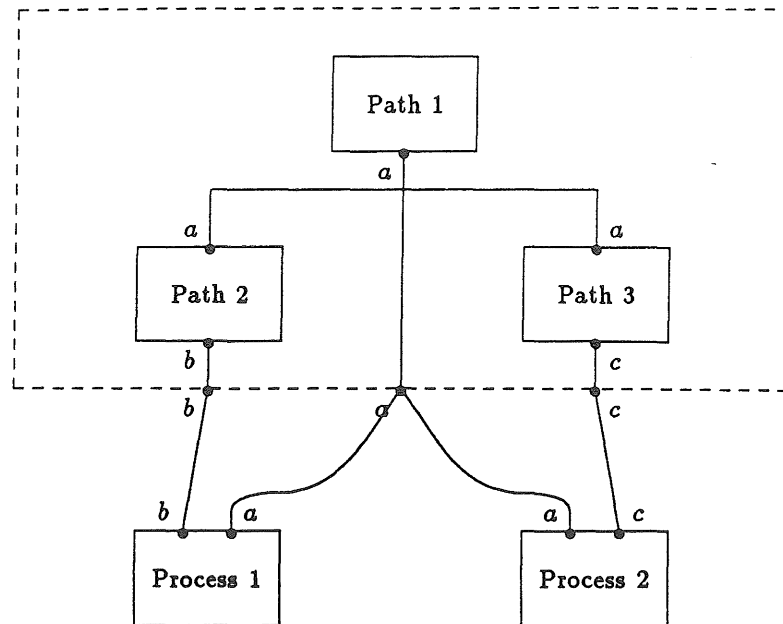

Figure 5
Agent Interactions in Path Expressions

the deposit request and r represents the remove request, a path expression which models the buffer would be $(d;r)^*$. If, instead of a buffer, we wish to model a process which controls a shared memory location, the expression would be slightly different—a shared memory location may be read many times after one write. The first operation must be a write, but reads and writes may be done in any order thereafter. Using w and r (write and read) instead of d and r, the path expression is $w;(r,w)^*$. Finally, let us model a process which accepts an input i and sends it to one of two buffers, a or b, then accepts another input, and repeats this indefinitely. The path expression would be $(i;(a,b))^*$ or, since the comma has precedence over the semi-colon, $(i;a,b)^*$.

The interaction between agents is handled by dividing the agents into two groups: paths and processes. For paths, all ports with identical labels are connected together, and no port is active unless all ports on all of the path agents are active. For processes, the ports are connected to the path ports with the identical label, but not to each other, so a port is active only if all of the ports on the path agents are active, but only one process agent will be activated. It may be easier to think of all of the paths together forming a new agent, with which the processes interact, as shown in Figure 5. The paths are intended to model resources and their access restrictions, and the processes are intended to model processes which use the resources.

As an example of how paths and processes combine to form a model of a concurrent system, consider Figure 6, which is a path expression model of a system

path $(d_1; r_1)^*$ (buffers)

path $(d_2; r_2)^*$

process $(p; d_1, d_2)^*$ (producers)

process $(p; d_1, d_2)^*$

process $(r_1, r_2; c)^*$ (consumers)

process $(r_1, r_2; c)^*$

process $(r_1, r_2; c)^*$

Figure 6
Producers and Consumers with Path Expressions

path $(s_1; ((t_1; e_2), (e_2; t_1)); e_1)^*$

path $(e_2; s_2; ((t_2; e_3), (e_3; t_2)))^*$

$\vdots$

path $(e_n; s_n; ((t_n; e_1), (e_1; t_n)))^*$

Figure 7
A Scheduler Modeled Using Path Expressions

with two buffers, two producers, and three consumers. As described above, the buffers simply accept deposit and remove requests; always waiting for a deposit before a remove is allowed. The producers repeatedly produce a data item (p), and then deposit it in either of the buffers. The consumers repeatedly remove a data item from either of the buffers and then consume it (c).

As final example, let us model the scheduler described previously using path expressions. We must introduce another set of labels to control the interactions between the paths. We call it e_i because it is used to enable the path which is used to control the initiation and termination of p_i. The model is shown in Figure 7. The path for process p_1 is a little different since it is enabled when the scheduler begins. For a process p_i, we first wait for the path to be enabled, then allow the task to be initiated. Finally we allow the task to be terminated and enable the next path, in either order. The processes p_i themselves are not shown, but would be represented as processes in the path expression model.

An interesting extension to Path Expressions is Predicate Path Expressions. In this model, an expression can be conditioned by a predicate; the expression can only be executed if the predicate is true. For example, a path expression model of the readers/writers problem would be:

$$\text{path } ((w_s; w_e)[\#(r_e) - \#(r_s) = 0], ((r_s; r_e)[\#(w_e) - \#(w_s) = 0])^*.$$

$c = e\bar{s}(\bar{t}n + n\bar{t})c$

$c_i = c[s_i/s, t_i/t, e_i/e, \bar{e}_{i\oplus 1}/n]$

$g = \bar{e}_1\text{NIL}$

$sch = (g \mid c_1 \mid \cdots \mid c_n)\backslash e_1 \cdots \backslash e_n$

Figure 8
A Scheduler Modeled Using CCS

In this model, w_s indicates a writer starting, w_e indicates a writer ending, r_s and r_e indicate a reader starting or ending, and $\#(x)$ is the number of times x has occurred. The predicates are given in square brackets after the expression each conditions.

Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) was developed by Milner, and is described in [MILNER80]. In terms of our metamodel, each port is labeled with either a name (a label without a bar over it) or a co-name (a label with a bar over it). There are two operations which may be used to describe the behavior of an agent: concatenation (represented by juxtaposition), indicating that the operands are performed sequentially, and summation (+), indicating that a choice is made as to which operand is performed. The primitives are the port labels and NIL, indicating termination. There is no explicit looping operation, but expressions may be given names, which may appear in the expressions, allowing recursion. There are three operations which describe the interaction between agents: composition ($\mid$), restriction ($\backslash\alpha$) and renaming ($[S]$). Composition of two agents is done by connecting all ports with complementary labels (connecting names to co-names). Composed agents have the same behavior as the component agents (including activation of connected ports), but in addition, connected ports may activate each other, causing a change of state without an external stimulus. Restriction is done by simply removing the specified label from the agent, preventing further connections to that port, although existing connections remain intact. Renaming is done by changing the names of the ports according to the renaming scheme S. The renaming scheme S is usually of the form $\lambda_1/\alpha_1, \ldots, \lambda_n/\alpha_n$ which means to rename each label α_i to λ_i and each label $\bar{\alpha}_i$ to $\bar{\lambda}_i$.

As an example of CCS, we show a model for the scheduler described above in Figure 8. This example was originally written for CCS, and we adapt it here. For consistency between the example models, we use roman letters instead of the greek letters which are normally used in CCS. Since names are connected with co-names in CCS, the scheduler uses the co-names. We define the scheduler in four steps. First, we build a 'cycler' c which simply cycles between enabling (e), initiating ($\bar{s}$), and terminating ($\bar{t}$) and enabling the next one (n) in either order. We then build customized cyclers c_i for a process p_i by doing a rename operation on c ($i \oplus 1$ is

like $i+1$, except that $n \oplus 1$ is 1). This adds the subscript i to $\bar{s}$, $\bar{t}$, and e, and changes n to $\bar{e}_{i\oplus 1}$ (enable the next one). Next, we build a 'start button' g, which simply enables the first cycler and terminates. Finally, we build the scheduler by combining the start button and the customized cyclers, and removing the enable labels (e_i and $\bar{e}_i$) using the restriction operator.

CIRCAL

The CIRCAL calculus was developed by Milne ([MILNE85]) to allow modeling of simultaneously occurring events directly. Unlike the other models, in CIRCAL multiple ports may be activated at once (if the ports were physical buttons, this would be like pushing several buttons at the same time). The set of labels which label these ports are written between parentheses. For example, a typical label set is $(\alpha\,\beta\,\gamma)$ representing the simultaneous occurrence of the α, β, and γ actions.

The CIRCAL operators which describe the behavior of the agents are guarding (represented by juxtaposition), choice ($+$), nondeterminism ($\oplus$), and termination (δ). Guarding represents sequential operation. For example, $(\alpha\,\beta)\gamma$ means that the actions α and β must occur simultaneously, followed γ. The choice operator allows the agent to take different actions, the environment choosing which action will be performed. For example, the terms $(\alpha\,\beta)\epsilon$ and $\gamma\delta$ are composed to give $(\alpha\,\beta)\epsilon+\gamma\delta$, and the environment will choose whether action $(\alpha\,\beta)$ or γ (or neither) will take place. Nondeterminism is similar to choice, except that the environment has no control which of the operands will be chosen. The difference is apparent given a term such as $(\alpha\,\beta)\epsilon \oplus \gamma\delta$ where only γ is actually available. With the nondeterminism operator, γ choice may be taken, and deadlock would occur. If the choice operator were used instead, then the environment would simply choose the other alternative, and execution could continue. The termination operator Δ is used to terminate a program. For example, an agent which only performs action α and then terminates is represented by $\alpha\Delta$. As in CCS, there is no looping operator in CIRCAL, but recursion is allowed.

The operators which describe the interactions between agents are composition ($\bullet$), abstraction ($-\alpha$), hiding ($\backslash\alpha$), and relabeling ($[S]$). Intuitively, composition of two CIRCAL agents allows them to proceed independently either sequentially or concurrently, or allows them to interact with each other. For the composition $A \bullet B$, there are four possibilities: an action of A can be performed, an action of B can be performed, an action of A can be performed simultaneously with an independent action of B (one with no labels in common), or common labels of A and B can stimulate each other. For example, consider $((\alpha\,\beta)+\gamma)\bullet(\alpha+\epsilon)$. The result can be rewritten without using the $\bullet$ operator (think of this as a new agent resulting from the composition of two agents) as

$$N = \gamma(\alpha+\epsilon)+\epsilon((\alpha\,\beta)+\gamma)+(\gamma\,\epsilon)+(\alpha\,\beta)$$

$$c \Leftarrow es(tn + nt)c$$

$$c_i \Leftarrow c[s_i/s, t_i/t, e_i/e, e_{i\oplus 1}/n]$$

$$g \Leftarrow s_1\Delta$$

$$sch \Leftarrow (g \bullet c_1 \bullet \cdots \bullet c_n) - e_1 - \cdots - e_n$$

Figure 9
A Scheduler Modeled in CIRCAL

The abstraction $(-\alpha)$ and hiding $(\backslash\alpha)$ are similar in that they remove labels from ports, but they are slightly different in their operation. The abstraction operator allows the port to still be used for communicating with other agents, while the hiding operator does not. The abstraction operator removes α from all of the terms. The hiding operator removes all terms containing α. For example, using the result of the composition example N above,

$$N - \alpha = \gamma\epsilon + \epsilon(\beta + \gamma) + (\gamma\,\epsilon) + \beta$$

and

$$N\backslash\alpha = \gamma\epsilon + \epsilon\gamma + (\gamma\,\epsilon).$$

Relabeling simply changes the names of the labels in the expression, just as relabeling in CCS does.

Since CIRCAL and CCS have a common origin, the scheduler example works the same way in both models, although the operators have a different appearance. The CIRCAL version is shown in Figure 9. We again use roman letters, but co-names are not used in CIRCAL, so none of the labels need bars.

Net Based Models

The concurrent process based models described in the previous section are natural abstractions of the way concurrent systems are generally implemented. Another common method of modeling concurrent systems is to use a model based on net theory. Net-based models model concurrency and synchronization naturally, without having to arbitrarily divide the model into processes. For comparison with the parallel-process based models already described, a Coloured Petri net model (to be described shortly) of the scheduler used as an example in the previous section is shown in Figure 10.

The relationships between the net based models described here are illustrated in Figure 11. The starting point is Condition-Event systems, originally described by Petri in [PETR62]. However the model commonly referred to as 'Petri nets' today are not Condition-Event systems, but Place-Transition nets. Applying different variations to the Petri net theme, we arrive at the other models shown. The models are described individually below, but it is important to note that there are still many interesting combinations which have not been explored yet. The only models which

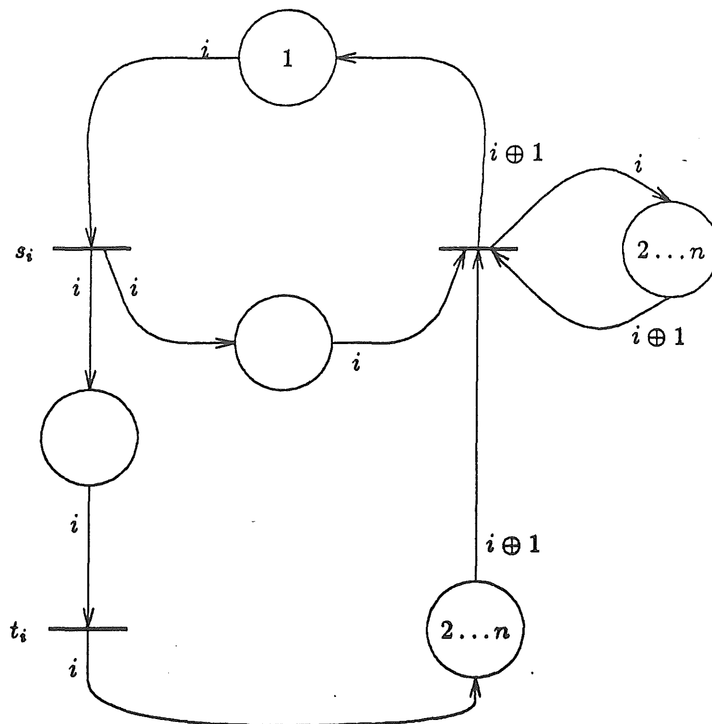

Figure 10
A Coloured Petri Net Model of a Scheduler

involve a combination of two variations are Numerical Petri nets, which combine the external memory of Predicate-Action nets and the distinguishable tokens of Coloured Petri Nets, and Razouk's Timed Petri nets, which combine the transition firing times of Ramchandani's timed Petri nets with the transition enable times of Merlin's Time Petri nets (in a limited way). Many other combinations are possible, and could be very interesting and useful, such as colored Timed Petri nets.

All of the net based models have elements in common. Nets, in general, consist of three types of components: *places*, *transitions*, and directed *arcs* which connect places to transitions and vice versa. Places can hold items called tokens; the amount and kind of tokens a place can hold are determined by the particular model. Transitions fire by removing tokens from some places and putting tokens on other places according to a firing rule. The exact details of when and how transitions fire depend, of course, on the particular model. When drawn graphically, places are drawn as circles, transitions as boxes or lines, and arcs as arrows between the circles and lines.

Condition-Event Systems

Condition-Event systems were designed to represent systems in which an event can occur if its preconditions are true; when it occurs, its preconditions are no longer true, but its postconditions are true. Conditions are modeled as places, which can either be marked with a token (if the condition is true) or be unmarked

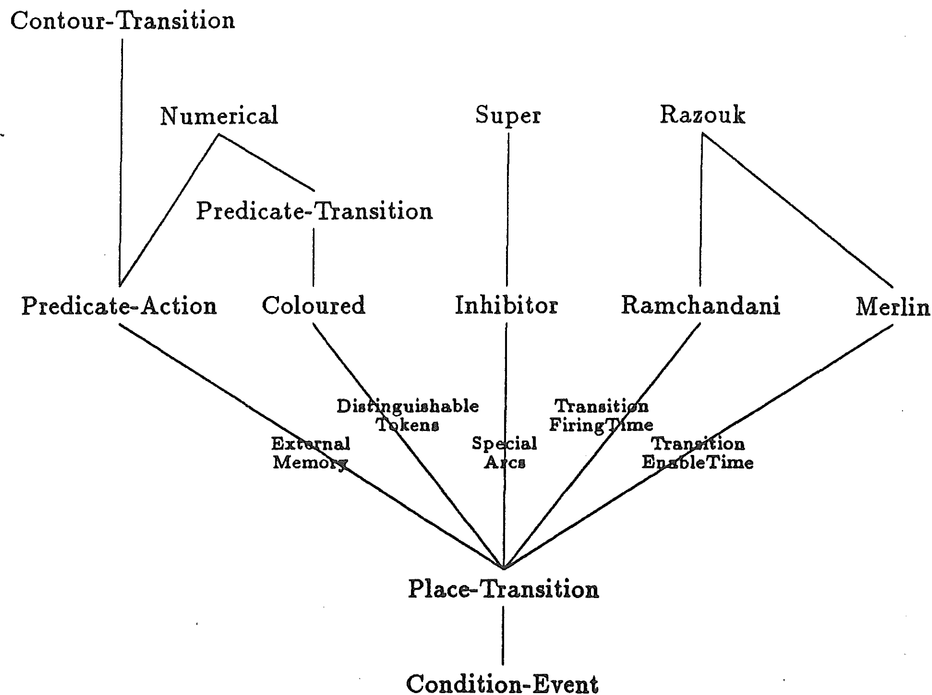

Figure 11
Relationships of Net Based Models

(if the condition is false). Events are modeled as transitions which fire when the event occurs. Preconditions for an event are modeled by arcs from the places representing the preconditions to the transition modeling the event. Post conditions are similarly modeled by arcs from transitions to places. A transition is *enabled* and may fire (an event may occur) if all of its input places are marked (all of the event's preconditions are true) and all of its output places are unmarked (all of the event's postconditions are false). An enabled transition fires by removing the token from each input place and placing a token on each output place. If multiple transitions are enabled simultaneously, the selection of which transition to fire is made non-deterministically.

An example of a Condition-Event system is shown in Figure 12. This system represents a computer system in which two devices which gather data from the outside world are serviced by a single processor. Transition t_1 models the event of the first device gathering the data, and transitions t_2 models the event of processing it. Transitions t_3 and t_4 model the same events for the other device. Places p_1 and p_3 model the conditions that the previous data gathered has been processed, places p_2 and p_4 model the conditions that new data has been obtained and is ready to process, and place p_5 models the condition that the processor is free. The arcs in the system model the constraints on the system, namely that new data can not be gathered until the previous data has been processed (p_1 (p_3) is an input place for

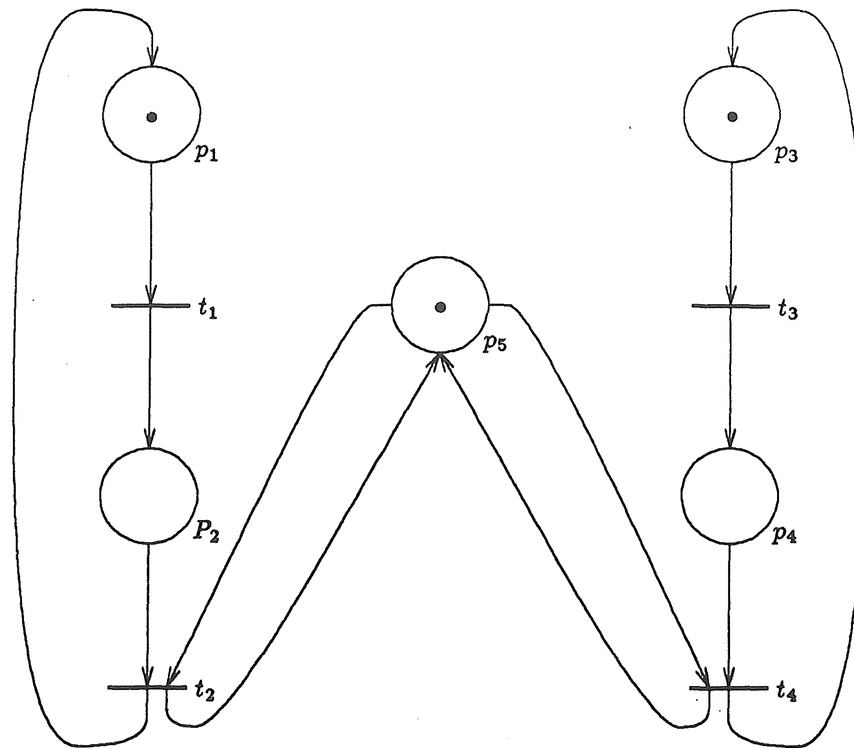

Figure 12
An Example Condition-Event System

t_1 (t_3)), and data can not be processed until both it and the processor are available (both p_2 and p_5 are input places for t_2, and p_4 and p_5 are input places for t_4).

Although this system is simple, it illustrates some of the basic properties of Condition-Event systems. Concurrency is modeled since t_1 and t_3 are independent of each other, and can be fired in either order (they can conceptually be fired at the same time, although the formal model prohibits this). The model also accurately reflects the conflict present in the modeled system; the processor must be shared by the two devices, but may not be used by both devices at the same time. If both t_1 and t_3 were to fire, leaving places p_2 and p_4 marked (modeling the fact that both devices have data ready to process), then both t_2 and t_4 are enabled and may fire. But firing either of these transitions disables the other one—the processor can only be used by one device at a time.

Place-Transition Nets

Condition-Event systems are useful, but limited in their modeling ability. Place-Transition nets (often referred to as Petri nets) extend this model by allowing a place to hold more than just one token—in fact, no limit is imposed on the number of tokens which can be in one place. A transition is enabled if each of its input places contains at least one token (regardless of whether its output places are empty or not), and fires by removing one token from each input place and placing

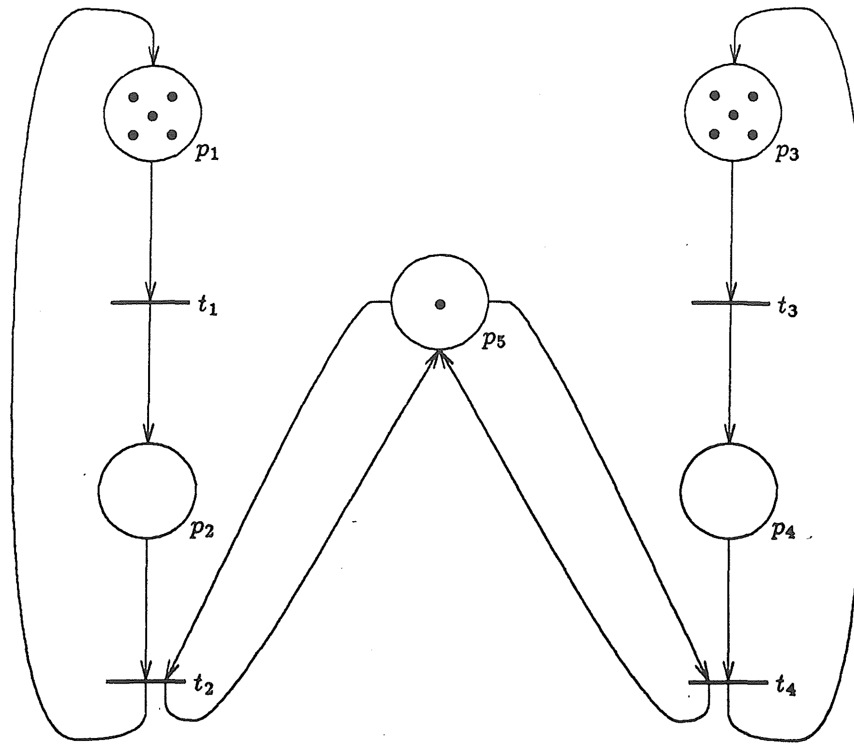

Figure 13
An Example Place-Transition net

an additional token on each output place. The model is often further extended by allowing multiple arcs between a single place and transition. The enabling and firing rules are extended accordingly—one token is moved for each arc. This extension can easily be modeled by the more basic model, so we make no distinction between them here.

The Condition-Event system in Figure 12 is also a Place-Transition net (this is not always the case due to the more restrictive enabling rule for Condition-Event systems). Suppose, however, that the devices being modeled each contain five buffers to hold the data until it can be processed. The same model can be used (considered as a Place-Transition net) by simply starting the model with five tokens each in p_1 and p_3, as shown in Figure 13. To model the five buffer system with a Condition-Event system would require additional places to represent each buffer—the Place-Transition net is much simpler.

Condition-Event systems and Place-Transition nets are equivalent in modeling capability (any system which can be modeled by one can be modeled by the other) if the Place-Transition net is bounded (no place can contain more than n tokens for some positive integer n), although Condition-Event systems may require a lot more places and transitions, and are harder to extend (consider, for example, extending the model to a ten buffer system), so Place-Transition nets are more powerful than

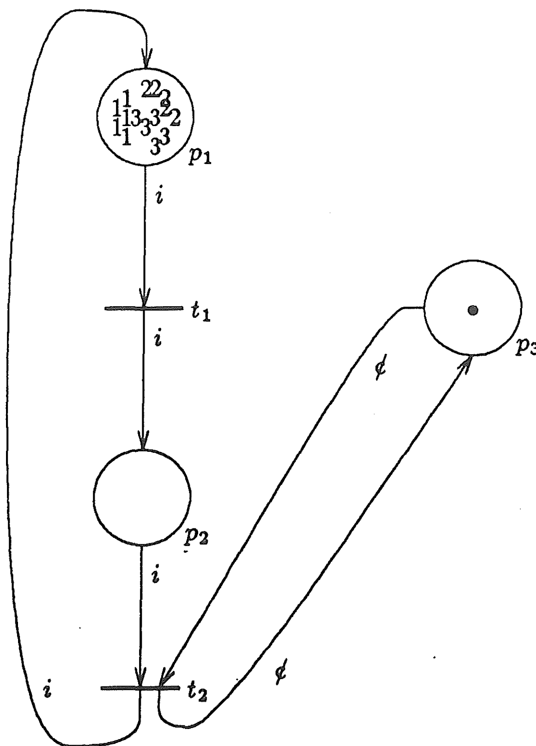

Figure 14
A Coloured Petri Net

Condition-Event systems. If the boundedness restriction is lifted, Place-Transition nets can model systems which Condition-Event systems can not, so general Place Transition nets have more modeling capability. However, this will not happen very often in attempting to model a finite real-world system, so the extra capability is mostly of theoretical interest.

Coloured Petri Nets

The tokens in Place-Transition nets are indistinguishable from each other. A way to make a more powerful model is to paint the tokens different colors to allow different tokens to be distinguished. In practice, the 'colors' are really numbers or some characteristic of the system being modeled, but the idea is the same. The transition enabling and firing rules for Coloured Petri nets are the same as for Place-Transition nets, except that each arc has an expression of one variable associated with it; when a transition is fired, each variable takes the color of the token which traverses the arc with which is it associated as its value. In order for a transition to be enabled, the expressions on the input arcs must be able to be consistently satisfied. When a transition fires, the expressions on the output arcs specify which color of token is to be placed in the output places.

To illustrate the extra power of colored tokens, consider once again the computer system of the previous examples. We extended the system before by adding

buffers to each of the devices; let us now add more devices. To extend the Place-Transition net in this way would require adding two places and two transitions for each device. This not only gets more complex, but makes the drawing hard to read since arcs are then required to cross over each other to and from the processor. This can be done easily in a coloured Petri net by combining the devices into one set of places and transitions, and distinguishing the different devices by the color of the tokens (the 'colors' are actually numbers in this example), as shown in Figure 14. Note that the token representing a free processor (in p_3) is colorless, represented along the arcs by a ¢ sign. The colored tokens are represented by numbers in the places.

Although this net is illustrative of the ability of coloured Petri nets to represent systems more compactly than place-transition nets, it is not really a very good example of the capabilities of coloured Petri nets. A better example is shown in Figure 15. It is an implementation of a sliding window communication protocol which uses sequence numbers between 0 and 7 to verify that all of the messages are received and given to the destination host in the correct order. The operator $\oplus$ represents modular arithmetic: $i \oplus j = i + j \bmod 8$.

Predicate-Transition Nets

Predicate-Transition nets extend coloured Petri nets in two ways. First, tokens can be multi-colored (i.e., they can have stripes of different colors rather than a being a single solid color). Each token is represented by an n-tuple, n being the number of colors the token has. A 0-tuple would be a colorless token (the kind used in standard colorless Petri nets), and a 1-tuple is a standard colored token (as used in coloured Petri nets). Second, a predicate may be associated with each transition which uses the variables that label the input arcs, and must be true in order for the transition to fire.

To illustrate the power of predicate-transition nets, consider extending the protocol modeled above to include several communication lines between the sender and receiver. Using a coloured Petri net, the part of the net representing the medium would have to be duplicated, with one copy for each communication line. With predicate-transition nets, all we need to do is add a second color to the tokens in the medium to represent which medium is being used. The resulting net is shown in Figure 16.

Predicate-Action Nets

Petri nets are very useful for modeling the control aspects of a system, but the data aspects are difficult to model using standard Petri nets. Although a number can be represented as the number of tokens in a place, or by the color of a token, many researchers have found it convenient to add a memory to the basic Petri net model. Predicate-Action nets are essentially Place-Transition nets which can access a memory. Associated with each transition is a predicate, which can read the memory; the transition is enabled only if the predicate is true. Note

Figure 15
Another Coloured Petri Net

that predicates are slightly different in Predicate-Transition nets and in Predicate-Action nets. Although both models require the predicate to be true, the variables

Figure 16
A Predicate-Transition Net

in Predicate-Transition nets depend only on the colors of the tokens used to enable the transition while the variables in Predicate-Action nets reference a memory. Also

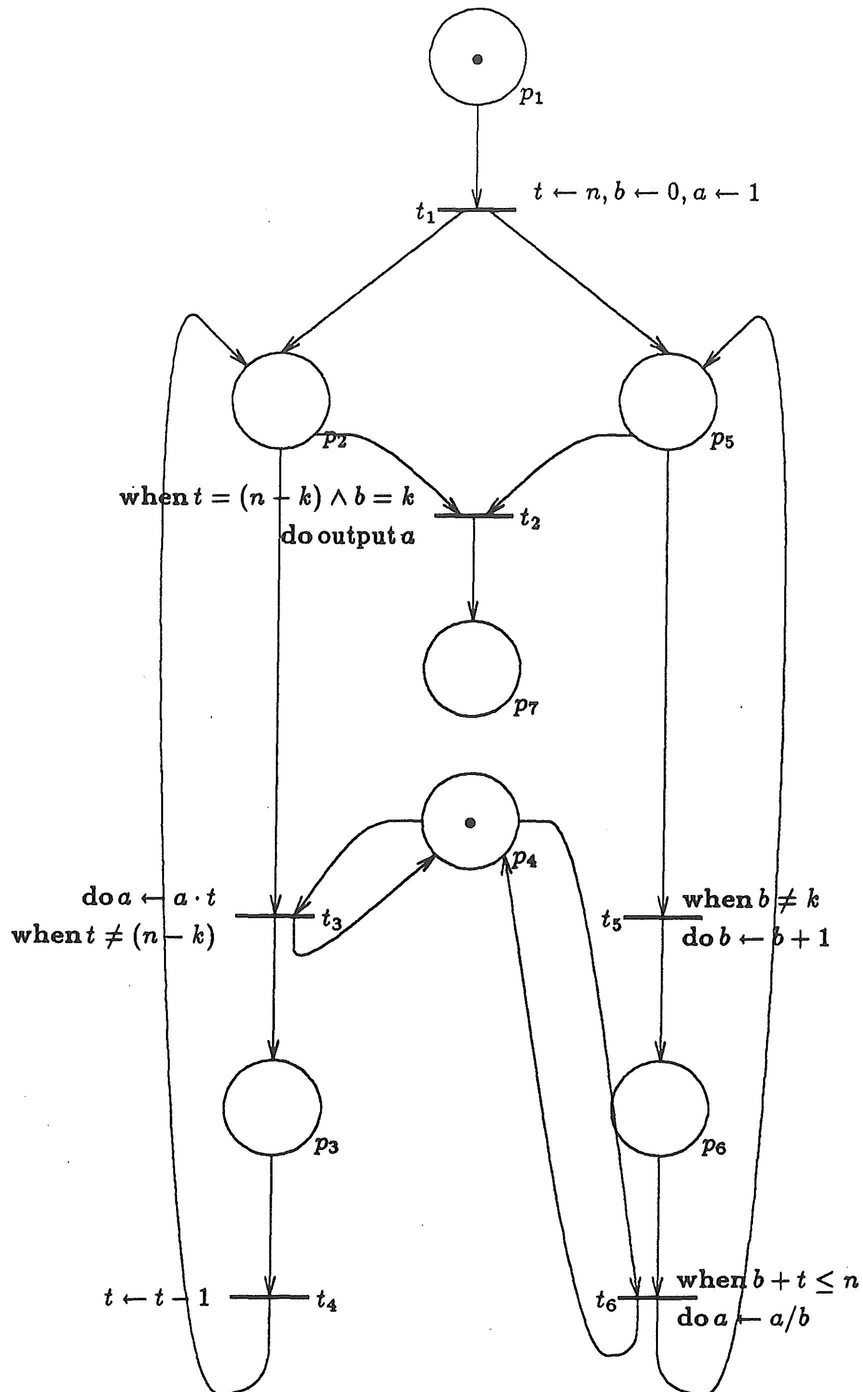

Figure 17
An Example Predicate-Action Net

associated with each transition is an action, which can both read and modify the memory; the action is performed when the transition is fired.

An example of a Predicate-Action net is shown in Figure 17. This net calculates the binomial coefficient $\binom{n}{k}$. When the net halts with a token in p_7, the variable a will contain the answer. Transitions t_3 and t_4 calculate the numerator by calculating $n \cdot \cdots \cdot (n-k+1)$. At the same time, transitions t_5 and t_6 calculate the denominator by dividing a by $1 \cdot \cdots \cdot k$. Place p_4 is used to implement mutual exclusion—transitions t_3 and t_6 both modify a, and so should not execute at the same time.

When an external memory is added to any model, it is relatively easy to simulate a RAM, so it increases the modeling capability to that of a Turing machine. Predicate-Action nets also have more modeling power than Place-Transition nets, since it adds the capability to model data easily. In fact, a Place-Transition net is simply the special case of a Predicate-Action net where all of the predicates are true and all of the actions are null.

Contour-Transition Nets

Note that all of the variables in Predicate-Action nets are global—each variable can be used by each transition. The use of global variables has the same disadvantages for Petri nets as it does for programming languages: Variable names have to be checked to ensure that another part of the net does not use them, and if a net is divided into subnets (a relatively easy thing to do), the variables used by one subnet are not safe from the effects of another. Perhaps most importantly, recursion is not possible, since the different invocations of a net would interfere with each other. Contour-Transition nets were developed to allow the use of recursion in Petri nets. Like Predicate-Transition nets, the basic Petri net model is augmented by a memory, but the variables in Contour-Transition nets are local to the subnet in which they appear. In addition, the tokens are colored to prevent different invocations of a subnet from interfering with each other. Colors may not interact with each other in Contour-Transition nets—in order for a transition to be enabled, all of the enabling tokens must have the same color. This is in sharp contrast to Coloured Petri nets, where the token colors may interact with each other to achieve the desired result.

Contour-Transition nets have the same capability as Predicate-Action nets—Turing machine capability. However, the modeling power is increased since modeling recursion with a Predicate-Action net is difficult, and because a Predicate-Action net is a special case of a Contour-Transition net.

Numerical Petri Nets

Two different methods of extending Place-Transition nets have been described so far: distinguishing the tokens and adding an external memory. Numerical Petri nets use both of these methods, resulting in a model with the advantages of both: data operations can be represented easily, and the control structure can be expressed concisely.

Compared to Predicate-Action nets, Numerical Petri nets have the same capability (that of a Turing machine), and are more powerful—the increase in power is the same as moving from a Place-Transition net to a Coloured Petri net. Compared to Coloured Petri nets, Numerical Petri nets have more capability (in the unbounded case) and are more powerful (since data can be modeled easily). It is difficult to compare the power of Numerical Petri nets and Contour-Transition nets since the colors are used differently. Numerical Petri net models in which the colors interact are difficult to model using Contour-Transition nets, and Contour-Transition net models which use recursion are difficult to model using Numerical Petri nets. The relative power depends upon the system being modeled.

Extended Petri Nets

Petri nets are very useful for modeling many systems. However, they do have some limitations which make it difficult to use them for modeling some systems. For example, there is no easy way using Place-Transition nets to model a system in which some processes have a higher priority than others; the choice of which of several enabled transitions to fire is non-deterministic—priorities are a detail which has been abstracted away. Another limitation is the inability of the presence of a token in a place to inhibit a transition from firing; this limitation prevents modeling the unbounded readers-writers problem. Allowing tokens to be distinguishable results in nets which are more compact, but which otherwise have the same limitations of Place-Transition nets. Most of the limitations can be easily removed by adding an external memory, but the external memory requires very different analysis techniques. A third method of extending Place-Transition nets addresses the limitations directly by adding special arcs which behave differently from the normal arcs.

A number of extensions of this type are described in [PETE81, pp 195–200]; only two of them will be mentioned here. Most of these extensions are equivalent in modeling capability—unbounded extended nets have the capability of Turing machines. Bounded extended nets, however, still have the same modeling capability (equivalent to a finite-state automaton), and the extensions could make building many models much easier. However, many analysis techniques for regular Petri nets will not work for extended ones, especially if the analysis is assisted by computer programs.

The earliest extension of this type to Petri nets was the addition of *inhibitor arcs* [FLYN73]; an inhibitor arc prevents a transition from firing if a place contains any tokens. They are represented by an arc with a small circle instead of an arrow on the transition end.

Another useful extension is made by assigning priorities to transitions (first proposed in [HACK76]); when multiple transitions are enabled, the one with the highest priority fires. As expected, if several transitions have the same high priority, the choice is made non-deterministically.

Super Nets

Super nets were introduced by Etzion ([Etzi83]) to study the effects of different types of arcs on the languages of Petri nets. In addition to the inhibitor arcs described above, super nets can contain *emptying arcs* and *OR-logic arcs*. Emptying arcs help enable the transition if the input place contains at least one token just like an ordinary arc, but when the transition fires, the emptying arc removes all of the tokens from the place rather than just one. This simulates a counter with a reset-to-zero feature. An OR-logic arc helps enable the transition if any of the OR-logic arc connected input places are marked (rather than all of them, as an ordinary (AND-logic) arc does). When the transition fires, one token is removed from each marked OR-logic place. This feature does not increase the capability of Petri nets by itself, although it does increase the power—to model an OR-logic place in a regular Petri net requires a duplicate of the transition for each OR-logic arc.

Ramchandani's Timed Petri Nets

There is one important item which is not included in any of the above net-based models: time. For many purposes, it is not needed for analysis, and is simply abstracted out. Indeed, for these purposes, it might be dangerous to leave time in the model since its inclusion could make the system depend on timing constraints which may change if the system being modeled were implemented in a different way. For example, a model of a communication protocol which incorporated time may accurately reflect a system in which the medium is an Ethernet, but fail to accurately model a satellite-based system. This is not meant to imply that representing time is bad—most people would not expect an Ethernet protocol to be the same as a satellite protocol—only that care must be taken when time is included in the model.

There are basically two main reasons for including time in a model. First, some things, such as timeouts, can not be conveniently modeled without time. Consider, for example, the communication protocol model in Figure 15. Since the coloured Petri net model does not include time, the loss of a message must be one of the enabling conditions for the transition which represents the timeout to fire. This is not a very realistic model; if the sender could detect that a message was lost in this manner, there would be no need for the receiver to acknowledge the message! A more realistic model for the protocol would omit place *messages lost* and the arcs connected to it. However, without a way to regulate the firing of the timeout transition, multiple copies of a message could flood the system and ruin the protocol. Timeouts are also essential in the design and analysis of fault-tolerant systems.

The other reason for including time in a model is to allow analyzing the performance of the system. Although some rudimentary performance statistics may be obtained for time-less models in the form of relative transition firing frequencies,

they are not very meaningful. The events modeled by some transitions take longer than others, and there is no way of specifying this except by adding time.

There are many ways of adding time to a Place-Transition net (or any of its variations); we will consider two of them here: specifying the time it takes for a transition to fire, and specifying how much time must elapse before an otherwise enabled transition may fire. The first approach is generally the one taken by researchers interested in performance analysis. Ramchandani was the first to associate a firing time with each transition ([RAMC74]). Other researchers who used essentially the same method were Ramamoorthy and Ho ([RAMAMO80]), Zuberek ([ZUBE80]), Godbersen ([GODB80]), and Magott ([MAGO84]).

Petri nets in which transitions take time to fire are called *Timed* Petri Nets by most researchers. This is in contrast to Petri nets in which transitions fire instantaneously after being enabled for some time, which are generally called *Time* Petri nets. The one letter difference has not been noticed by all researchers, however, and there are other ways of adding time to Petri nets (see, for example, [SIFA77]), so beware when interpreting a reference to timed or time Petri nets as a reference to a particular model.

Timed Petri nets differ from regular Petri nets in the way transitions fire. Rather than removing tokens from the input places and adding tokens to the output places instantaneously, these two operations are separated, the the firing time associated with the transition elapses between them, when the transition is said to be firing. While a transition is firing, other transitions may also begin to fire (and finish firing). This tends to complicate the analysis slightly since a state of the Timed Petri net must then include not only the Petri net marking, but also which transitions are currently firing, and how much time remains before they finish firing.

Merlin's Time Petri Nets

A slightly different way of adding time to Petri nets was used by Merlin. In Time Petri nets, transitions fire instantaneously, as in regular Petri nets. However, there are two timing constraints associated with each transition: $t_{\min}$ and $t_{\max}$. Before the transition can fire, it must be enabled for at least $t_{\min}$ time. If the transition remains enabled for $t_{\max}$ time, then the transition must fire.

Note that time Petri nets include untimed Petri nets as a special case. When $t_{\min}$ is 0 and $t_{\max}$ is ∞, the behavior is exactly the same as for untimed Petri nets. It is also quite easy to simulate a timed Petri net with a time Petri net. Each transition must be replaced by two transitions and a place, essentially dividing the timed Petri net transition into three stages: the transition starts firing (the first replacement transition fires), the transition is firing (the replacement place is marked), and the transition finishes firing (the second replacement transition fires). Time Petri nets can also simulate a Petri net with inhibitor arcs, and thus have the

same capability as a Turing machine. Time Petri nets have thus more capability and more power than either regular Petri nets or timed Petri nets.

Razouk's Timed Petri Nets

One of the major disadvantages of using Ramchandani's Timed Petri nets for modeling concurrent systems, especially communication protocols, is the lack of an easy way to model timeouts. Merlin's Time Petri nets can model timeouts easily, but their non-determinism makes it difficult to analyze their performance. Razouk proposed a Timed Petri net model which combines the analyzability of Ramchandani's Timed Petri nets with a timeout modeling capability.

In Razouk's Timed Petri nets, each transition has two times associated with it: an enable time and a firing time. The firing time is just as in Ramchandani's model: when a transition is fired, it begins firing, waits for the firing time, and then finishes firing. The enable time is like the min and max times of Merlin's model: a transition fires after it has been enabled for the enable time. Note that the non-determinism inherent with Merlin's range of times has been removed by using a single time; a transition will fire after that time has elapsed unless a conflicting transition fires instead. The remaining non-determinism has been removed by associating a relative firing probability with each transition. This firing probability is used to determine which of several conflicting transitions will fire; non-determinism is replaced by randomness. The lack of non-determinism makes it impossible for Razouk's timed Petri nets to model an untimed net, but for the systems it can model, it makes analysis easy.

Analysis Techniques

Having described several modeling techniques, we now discuss some methods used for analyzing the models. One purpose of analyzing a model of a concurrent system is to determine whether or not the modeled system has certain properties. These properties can be general properties (applicable to all concurrent systems, e.g., deadlock and starvation freedom), model-specific properties (applicable to certain models, e.g., Petri net conservatism), or system-specific (applicable to specific systems, e.g., distributed data base consistency). The ultimate goal of analyzing a model of a concurrent system is of course to show that the system will work correctly, or to determine why it will not work in order to make appropriate modifications. As the correct operation of a system is determined by its specification, it seems appropriate to discuss specification techniques briefly.

There are a number of ways to specify the behavior a system should exhibit. One way, of course, is to use natural language, but formal analysis requires a formal specification. Another method is to use a modeling technique and actually build a prototype of the system, for example, a Petri net; any other implementation of the system should have the same behavior as this model. In order to show that a model of a new implementation meets its specification, the analyst shows that this model is equivalent to the specification model. Another very common specification technique

is to base the specification on an analysis technique, for example, temporal logic. To verify a model against a specification of this type generally involves using the analysis technique on which the specification is based in a natural manner.

The details of analyzing a particular model depend, of course, on the details of the model. However, the most common analysis techniques can be classified into general categories. This section will describe these categories, and how the techniques in each category are used in general. Some specific examples will be given, but we do not discuss the application of every analysis technique to every modeling technique for several reasons. First, there are just too many combinations, and there is not enough room to do it. Second, the application of an analysis technique to a different model is often a straightforward modification. Third, every analysis technique has not been applied to every modeling technique—either it can not be done easily or no one has bothered doing it (or no one has bothered writing a paper on the application). Table 1 gives references for the applications of analysis to modeling techniques which have been described in the literature. Blank spaces represent possible topics of future research.

State-Space Analysis

Also known as *reachability analysis*, state space analysis is done by enumerating all of the states which a model can reach from its initial state, and examining them for desired properties. For example, if a system to be free from deadlock, each state should have a successor. For some models this is easy—the abstraction present in the model makes the number of states manageably small. However, for many models there are too many states (perhaps even an infinite number) to enumerate. State space analysis can often still be used in this case; the number of states is simply reduced by combining many states with common properties into one state. For example, an unbounded Petri net has an infinite reachability graph, but its coverability graph is finite and may be used to help analyze the net.

One advantage to this technique is that the state enumeration can usually be automated. This has been done for many models, including Petri nets ([MORG85]) and Time Petri nets ([BERTHO83]). Taylor also shows how to use build a reachability graph (which he calls a Concurrency State Graph) for Ada programs in [TAYL83]. This technique should also work for CSP models.

As an example of state space analysis, consider the Petri net in Figure 18. Several facts are obvious from an examination of the reachability graph, which is also shown. First, it is possible for this system to deadlock: the reachability graph has two nodes which have no successors: $(0, 2, 0)$ and $(0, 0, 2)$. If the system ever reaches one of these states, it must halt. The paths which lead to these states can also be obtained from the graph: firing either t_1 or t_2 twice in a row will do it. It is also apparent that the system is bounded—no place ever contains more than two tokens. This is important, as it is difficult to implement an unbounded system in the real world. Another property which this system has is that it is conservative—the sum of the number of tokens in each place is always two.

Model	Reference	State Space	Temporal Logic	Structural Analysis	Inductive Analysis	Language	Reduction	Other
CSP	[HOAR78]	[TAYL83]	[HAIL80]	[SOUN84]	[LAMP84]			[DENI84]
Path Expressions	[CAMPB74]			[LAUE79]	[SHIE79]	[LAUE81]		
CCS	[MILNER80]		[KARJ85]	[MILNER80]	[MILNER80]			
CIRCAL	[MILNE85]			[MILNE85]				
Condition-Event System	[GENR80]			[GENR80]	[QUEI80]	[MERC84]		[MERC84]
Place-Transition Net	[PETE81]	[MORG85]	[GENR80]	[PETE81]	[MULL80]	[PETE81]	[SUZU83]	[DATT84]
Coloured Petri Net	[JENS81]			[JENS81]				
Predicate-Transition Net	[GENR79]		[GENR80]	[GENR79]	[BERTHE82]			
Predicate-Action Net	[KELL76]		[PNUE79]		[KELL76]		[KWON82]	
Contour-Transition Net	[ROSE84]				[ROSE84]			
Numerical Petri Net	[SYMO80]	[SYMO80]				[BILL85]		
Inhibitor Petri Net	[PETE81]							[BAUM85]
Ramchandani's Timed Net	[RAMC74]			[MAGO84]				
Merlin's Time Petri Net	[MERL74]	[BERTHO83]						
Razouk's Timed Petri Net	[RAZO83]	[RAZO83]						

Table 1

References to Modeling and Analysis Techniques

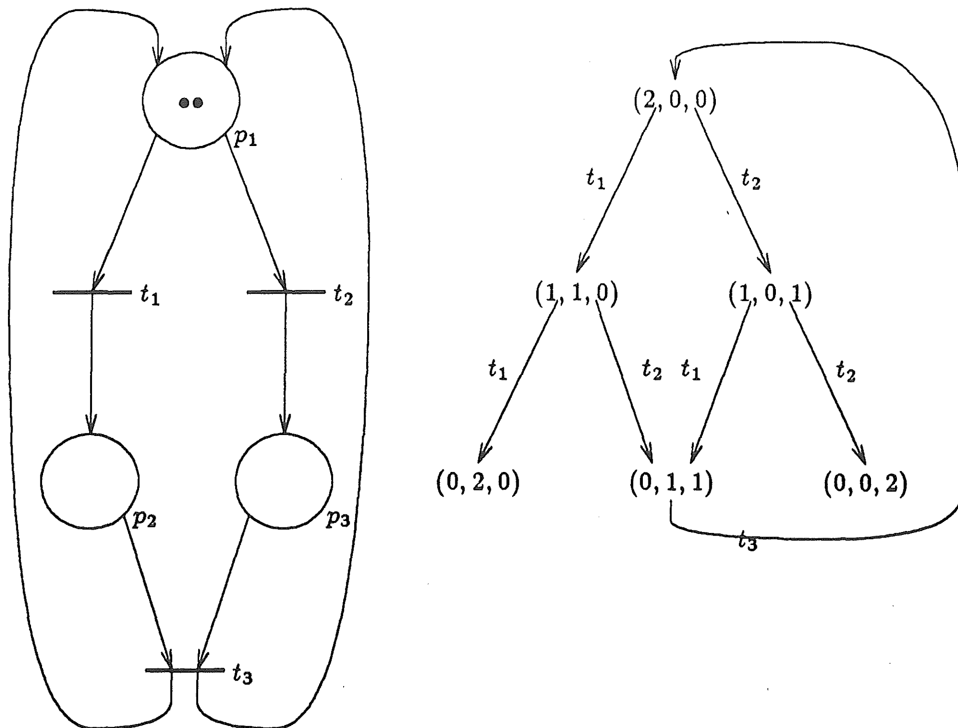

Figure 18
A Petri Net and its Reachability Graph

Temporal Logic

Temporal logic was first introduced in [PNUE77] and has many proponents today. It is an extension of formal logic, adding such operators as $\Box x$, meaning that x is true from now on, and $\Diamond x$, meaning that x will be true sometime in the future.

As an example of how temporal logic is used to analyze concurrent systems, consider the Predicate-Action net which computes a binomial coefficient in Figure 17. There are a number of assertions we can make about this system using temporal logic. For example, partial correctness[1] would be expressed using the statement $\Box\, \mathit{marked}(p_7) \supset a = \binom{n}{k}$. We expect this particular concurrent system to terminate—this is expressed as $\Diamond\, \mathit{marked}(p_7)$. We could also express that the computation is clean by the statement $\Box a = \lfloor a \rfloor$. These statements can be proven using the same techniques as used in normal logic. As an example, we will prove the cleanness condition that a will always be an integer. The condition is true to begin with ($a \leftarrow 1$), and the only way it can be made false is if the division in the action of t_6 does not yield an integral result. When t_6 fires, a will contain the partial result:

$$a = \frac{n \cdot (n-1) \cdot \cdots \cdot t}{1 \cdot 2 \cdot \cdots \cdot (b-1)}$$

[1] Partial correctness means that if the program terminates, the correct result will be produced, but does not imply that the program will terminate

or

$$a = \frac{n \cdot (n-1) \cdot \dots \cdot (t+1)}{1 \cdot 2 \cdot \dots \cdot (b-1)}$$

depending upon whether t_4 has fired or not. In the first case, the numerator is the product of $n - t + 1$ consecutive integers; in the other case, the numerator is the product of $n - t$ consecutive integers. In either case, we know that the product of i consecutive integers is always divisible by $i!$, so the numerator is divisible by $n - t$. Because of the condition on t_6, we also know that $b + t \leq n$, so $b \leq n - t$, so the numerator is divisible by $b!$. Since after the division, we will have divided it by $b!$, the result will always be an integer, so a is always equal to $\lfloor a \rfloor$.

Structural Analysis

A very common method of analyzing a model is to use information on the structure of the model. Naturally, the exact details of how this is done depend upon the model itself. For example, the firing rule of a Petri net can be expressed as a matrix (called the *incidence matrix*) whose rows represent the transitions and whose columns represent the places. For example, the incidence matrix of the Petri net in Figure 18is:

$$C = \begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \end{array} \begin{array}{c} \begin{array}{ccc} p_1 & p_2 & p_3 \end{array} \\ \begin{pmatrix} -1 & +1 & 0 \\ -1 & 0 & +1 \\ +2 & -1 & -1 \end{pmatrix} \end{array}.$$

If y is a solution to the system of equations $C \cdot y = 0$, then y is an invariant. For example, the system of equations is:

$$\begin{aligned} -y_1 + y_2 &= 0 \\ -y_1 + y_3 &= 0 \\ 2y_1 - y_2 - y_3 &= 0. \end{aligned}$$

Although there are an infinite number of solutions, they are all equivalent to $(1,1,1)$, the only invariant for this net (most nets will have more). This invariant means that the sum of the number of tokens in each of the places will remain constant, or that the Petri net is conservative. These invariants can be determined automatically (see [MART81]), and can be used to prove useful properties about the system (it is hard to show useful properties about this system since the system is not very useful).

As another example of structural analysis of a model, we choose a very different model, CIRCAL, and show an example taken from [MILNE85]. Suppose two agents A and B wish to access a shared resource using α and β labels, respectively. The constructed system is to send a sequence of two α or two β signals to the resource. The specification of the behavior of our system is:

$$\text{SPEC} \Leftarrow \alpha\alpha\text{SPEC} \oplus \beta\beta\text{SPEC}.$$

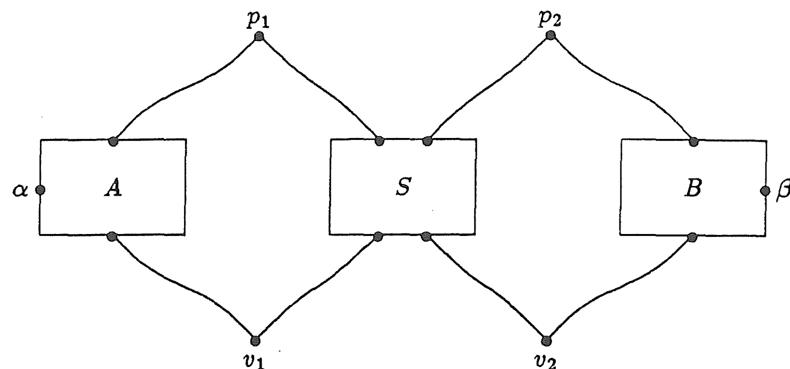

Figure 19
A CIRCAL System

The system can interact with either of the two agents, but the environment has no control over which, so the nondeterminism operator is used. Agent A has the behavior:

$$A \Leftarrow p_1 \alpha\alpha v_1 A$$

and B is similar:

$$B \Leftarrow p_2 \beta\beta v_2 B.$$

The p_i and v_i labels interact with a semaphore to guard the critical sections $\alpha\alpha$ and $\beta\beta$, as these sections must interact mutually exclusively with the resource. The semaphore is defined by

$$S \Leftarrow p_1 v_1 S + p_2 v_2 S.$$

The system is constructed by composing the three agents,

$$C \Leftarrow A \bullet B \bullet S$$

and is shown in Figure 19.

We first derive an equation for the system as a single agent by removing the dot operators:

$$C = p_1 \alpha\alpha v_1 C + p_2 \beta\beta v_2 C.$$

We then remove the p_1 label by abstraction (the rule used to do so is given in [MILNE85]):

$$C - p_1 = \alpha\alpha v_1 C - p_1 + p_2 \beta\beta v_2 C - p_1 \oplus \alpha\alpha v_1 C - p_1.$$

By similarly removing the p_2, v_1, and v_2 labels, we get:

$$C - p_1 - p_2 - v_1 - v_2 = \alpha\alpha C - p_1 - p_2 - v_1 - v_2 \oplus \beta\beta C - p_1 - p_2 - v_1 - v_2,$$

which matches our specification.

Inductive Analysis

A method which is often very useful for analyzing a concurrent system is to use induction. This is done by determining an invariant of the system, an equation which is always true during system execution, and using induction to prove that the invariant always holds, no matter what the system does. As an example, consider once again the Predicate-Action net of Figure 17 which computes the binomial coefficient $\binom{n}{k}$. We wish to prove partial correctness, i.e., that the equation $marked(p_7) \supset a = \binom{n}{k}$ is always true. We wish to do this by showing that if it is true before firing any transition, that it is still true after the transition has been fired. We immediately encounter a problem—it can not be done because the equation is too strong. It is a true invariant of the system, but it is not an *inductive invariant*, and thus can not be proven by induction. The solution is to weaken the invariant so that it becomes an inductive invariant whose truth implies the truth of the original equation. Doing this, we get the equation:

$$\left(marked(p_7) \wedge a = \binom{n}{k}\right) \vee$$

$$\left(marked(p_2) \wedge marked(p_5) \wedge a = \frac{n \cdot (n-1) \cdot \cdots \cdot (t+1)}{1 \cdot 2 \cdot \cdots \cdot b}\right) \vee$$

$$\left(marked(p_2) \wedge marked(p_6) \wedge a = \frac{n \cdot (n-1) \cdot \cdots \cdot (t+1)}{1 \cdot 2 \cdot \cdots (b-1)}\right) \vee$$

$$\left(marked(p_3) \wedge marked(p_5) \wedge a = \frac{n \cdot (n-1) \cdot \cdots \cdot t}{1 \cdot 2 \cdot \cdots \cdot b}\right) \vee$$

$$\left(marked(p_3) \wedge marked(p_6) \wedge a = \frac{n \cdot (n-1) \cdot \cdots \cdot t}{1 \cdot 2 \cdot \cdots \cdot (b-1)}\right) \vee marked(p_1).$$

Notice that if this weak invariant is true that the strong invariant is also true. To prove that is it always true, observe first that it is true for the initial marking of the net. The initial marking has a token in p_1, and this is one of the terms of the invariant. Now consider what happens when t_1 is fired. The last term is no longer true, but now both p_2 and p_5 are marked, making the second term the only one which can possibly be true. The last factor in the term is true by the initial conditions set up by firing t_1; since neither the numerator nor the denominator have any factors for the product, the product is 1, which is the initial value for a. Next, consider firing transition t_2. It can only fire when places p_2 and p_5 are marked, so we know that $a = (n \cdot \cdots \cdot (t+1))/(1 \cdot \cdots \cdot b)$. We also know that $t = n - k$ and $b = k$ because of the predicate on t_2. Putting these together, we have:

$$a = \frac{n \cdot (n-1) \cdot \cdots \cdot (n-k+1)}{1 \cdot 2 \cdot \cdots \cdot k}$$

which is exactly the definition of $\binom{n}{k}$. We can continue by doing the same thing for the other transitions, and prove that the invariant is true, and thus that the

Place-Action net is partially correct; proving that the other transition preserve the truth of the invariant is left as an exercise for the reader.

Language Theory

Most models can be made to either generate or recognize a formal language, which characterizes the modeled system. If the specification can be specified as a language (this is one approach to communication protocol specification, see for example [Haas85]), then the analyst can try to show that the two languages are equivalent. Even if the specification is not a language, the language of the modeled system can be checked for certain properties. For example, in the language of a model of a communication protocol, if s represents a message being sent and r represents a message being received, the language could be checked to make sure that in every string of the language, the occurrence of an s is followed by an r.

Reductions

Analysis by reduction involves changing the model of the system to an equivalent but simpler model by replacing parts of it by simpler but equivalent ones. Information is often lost when doing so, so the analyst must make sure that the information is not important for the analysis. Reduction is essentially a way of increasing the level of abstraction of the model, and is often combined with other analysis techniques to make the model more manageable.

Simulation

Analysis by simulation of the model is generally not as powerful as the other techniques, since it generally does not check all possible paths through the model. Like testing a computer program, it can only show the presence of errors, not the lack thereof. However, like testing, simulation has its uses. It is often faster and easier than other techniques, so it can be used as a first approach to give the analyst confidence that the model is working properly, or to find obvious errors quickly. When simulation shows that the model seems to be correct, then other methods can be used to prove this. Simulation is also useful for performance analysis since performance statistics can be kept during the simulation.

Other Analysis Techniques

Occasionally, a researcher develops a technique which does not fit any of the above categories. For example, [Herz80] describes a technique for analyzing Petri nets using graph theory. We will not describe any of them in detail here, but a few references are listed in Table 1.

Summary

We have presented several modeling and analysis techniques in this paper; to finish, we now review some of the advantages and disadvantages of each method.

Communicating Sequential Processes were developed as a concurrent programming language, and not primarily as a modeling technique. It has been presented

here because it can be used for modeling, and because many analysis techniques have been developed for it. A few of these are explained in [APT80], [LAMP84], and [LEVI81]. One of the main advantages of CSP is that many concurrent programming languages are based on it, and systems which have been modeled in CSP will probably be quite easy to implement in software. A disadvantage is that a process must know the name of each process with which it communicates—it is impossible, for example, to have a generic process which performs a service for whatever processes need it. For this reason, it is the least powerful of the communicating/synchronizing process models we have discussed.

Path expression models are also quite easy to implement in software—several programming languages exist which use path expressions as a synchronization technique, among them Path Pascal, described in [CAMPB79]. Although server processes in the path expression model do not need to know the name of every process who uses them, thus making them more powerful than CSP, they do have a disadvantage which makes them inappropriate for modeling some systems: it is difficult to express synchronization conditions which depend on the current state of a resource rather than the history of operations on it. For example, consider the readers/writers problem in which a number of processes need to access a resource such as a database, some to read only and some to write to it. Several readers may access the resource concurrently, but writers must have exclusive access. Suppose we wish to implement a readers/writers system in which the writers have precedence over the readers—no readers must be allowed access unless no writers wish access. This system is difficult to model using path expressions since the question of whether a reader will be granted access depends on the state of the system at that moment, not on the past history.

The Calculus of Communicating Systems was developed by Milner for the purpose of modeling and analyzing concurrent systems consisting of processes which communicate with each other. It is a powerful technique—the most powerful we have described—and works well for this purpose. Its main disadvantage is that it is sometimes hard to read and understand a CCS model.

CIRCAL and CCS have a common origin, concurrent processes, described in [MILNE79]. The main differences are that CIRCAL does not allow processes to communicate values to each other, while CCS does, and that CIRCAL allows several ports to be activated at once, representing true concurrency rather than the arbitrary interleaving of activations which simulates concurrency in a sequential way. For many systems, true concurrency does not happen; for the ones in which it does, such as hardware systems (like VLSI design), CIRCAL seems to be a good model.

Condition-Event systems do not appear in the literature as much as Place-Transition nets (Petri nets), but there is a good reason: Analysis techniques for Petri nets will work just fine on Condition-Event systems, so most researchers have concentrated their work on the more general case. Many Petri net models

of real systems are safe (no place ever contains more than one token at a time), and are therefore Condition-Event systems, even if the analyst calls them Petri nets. The main advantage of Petri nets (and other net-based models) over the synchronizing/communicating process models is that concurrency is inherent in the Petri net model—the person modeling a system does not need to determine in advance how many processes will be needed or how to distribute the tasks among them. Petri nets also represent the conflict present in systems very nicely. Perhaps their main disadvantage is that it may be difficult to convert a Petri net model to an efficient real implementation using multiple processors which communicate with each other. Another advantage which should not be overlooked is the fact that Petri nets are currently quite popular, and many researchers are developing tools to help automate the analysis of Petri net based models. With the availability of these tools, using Petri nets to model a system may decrease the total effort needed to analyze a system, even if a different modeling technique would be more appropriate for the particular system.

The various Petri net extensions increase the power of Petri nets to make it easier to create models. The problem with making the modeling process easier is that this often makes the analysis more difficult. For example, several automated tools exist which generate the reachability graph of a Place-Transition net. If the person modeling a system uses a coloured Petri net to make the model smaller, he may not be able to use these tools to generate a reachability graph for it. One solution to this problem is to use a pre-processor to convert the coloured Petri net to a Place-Transition net, for which a reachability graph may be generated. However, then the interface to the tools which he uses to analyze the reachability graph will use the terminology (place names, etc.) of the Place-Transition net, not the original coloured Petri net—in fact the whole idea of coloured tokens will not be available during the analysis.

The only Petri net variations described here which do not increase their power are the Timed Petri net models. These models actually decrease the modeling power by removing non-determinism. The advantage of doing so is that a type of analysis not possible on regular or extended Petri nets may be done on them: performance analysis. That, of course, is the reason for which they were designed.

There are many other modeling techniques for concurrent systems which have not been discussed here. Many of these techniques are simply variations on the techniques which have been discussed; for example, Milner describes a synchronous version of his CCS in [MILNER83]. Many other techniques have not been widely published, for example Milne's dot calculus, a forerunner of CIRCAL, described in [MILNE78]. The fact that researchers continue to develop new modeling techniques indicates that there is no ideal modeling technique for all concurrent systems. The techniques which we have described seem to be the most popular and interesting ones at the present time. Meanwhile, the search for new and better techniques goes on.

References

[Apt80] K.R. Apt, N. Francez, and W.P. De Roever, "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 3, pp. 359–385, July, 1980.

[Baum85] B. Baumgarten, P. Ochsenschläger, and R. Prinoth, "Building Blocks for Distributed System Design," *Fifth International Workshop on Protocol Specification, Verification, and Testing*, Toulouse-Moissac, France, 1985, pp. 1-3–1-22.

[Berthe82] G. Berthelot and R. Terrat, "Petri Nets Theory for the Correctness of Protocols," *Protocol Specification, Testing, and Verification*, C. Sunshine, Ed., North-Holland, 1982, pp. 325–342.

[Bertho83] B. Berthomieu and M. Menasche, "An Enumerative Approach for Analyzing Time Petri Nets," *Information Processing '83*, Proceedings of IFIP 9th World Computer Congress 1983, Paris, 1983, pp. 41–46.

[Bill85] J. Billington and M.C. Wilbur-Ham, "Automated Protocol Verification," *Fifth International Workshop on Protocol Specification, Verification, and Testing*, Toulouse-Moissac, France, 1985, pp. 2-11–2-22.

[Campb74] R.H. Campbell and A.N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science 16*, New York: Springer-Verlag, 1974, pp. 89–102.

[Campb79] R.H. Campbell and R.B. Kolstad, "Path Expressions in Pascal," *Proceedings of the 4th International Conference on Software Engineering*, Munich, September 17–19, 1979, New York: IEEE, 1979, pp. 212–219.

[Datt84] A. Datta and S. Ghosh, "Synthesis of a Class of Deadlock-Free Petri Nets," *Journal of the Association for Computing Machinery*, vol. 31, no. 3, pp. 486–506, July, 1984.

[DeNi84] R. De Nicola and M.C.B. Hennessy, "Testing Equivalences for Processes," *Theoretical Computer Science*, vol. 34, pp. 83–133, 1984.

[Etzi83] T. Etzion and M. Yoeli, "Super-Nets and Their Hierarchy," *Theoretical Computer Science*, vol. 23, pp. 243–272, 1983.

[FLYN73] M. Flynn and T. Agerwala, "Comments on Capabilities, Limitations, and 'Correctness' of Petri Nets," *Computer Architecture News*, vol. 2, no. 4, December, 1973.

[GENR79] H.J. Genrich and K. Lautenbach, "The Analysis of Distributed Systems by Means of Predicate/Transition Nets," *Semantics of Concurrent Computation*, G. Kahn, Ed., Lecture Notes in Computer Science 79, Berlin: Springer-Verlag, 1979, pp. 123–146.

[GENR80] H.J. Genrich, K. Lautenbach, and P.S. Thiagarajan, "Elements of General Net Theory," *Net Theory and Applications*, W. Brauer, Ed., Lecture Notes in Computer Science 84, New York: Springer-Verlag, 1980.

[GODB80] H.P. Godbersen, "On the Problem of Time in Nets," *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 52, New York: Springer-Verlag, 1982, pp. 23–30.

[HAAS85] O. Haas, "Formal Protocol Specification Based on Attribute Grammars," *Fifth International Workshop on Protocol Specification, Verification, and Testing*, Toulouse-Moissac, France, 1985.

[HACK76] M. Hack, "Petri Net Languages," Technical Report 159, Massachusetts Institute of Technology: Laboratory for Computer Science, March, 1976.

[HAIL80] B.T. Hailpern, and S.S. Owicki, "Verifying Network Protocols Using Temporal Logic," Ph.D. Dissertation, Technical Report No. 192, Departments of Electrical Engineering and Computer Science, Stanford University: Computer Systems Laboratory, June, 1980.

[HERZ80] O. Herzog, "Graph-Theoretical Analysis of a Subclass of Petri-Nets," *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 52, New York: Springer-Verlag, 1982, pp. 178–182.

[HOAR78] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, August, 1978.

[JENS81] K. Jensen, "Coloured Petri Nets and the Invariant-Method," *Theoretical Computer Science*, vol. 14, pp. 317–336, 1981.

[KARJ85] G. Karjoth, "An Interactive System for the Analysis of Communicating Processes," *Fifth International Workshop on Protocol Specification, Verification, and Testing*, Toulouse-Moissac, France, 1985, pp. 3-1–3-12.

[KELL76] R.M. Keller, "Formal Verification of Parallel Programs," *Communications of the ACM*, vol. 19, no. 7, pp. 371–384, July, 1976.

[KWON82] Y.S. Kwong, *On Reductions and Livelocks in Asynchronous Parallel Computation*, Ann Arbor, Michigan: UMI Research Press, 1982.

[LAMP84] L. Lamport and F.B. Schneider, "The "Hoare Logic" of CSP, and All That," *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 281–296, April, 1984.

[LAUE79] P. Lauer, P.R. Torrigiani, and M.W. Shields, "COSY—A System Specification Language Based on Paths and Processes," *Acta Informatica*, vol. 12, pp. 109–159, 1979.

[LAUE81] P.E. Lauer, M.W. Shields, and J.Y. Cotronis, "Formal Behavioural Specification of Concurrent Systems Without Globality Assumptions," Peniscola, Spain.

[LEVI81] G.M. Levin and D. Gries, "A Proof Technique for Communicating Sequential Processes," *Acta Informatica*, vol. 15, pp. 281–302, 1981.

[MAGO84] J. Magott, "Performance Evaluation of Concurrent Systems Using Petri Nets," *Information Processing Letters*, vol. 18, no. 1, pp. 7–13, 20 January, 1984.

[MART81] J. Martínez and M. Silva, "A Simple and Fast Algorithm to Obtain All Invariants of a Generalised Petri Net," *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 52, New York: Springer-Verlag, 1982, pp. 301–310.

[MERC84] A. Merceron, "A Study of Some Dependencies Between a Concurrent System and its Processes Applied to Petri Nets," *First International Conference on Computers and Applications,*, Beijing, China, 1984, pp. 414–422.

[MERL74] P. Merlin, "A Study of the Recoverability of Computing Systems," Ph.D. Dissertation, University of California, Irvine: Department of Information and Computer Science, 1974.

[MILNE78] G. Milne, "A Mathematical Model of Concurrent Computation," Ph.D. Dissertation, Edinburgh, Scotland: University of Edinburgh, 1978.

[MILNE79] G. Milne and R. Milner, "Concurrent Processes and Their Syntax," *Journal of the ACM*, vol. 26, no. 2, pp. 302–321, 1979.

[MILNE85] G.J. Milne, "CIRCAL and the Representation of Communication, Concurrency, and Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 2, pp. 270–298, April, 1985.

[MILNER80] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Berlin: Springer-Verlag, 1980.

[MILNER83] R. Milner, "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science*, vol. 25, no. 3, pp. 267–310, 1983.

[MORG85] E.T. Morgan and R.R. Razouk, "Computer-Aided Analysis of Concurrent Systems," Technical Report 85-06, University of California, Irvine: Department of Information and Computer Science, February, 1985.

[MULL80] H. Müller, "Inductive Assertions for Analyzing Reachability Sets," *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 52, New York: Springer-Verlag, 1982, pp. 168–171.

[PETE81] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[PETR62] C.A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Bonn, West Germany: University of Bonn.

[PNUE77] A. Pnuelli, "The Temporal Logic of Programs," *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science, Providence*, 1977, pp. 46–57.

[PNUE79] A. Pnuelli, "The Temporal Semantics of Concurrent Programs," *Semantics of Concurrent Computation*, G. Kahn, Ed., Lecture Notes in Computer Science 70, Berlin: Springer-Verlag, 1979, pp. 1–20.

[QUEI80] J.P. Queille and J. Sifakis, "Iterative Methods for the Analysis of Petri Nets," *Application and Theory of Petri Nets*, Lecture Notes in Computer Science 52, New York: Springer-Verlag, 1982, pp. 161–167.

[RAMAMO80] C.V. Ramamoorthy and G.S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440–449, September, 1980.

[RAMC74] C. Ramchandani, *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, Project MAC, TR 120, Massachussets Institute of Technology, 1974.

[RAZO83] R.R. Razouk, "The Derivation of Performance Expressions for Communication Protocols from Timed Petri Net Models," Technical Report 211, University of California, Irvine: Department of Information and Computer Science, October, 1983.

[ROSE84] M.T. Rose, "Modeling and Analysis of Concurrent Systems using Contour/Transition Nets," Ph.D. Dissertation, Technical Report 245, University of California, Irvine: Department of Information and Computer Science, 1984.

[SHIE79] M.W. Shields, "Adequate Path Expressions," *Semantics of Concurrent Computation*, G. Kahn, Ed., Lecture Notes in Computer Science 70, Berlin: Springer-Verlag, 1979.

[SIFA77] J. Sifakis, "Petri Nets for Performance Evaluation," *Measuring, Modelling, and Evaluating Computer Systems*, H. Beilner and E. Gelenbe, Eds., Proceedings of the Third International Symposium of IFIP Working Group 7.3, 1977, pp. 75–93.

[SOUN84] N. Soundararajan, "Axiomatic Semantics of Communicating Sequential Processes," *ACM Transactions on Programming Langauges and Systems*, vol. 6, no. 4, pp. 647–662, October, 1984.

[SUZU83] I. Suzuki and T. Murata, "A Method for Stepwise Refinement and Abstraction of Petri Nets," *Journal of Computer and System Sciences*, vol. 27, no. 1, pp. 51–76, August, 1983.

[SYMO80] F.J.W. Symons, "Representation Analysis and Verification of Communication Protocols," Research Laboratories Report 7380, Telecom Australia, 1980.

[TAYL83] R.N. Taylor, "A General-Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM*, vol. 26, no. 5, pp. 362–376, May, 1983.

[ZUBE80] W.M. Zuberek, "Timed Petri Nets and Preliminary Performance Evaluation," *7th Annual Symposium on Computer Architecture*, 1980, pp. 88–96.