CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

Technical Report 5044

Hierarchical Nets

A Structured Petri Net Approach to Concurrency

by

Young-il Choo

Master's Thesis

2 November 1982

The research in this report was sponsored by the Defense Advanced Research Projects Agency, ARPA Order 3771, and monitored by the Office of Naval Research under Contract N00014-79-C-0597.

Abstract

Liveness and safeness are two key properties Petri nets should have when they are used to model asynchronous systems. The analysis of liveness and safeness for general Petri nets, though shown to be decidable by Mayr [1981], is still computationally expensive (Lipton [1976]). In this paper an hierarchical approach is taken: a class of Petri nets is recursively defined starting with simple, live and safe structures, becoming progressively more complex using net transformations designed to preserve liveness and safeness.

Using simple net transformations, ***nice nets***, which are live and safe, are defined. Their behavior is too restrictive for modeling non-trivial systems, so the ***mutual exclusion*** and the ***repetition*** constructs are added to get ***μ-ρ-nets***. Since the use of mutual exclusions can cause deadlock, and the use of repetitions can cause loss of safeness, restrictions for their use are given. Using μ-ρ-nets as the building blocks, ***hierarchical nets*** are defined. When the mutual exclusion and repetition constructs are allowed between hierarchical nets, ***distributed hierarchical nets*** are obtained. Examples of distributed hierarchical nets used to solve synchronization problems are given.

General net transformations not preserving liveness or safeness, and a notion of ***duality*** are presented, and their effect on Petri net behavior is considered.

Acknowledgments

I am grateful to Jim Kajiya for introducing me to the charms of Petri nets and for the many discussions in which these ideas developed.

Carver Mead first kindled my interest to seek a nicely behaved class of control structures in which concurrent programs can be written that are "correct by construction." His ideal remains tantalizingly beyond our grasp, although this exercise in Petri nets suggests some tentative steps.

Contents

Introduction

Petri Nets and Concurrency

Petri nets are abstract mathematical objects used to model the flow of information and control in concurrent systems. Since they were first defined by C. A. Petri in his 1962 Ph.D. dissertation, much research has been done and a considerable body of knowledge has been accumulated. Peterson's [1981] new text introduces a wide range of theory and application of Petri nets and includes an amazingly comprehensive annotated bibliography.

Petri nets can be used to model complex concurrent systems by suitably interpreting the transitions and places as, for example, events and conditions (Holt and Commoner [1970]). The system's behavior can then be analyzed by simulating the net and observing the flow of tokens.

Two important properties one would like a system to have are liveness and safeness. The analysis of liveness and safeness for general Petri nets, though shown to be decidable by Mayr [1981], who proved the decidability of the equivalent reachability problem, is computationally hard, requiring exponential space (Lipton [1976]).

In order to simplify the analysis, various subclasses of Petri nets have been defined by restricting the types of interconnections. For example, ***state machines*** and ***marked graphs*** are Petri nets whose liveness and safeness properties are easily decidable, but they have been found to be too restrictive. Hack [1972] proposed ***free choice nets***, which include both of the former and can model relatively complex systems, but the analysis of liveness and safeness for this class is already combinatorial because all possible subnets of certain types must be checked.

An Hierarchical Approach

State machines, marked graphs and free choice nets are essentially ***syntactic*** restrictions of general Petri nets. They specify locally allowable configurations of transitions, arcs, and places. Since the behavior of a net can be drastically affected by local change, knowing the behavior of a part of a net does not help in determining the behavior of the whole.

In this paper an ***hierarchical*** approach is explored. Beginning with a class of Petri nets that are easily shown to be live and safe, progressively more complex nets are generated using previously generated nets and net transformations to ultimately get ***distributed hierarchical nets***. By applying net transformations that could cause loss of liveness or safeness — the ***mutual exclusion*** and the ***repetition*** constructs — to one level of the hierarchy at a time, the analysis for liveness and safeness can also be restricted to one level at a time. Necessary and sufficient criteria are given for the safe use of mutual exclusion and repetition constructs. In a similar work, Valette [1979] in his "Analysis of Petri Nets by Stepwise Refinement" generates a class of live and safe Petri nets very similar to ***structured nets*** defined in this paper, but since he does not allow non-liveness and safeness preserving transformations, the kinds of behavior that can be modeled by his nets are severely restricted.

Comparison with Path Expressions

The hierarchical approach to concurrency using Petri nets suggests a work by Lauer and Campbell [1975] on "Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes" where they present a language for generating systems of concurrent processes called ***path expressions***.

Path expressions specify the ordering of events and synchronizations between events, and the semantics is given by providing a simple Petri net for each language construct and a net transformation for each composition construct. They define a hierarchy of paths based on the kinds of interactions between events:

Elementary Paths (***E-paths***): Each letter (event) occurs uniquely and there is choice.

Repeat Paths (*R-paths*): An *E-path* where repetition of letters is allowed.

Disjoint Paths (*D-paths*): A set of *R-paths* where each letter occurs in at most one *R-path*. Therefore, *D-paths* allow concurrency, but no interaction between different paths.

General Paths (*G-paths*): *D-paths* with letters occurring in more than one path. Interaction is by synchronizing letters named the same.

The corresponding Petri nets that are generated are called *E-nets*, *R-nets*, etc. The letters become transitions and synchronization is done by the identification of the corresponding transitions in the net.

A set of paths is used to specify a system of coordinating processes. To determine its behavior, the corresponding Petri net is analyzed. Since paths are sets of looping sequences of events where the same named transitions are synchronized, very general behaviors are producible, but their analysis is difficult. Certain restrictions are conjectured to exist that makes each class of paths live and safe, but it is not clear what these restrictions are. The heart of the problem is that arbitrary identification of transitions can cause deadlock because it is not easy to tell whether two paths are consistent. Concurrency is not explicitly given, but appears as the result of multiple loops.

In distributed hierarchical nets, concurrency and synchronization are introduced either through explicit production rules that preserve liveness and safeness (like the parallel **begin end** constructs), or by the use of special constructs (mutual exclusion and repetition) with suitable restrictions to ensure that the net is well behaved.

As an example of their notation and method of analysis, Lauer and Campbell present the Smokers Problem along with two solutions: a concurrent and a sequential one. The solutions consist of sets of paths, each path representing an agent giving out the different resources or one of the three smokers. To analyze the solutions, the corresponding Petri nets are examined. Each process is pretty simple by itself, but the composite net contains as many loops as the number of processes and is quite complex. So, part of the task in the analysis is to simplify the nets using certain operations that preserve the behavior.

These two solutions are presented in this paper using distributed hierarchical

nets and are shown to be live and safe because they satisfy the restrictions on their use. After simplification, the path expression solutions are very close to the distributed hierarchical nets solutions. The sequential solution uses a global scheduler while the concurrent version utilizes three-way synchronizations.

An Overview

For sake of completeness, an informal introduction to Petri nets using pictures and examples of applications is presented. Liveness and safeness and other notions necessary for this paper are introduced. Those already familiar with the rudiments of Petri nets may safely skip over it.

In Chapter 2, the first level of our hierarchy, nice nets, are defined using simple net transformations, and are shown to be live and safe.

A more homogeneous unit of Petri net structure, based on the notion of a ***process*** that has a beginning, finite computation, and an end, is introduced in Chapter 3, and ***structured nets*** are defined using process transformations that are based on the well known sequential, parallel, choice and loop constructs. Structured nets are shown to be live and safe.

Chapter 4 describes the ***mutual exclusion construct***, which can be thought of as loosely coupled communication between parallel processes, and the ***repetition construct***, which allows parallel processes to activate the same transition. Nets with both mutual exclusion and repetition constructs are called ***μ-ρ-nets***. Since the addition of mutual exclusion can cause deadlock, and repetition construct can make nets not safe, necessary and sufficient conditions for μ-ρ-nets to be live and safe are needed.

By recursively substituting processes with μ-ρ-nets, ***hierarchical nets***, and, by allowing mutual exclusion and repetition between processes of different hierarchical nets, ***distributed hierarchical nets*** are obtained. Applications of distributed hierarchical nets in the solution of the smokers and the dining philosophers problem are given.

In Chapter 5, general net transformations not preserving liveness or safeness and a notion of ***duality*** are presented, and their effect on Petri net behavior is

considered. Certain symmetries that arise from the general net transformations are considered, and their effect on Petri net behavior is explored.

Thoughts on the use of Petri nets in the design and verification of concurrent programs, and consideration of how hierarchical nets relate to concurrent computation in general, are contained in the concluding Chapter 6.

Chapter 1

Petri Nets

This is an informal introduction to Petri nets designed to acquaint the reader with the necessary concepts in an intuitive a way as possible. Examples are given of different types of applications.

Intuitive Petri Nets

Petri nets are usually represented as graphs with two kinds of nodes (see Figure 1–1). One kind is drawn as small circles while the other as short lines. The circles are called *places* (p_1, p_2, p_3 and p_4) and the lines are called *transitions* (t_1, t_2 and t_3). Directed arcs — drawn as arrows — link one kind of node with another.

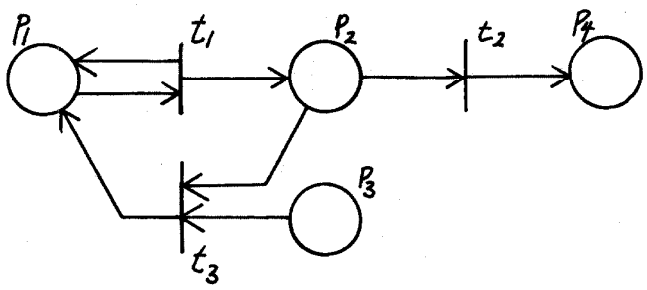


Figure 1–1. A Petri Net.

The graph represents the static structure of a Petri net. The dynamic behavior of the Petri net comes about by moving *tokens* — represented as small dots — around on the Petri net graph (see Figure 1–2) according to rules that will be described below. Tokens reside only in the places (p_2, p_3). A Petri net with tokens

is said to be *marked*; and the particular distribution of tokens among the places is known as a *marking*.

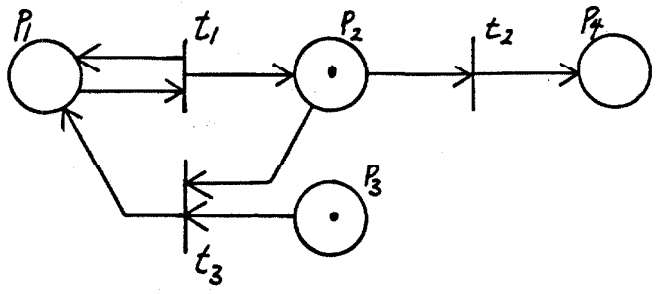


Figure 1–2. A Marked Petri Net.

A transition may have places that have arcs leading to it; these places will be called the *input places* of the transition. Similarly, the places that have arcs coming from the transition will be called the *output places* of that transition. For example, in Figure 1–2, the input places of transition t_3 are p_2 and p_3, and the output place is p_1.

When each of the input places of a transition has at least one token in it, the transition is *enabled*. Once a transition is enabled, it can *fire* by removing a token from each of its input places and adding a token into each of its output places. Firing occurs instantaneously.

In general, if a place has more than one arc to a transition, the number of tokens in the place must equal the number of arcs in order to enable the transition. When the transition fires, as many tokens as there are arcs are removed from each input place and as many tokens as there are arcs are added to each output place.

In Figure 1–2, transition t_3 is enabled; so t_3 can fire by removing a token from each of the input places p_2 and p_3, and adding one token to p_1, the output place. The result is Figure 1–3, where now t_1 is enabled. When t_1 fires, the result is Figure 1–4. Firings can continue until no transition is enabled. In this example, t_1 can fire forever, adding tokens into p_1 and p_2, which allows t_2 to fire, gradually filling p_4 with many tokens. Once t_3 has fired and used up the token in p_3, it can never fire again.

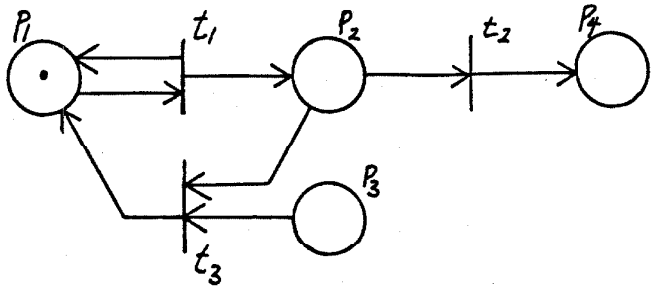


Figure 1–3. Figure 1–2 after t_3 fires.

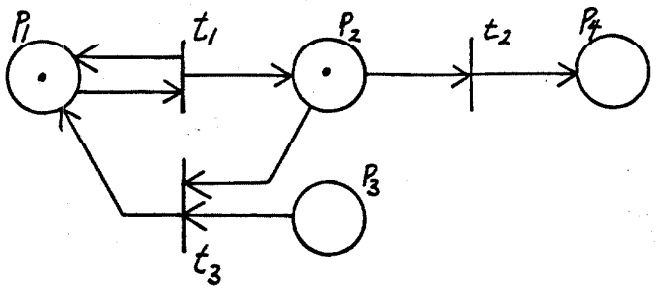


Figure 1–4. Figure 1–3 after t_1 fires.

Interpretation and Modeling

Petri nets were defined to model the flow of information in a system. The tokens moving through a net can be thought of as, for example, the flow of information that merges and diverges as they enable and fire transitions in an asynchronous manner. Another general way of looking at Petri nets is to think of the places as conditions that are satisfied if they have tokens, and the firing of transitions as events occurring (Holt and Commoner [1970]). For computer programmers the most natural way might be to think of the tokens as representing the flow of control

through a program. The presence of more than one token means that the program is running in parallel.

Examples.

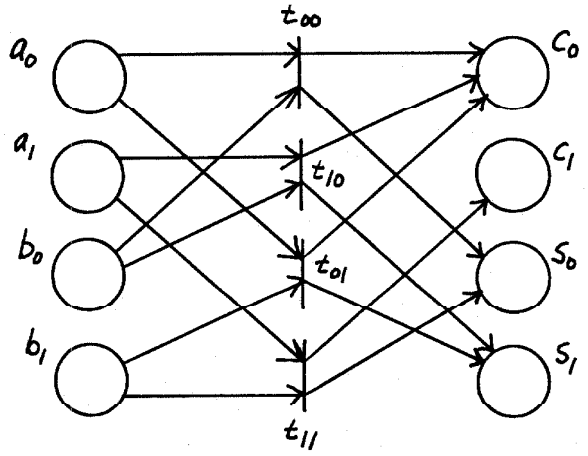


Figure 1–5. A one bit adder.

Figure 1–5 is a model of a one bit adder with carry. Each of the two inputs has two places (a_0, a_1 and b_0, b_1) which represent the value zero or one, depending on which place has a token. One of the transitions t_{00}, t_{01}, t_{10}, or t_{11} will fire, depending on which places have tokens, and place tokens into the appropriate sum and carry places. If this Petri net were to be a part of a larger system, then new structures can be added to ensure that tokens do not appear in both the zero and one places at the same time, and also that new inputs do not arrive until the outputs have been defined.

Figure 1–6 represents in Petri net form Dijkstra's [1968] classic dining philosophers problem requiring mutual exclusion. The place e_i represents the proposition "philosopher i is eating," while the place p_i means "philosopher i is thinking." Between each philosopher is a chopstick. Clearly, one needs two chopsticks in order to eat, so each philosopher must obtain two, one from his right and one from his left. However, two neighboring philosophers cannot be allowed to grab the same chopstick at the same time, so mutual exclusion is necessary. Presence of a token in place c_i means that the chopstick is available.

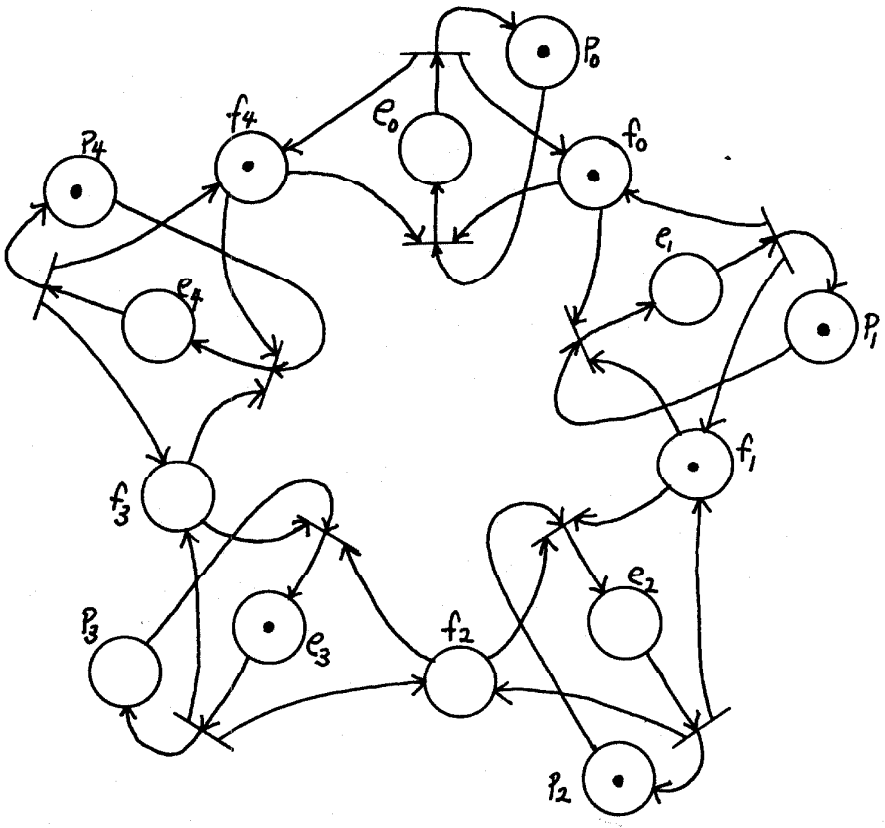


Figure 1–6. Dining philosophers.

In this solution to the problem, a philosopher cannot grab only one chopstick, it's either both or none, thereby avoiding any deadlock. Figure 1–6 shows philosopher *3* eating while the rest are thinking.

Liveness and Safeness

Once a Petri net is given that models a system, there are couple of general properties that one would like to know. The first is whether from an initial marking every transition can be fired infinitely often. This property is known as ***liveness.*** The second is whether every place will have at most one token at all times. This property is called ***safeness.*** Figures 1–7 and 1–8 show examples of small live and

safe Petri nets.

Formally, let A be a Petri net with P the set of places and T the set of transitions. Let M be the set of markings of A where a marking ν is a function from P to the non-negative integers. Marking ν_2 is ***immediately reachable*** from marking ν_1 if there is a transition t, enabled in ν_1, such that firing it produces the new marking ν_2. A marking is ***reachable*** from another if there is a sequence of zero or more transitions and markings where each is immediately reachable from the previous one. Note that a marking may be reachable by different sequences of transitions. Let $R(A, \nu_0)$ denote the set of all markings reachable from ν_0 in the Petri net A.

The Petri net A with initial marking ν_0 is ***live*** if for every transition t and every marking ν_1 in $R(A, \nu_0)$, there exists another marking ν_2 in $R(A, \nu_1)$ (reachable from ν_1) such that t is enabled in ν_2. The Petri net A with initial marking ν_0 is ***safe*** if for every marking ν in $R(A, \nu_0)$ the value of ν never exceeds one.

Liveness is concerned about whether the desirable goals can be reached, while safeness ensures that no contradictory states are reached. For example, if an operating system was modeled using Petri nets, liveness would guarantee that the operating system will continue doing its job, while safeness would say, for example, that it would not be assigning the same printer to two people at the same time.

Subclasses of Petri Nets

For general Petri nets, determining whether the net is live and safe is not easy because of the asynchronous movement of the tokens in the net. In order to simplify the analysis, various subclasses of Petri nets have been defined that impose restrictions in the way the places and transitions are connected.

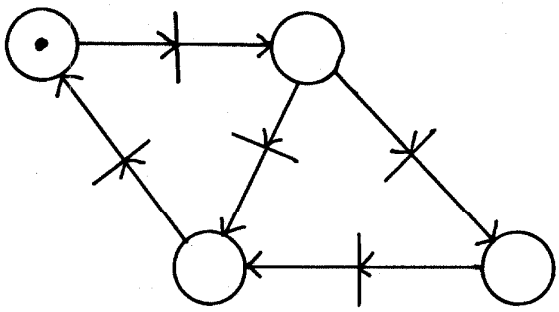


Figure 1–7. A State Machine.

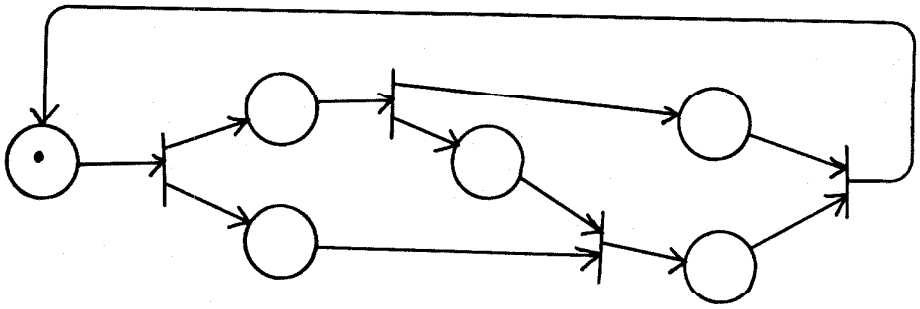


Figure 1–8. A Marked Graph.

For example, if each transition has at most one incoming and one outgoing arc, then the subclass is called ***state machines***. State machines model choices but not concurrency, and the complete characterizations for liveness and safeness are known: Since no transition has more than one outgoing arc, the number of tokens cannot increase, it is safe; if every place is reachable from every other and there is at least one token in the net, it is live. Figure 1–7 is an example of a live and safe state machine.

As a dual to state machines, if each place has at most one incoming and one outgoing arc, then the subclass is called ***marked graphs***. In a way that is dual to state machines, marked graphs exhibit concurrent behavior but are totally deterministic.

Liveness and safeness have also been completely characterized for marked graphs. If every transition is in a loop, and every loop has one token on it, then it is live and safe. Figure 1–8 shows a live and safe marked graph.

Decision Problems

Marked graphs and state machines have high decision power, but they are quite restrictive in their modeling ability. Increasing the modeling power results in harder decision procedures. Hack [1972] has defined a subclass of Petri nets ***free-choice nets*** that contains both the marked graphs and the state machines. A Petri net is a free-choice net if every arc is either a unique outgoing arc of a place or a unique incoming arc into a transition (Figure 1–9). In this way there is choice and concurrency, but once a transition has been enabled it can not be disabled by the firing of another. The necessary and sufficient conditions for liveness and safeness of free choice nets are given in Hack's paper.

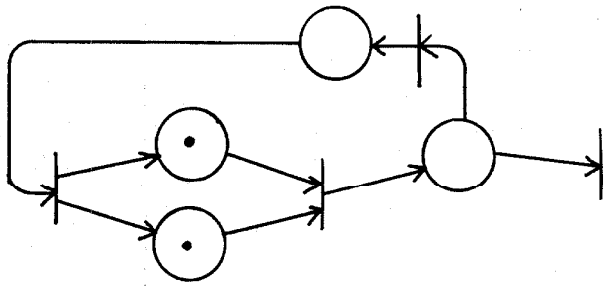


Figure 1–9. A Free-Choice Net.

Chapter 2

Nice Nets

To manage the design of complex entities, structured programming tells us to begin with simple building blocks and gradually build up using a few well defined composition rules. Nice nets are obtained from a trivially live and safe Petri net by applying rules corresponding to the sequential, parallel, choice, and loop constructs.

Liveness and Safeness Preserving Transformations

Consider the kinds of transformations that can be done to a Petri net so that the properties of liveness and safeness are preserved.

A place could be transformed into two places linked by a new transition (Figure 2–1). Likewise a transition can be made two with a place in between (Figure 2–2). These two transformations correspond to the transformation of one statement into two statements (S becomes $S_1; S_2$) in regular programming.

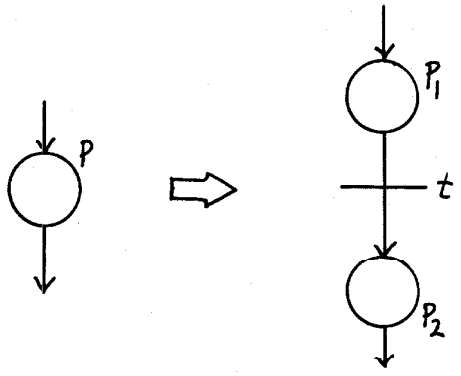


Figure 2–1. Place Refinement.

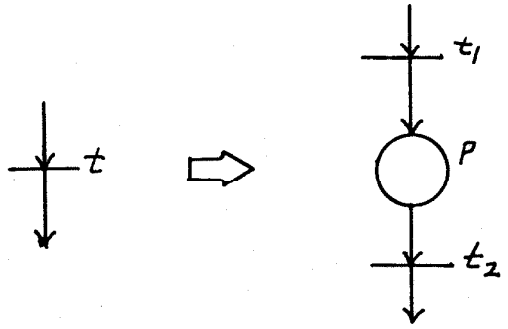


Figure 2–2. Transition Refinement.

Another transformation would be to split the place between two transitions so that they are connected in parallel to the original transitions (Figure 2–3). This creates extra tokens, but it is done in a very structured manner so that safeness is not lost. This models the **parbegin-parend** of some programming languages.

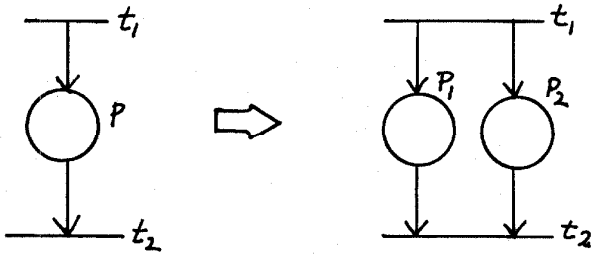


Figure 2–3. Place Splitting.

If a transition between its input and output places is split, then a structure with non-deterministic choice results where the token can fire one or the other transition (Figure 2–4). This corresponds to the **if-then-else** construct of programming languages. A cyclic loop can be put where there is a place to produce the equivalent of a **while-loop** (Figure 2–5). These transformations will be shown to preserve liveness and safeness.

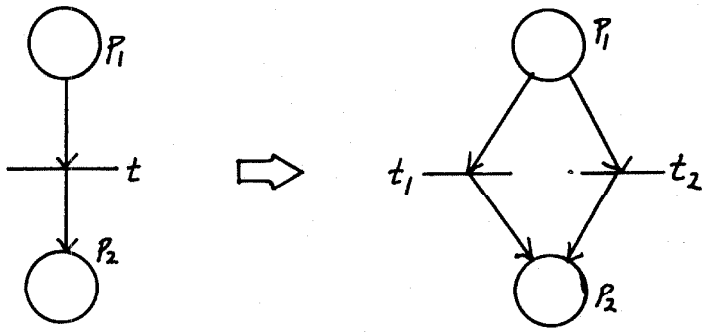


Figure 2–4. Transition Splitting.

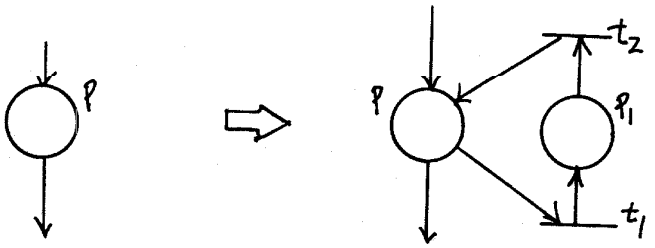


Figure 2–5. Place Loop.

Net Transformations

Notation. $\rightarrow$ will be used to represent the arcs connecting the places and transitions. So, $p \rightarrow t$ will mean that p is an input place of t and t is an output transition of p. If P denotes a set of places, then $P \rightarrow t$ will mean that every member of P is an input place of t. $\bullet t$ will denote the set of input places of t, and $t\bullet$ set of output places of t. Likewise, $\bullet p$ and $p\bullet$ will denote the sets of input and output transitions of p. A Petri net can be represented completely as a set of arcs between

transitions and places. In general, there may be multiple arcs between a place and a transition, in which case a multiset (or a bag) of arcs is used. A multiset is like a set but it allows multiple occurrences of an element.

$A \rightarrow x \rightarrow B$ will mean that $A = {\bullet}x$ and $B = x{\bullet}$, for x either a place or a transition. If X is a set of places (or transitions), then $A \rightarrow X \rightarrow B$ would mean that A is the set of input transitions (or places) for each element of X, and likewise for B. $\{a\} \rightarrow t \rightarrow \{b\}$ will be written $a \rightarrow t \rightarrow b$.

Once the structure of a Petri net is given, the marking can be defined as a function $\nu : P \mapsto \mathcal{N}$, where P is the set of places and $\mathcal{N} = \{0, 1, 2, \ldots\}$, the set of natural numbers. Only the places with non-zero markings need to be mentioned.

Example. The Petri net of Figure 1–2 can be represented as:

$$\{p_1 \rightarrow t_1 \rightarrow \{p_1, p_2\}, p_2 \rightarrow t_2 \rightarrow p_4, \{p_2, p_3\} \rightarrow t_3 \rightarrow p_1\} \text{ and } \nu(p_2) = 1, \nu(p_3) = 1.$$

In the definition below, new places and transitions will be created. Every new name will be distinct from all other names already occurring in the net. $\setminus$ will denote set difference, and $\Rightarrow$ will mean: replace the node or subnet on the left with the new structure on the right.

Definition. A *net transformation* is one of the following:

Transition Refinement (**TR**): $t \Rightarrow t_1 \rightarrow p \rightarrow t_2$ and make ${\bullet}t_1 = {\bullet}t$ and $t_2{\bullet} = t{\bullet}$.

Place Refinement (**PR**): $p \Rightarrow p_1 \rightarrow t \rightarrow p_2$ and make ${\bullet}p_1 = {\bullet}p$ and $p_2{\bullet} = p{\bullet}$.

Transition Splitting (**TS**): $p_1 \rightarrow t \rightarrow p_2 \Rightarrow p_1 \rightarrow \{t_1, t_2, \ldots, t_n\} \rightarrow p_2$ where $n \geq 1$.

Place Splitting (**PS**): $t_1 \rightarrow p \rightarrow t_2 \Rightarrow t_1 \rightarrow \{p_1, p_2, \ldots, p_n\} \rightarrow t_2$ where $n \geq 1$.

Place Loop (**PL**): $p \Rightarrow \{p_1 \rightarrow t_1 \rightarrow p_2, p_2 \rightarrow t_2 \rightarrow p_1\}$ and make ${\bullet}p_1 \cup {\bullet}p_2 \setminus \{t_1, t_2\} = {\bullet}p$ and $p_1{\bullet} \cup p_2{\bullet} \setminus \{t_1, t_2\} = p{\bullet}$.

Definition. The Petri net $\{p_1 \rightarrow t_1 \rightarrow p_2, p_2 \rightarrow t_2 \rightarrow p_1\}$ with a token in p_1 (Figure 2–6) is clearly live and safe. Call this a *live loop*.

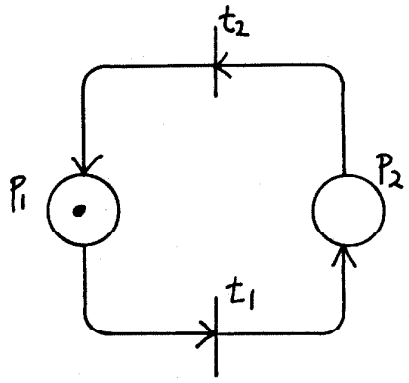


Figure 2–6. A live loop.

Definition. Petri nets obtained from a live loop by finite number of applications of the net transformations **TR**, **PR**, **TS**, **PS**, and **PL** will be called *nice nets*.

Proposition. Nice nets are subclasses of free-choice nets.

Sketch of proof: It is impossible to produce a nice net with an arc that is both a multiple output of a place and a multiple input to a transition. □

Liveness and Safeness of Nice Nets

Liveness is defined for transitions and safeness for places, but to facilitate the proofs below we extend these notions.

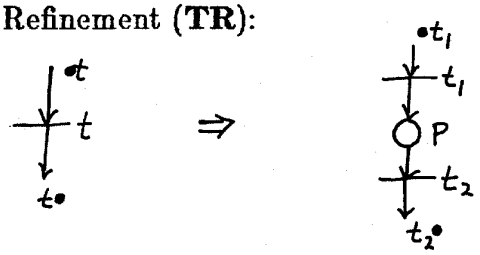
Definition. A place will be called *live* if every input transition to it is live. A transition will be called *safe* if every input place to it is safe and the set of input places cannot be marked when any of the output places are marked.

Theorem. The net transformations **TR**, **PR**, **TS**, **PS**, and **PL** preserve liveness and safeness.

Proof: The idea is that a live and safe Petri net remains live and safe after an

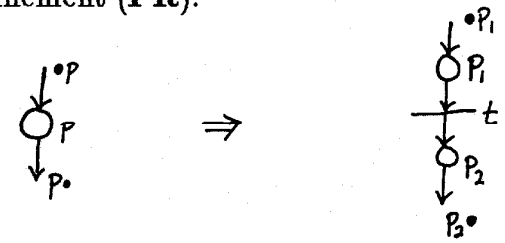
application of a net transformation on a place or a transition.

(a) Transition Refinement (**TR**):



If transition t is live, then since $\bullet t = \bullet t_1$, t_1 is live and this makes t_2 live too. Since the net is assumed safe, the two sets of places, $\bullet t$ and $t\bullet$, cannot be marked simultaneously. Which means that t cannot be enabled until $t\bullet$ is empty of tokens. So, in the transformed net, after t_1 fires and puts a token in p, t_1 cannot be enabled again until after t_2 has fired and $t_2\bullet = t\bullet$ is empty. This shows that p is safe.

(b) Place Refinement (**PR**):

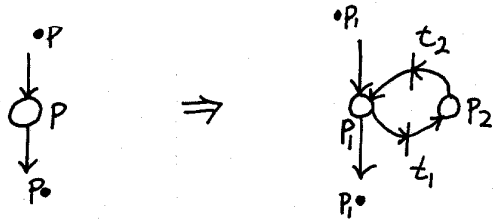


Since p is live and safe, this implies that p_1 is, and since $\{ p_1 \} = \bullet t$, t is live. t live implies that p_2 is live. To show that p_2 is safe, observe that p safe means that $\bullet p$ will not fire until $p\bullet = p_2\bullet$ have fired, so p_1 will not receive a token until $p_2\bullet$ have fired.

(c) Transition Splitting (**TS**): If t is live, then clearly $t_1, t_2, \ldots, t_n$ are all live. Safeness is not affected since no extra tokens can be generated.

(d) Place Splitting (**PS**): If p is safe, then $p_1, p_2, \ldots, p_n$ will also be safe, since t_1 will not fire while there is a token in p. Trivially, if p is live, then $p_1, p_2, \ldots, p_n$ are also live, maintaining the liveness of t_2.

(e) Place Loop (**PL**):



Since p is live and safe, only one of the transitions in $\bullet p$ will fire until the token in p has been used up by firing another transition in $p\bullet$. Since p_1, t_1, p_2, t_2 are strongly connected, they are all live. Since no new tokens will enter the loop until one has been used up in firing a transition in $p\bullet$, it is still safe. □

Chapter 3

Structured Nets

In the usual interpretation of Petri nets, the places and transitions are considered individually. The net transformations defined in the previous section also took individual places and transitions and replaced them with small Petri net structures. In this chapter a more homogeneous unit, called a ***process***, that consists of a transition, place, and another transition will be introduced. The motivation behind this structure is that of a program block which has a **begin**, finite computation, and an **end**.

Using the notion of a process, ***process transformations*** are defined. These correspond to the structured programming constructs with parallelism added. The class of Petri nets obtainable from a process under finite number of application of process transformations will be called ***structured nets***. Liveness and safeness of structured nets is proved.

Processes

Definition. A ***process*** is a Petri net with the structure $t_i \rightarrow p \rightarrow t_o$. t_i is called the ***input*** and t_o is called the ***output***, and p is the place of the process. By convention on the notation introduced earlier, p has unique input and output transitions. A process is ***marked*** or ***busy*** if its place is marked. When the input transition fires, the process ***starts*** or is ***activated***, and when the output transition fires, it ***terminates***.

Notation. Let a, b, c, ... denote processes. If $a \equiv t_1 \rightarrow p \rightarrow t_2$, then $a^- = t_1$, $a_- = t_2$, and $a\circ = p$, will be the three selector functions that pick out the input, output and place of the process a, respectively. $\nu(a) = 1$ will mean $\nu(a\circ) = 1$.

Process Transformations

Process transformations will be defined to take a process and produce two or more sequential processes, two or more parallel processes, a choice of two or more processes or a multiple loop of processes.

The identification function, $\sim$, which takes two transitions or two places and identifies them and makes them one, will be used to paste together different Petri net structures. All the inputs and outputs of the arguments will be inherited by the new transition or place.

Definition. Given a set of processes $b_1, b_2, \ldots, b_n$, a ***composite structure*** is one of following:

Sequence: $\sigma(b_1, b_2, \ldots, b_n)$ is a Petri net structure with

$$b_{1-} \sim b_2^-, b_{2-} \sim b_3^-, \ldots, b_{n-1-} \sim b_{n-}, \text{ and}$$

$$\sigma(b_1, b_2, \ldots, b_n)^- = b_1^-, \sigma(b_1, b_2, \ldots, b_n)_- = b_{n-}.$$

Parallel: $\pi(b_1, b_2, \ldots, b_n)$ is a Petri net structure with

$$\pi(b_1, b_2, \ldots, b_n)^- = b_1^- \sim \cdots \sim b_n^-, \text{ and}$$

$$\pi(b_1, b_2, \ldots, b_n)_- = b_{1-} \sim \cdots \sim b_{n-}.$$

Choice: $\phi(b_1, b_2, \ldots, b_n)$ is a Petri net structure with two new places p_1 and p_2, and two new transitions $\phi(b_1, b_2, \ldots, b_n)^-$ and $\phi(b_1, b_2, \ldots, b_n)_-$, such that

$$\phi(b_1, b_2, \ldots, b_n)^- \rightarrow p_1 \rightarrow \{ b_1^-, \ldots, b_n^- \}, \text{ and}$$

$$\{ b_{1-}, \ldots, b_{n-} \} \rightarrow p_2 \rightarrow \phi(b_1, b_2, \ldots, b_n)_-.$$

Loop: $\lambda(b_1, b_2, \ldots, b_n)$ is a Petri net structure with one new place p and two new transitions $\lambda(b_1, b_2, \ldots, b_n)^-$ and $\lambda(b_1, b_2, \ldots, b_n)_-$, such that

$$\{ \lambda(b_1, b_2, \ldots, b_n)^-, b_{1-}, \ldots, b_{n-} \} \rightarrow p, \text{ and}$$

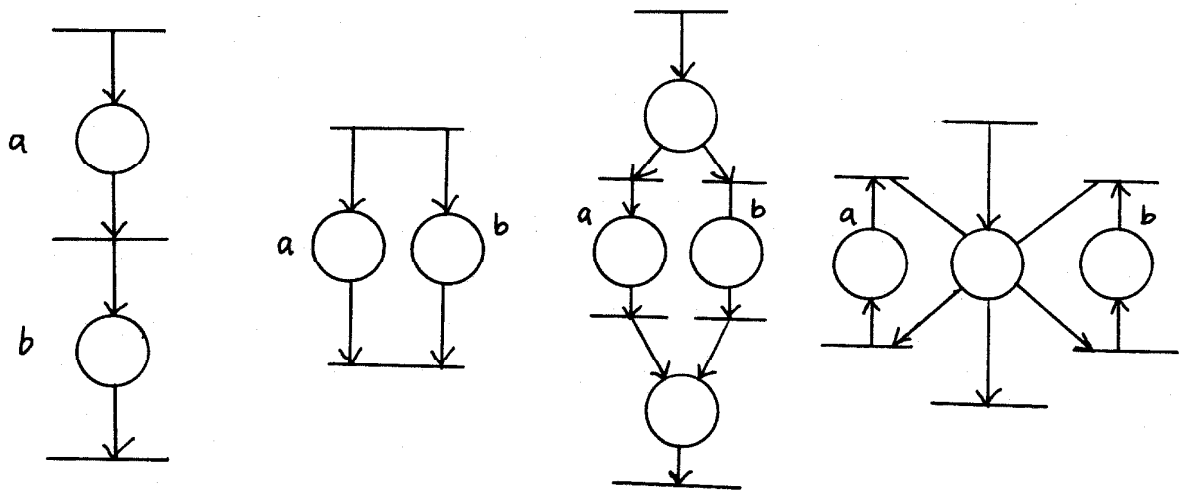
$$p \rightarrow \{ \lambda(b_1, b_2, \ldots, b_n)_-, b_1^-, \ldots, b_n^- \}.$$

Definition. If a and $b_1, b_2, \ldots, b_n$ are processes, a *process transformation* replaces a with $\chi(b_1, b_2, \ldots, b_n)$ where χ is one of σ, π, ϕ or λ, such that

$$a^- \sim \chi(b_1, b_2, \ldots, b_n)^-, \text{ and } a_- \sim \chi(b_1, b_2, \ldots, b_n)_-$$

and $a\omega$ is removed.

Notation. $a \Rightarrow \chi(b_1, b_2, \ldots, b_n)$ will denote process transformation χ.



$\sigma(a,b)$ $\pi(a,b)$ $\phi(a,b)$ $\lambda(a,b)$

Figure 3–1. Composite Structures of two Processes

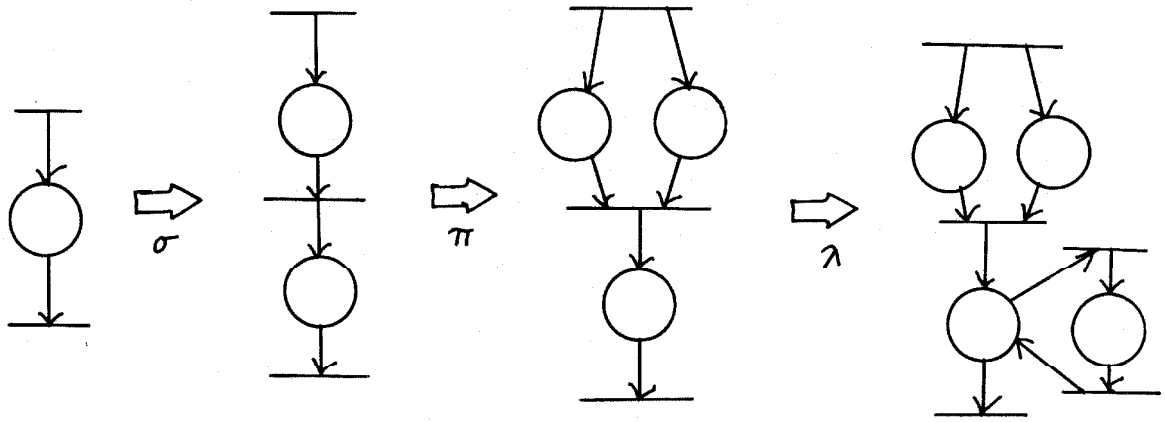


Figure 3–2. Examples of Process Transformations.

Lemma. Given a live and safe Petri net containing processes, process transformations σ, π, ϕ, and λ preserve liveness and safeness.

Sketch of proof: The proof is by showing that each process transformation is a composite of net transformations introduced before. σ is obtained by applying **PR** to $a\circ$ or **TR** to a transition and then **PR** or **TR** to the new places or transitions created. π is obtained by applying **PS** to $a\circ$. For ϕ, first apply **PR** to $a\circ$, then apply **TS** to the newly created transition, and finally apply **TR** to the two new transitions. λ is obtained by applying **PL** to $a\circ$ n times. So, since these transformations are compositions of net transformations, they automatically preserve liveness and safeness. □

Liveness and Safeness of Structured Nets

Nice nets were defined as the class of nets obtainable from the live loop under finitely many applications of net transformations. Similarly,

Definition. A *structured process* is a Petri net obtained from a process by finitely many applications of process transformations.

A structured process with unique input and output is made into a structured net by augmenting it with a single marked place.

Notation. Let $x, y, z, \ldots, x_1, x_2, \ldots$ denote structured processes. x^- and x_- will denote, respectively, the input and output of the structured process x.

Definition. A structured process x with an extra place p with $p \rightarrow x^-$, $x_- \rightarrow p$, and $\nu(p) = 1$, will be called *augmented*, and will be denoted $\odot x$. An augmented structured process will be called a *structured net*. Structured nets will also be denoted by $x, x_1, \ldots$. For a given structured process, its augmented counterpart will be called its *associated net*. Note: Since a live loop consists of two processes joined in a loop, a structured net is the equivalent of applying the process transformations to the unmarked half of a live loop.

Definition. A structured process will be called *live* and *safe* if and only if its associated net is live and safe. The notions of liveness and safeness of nets is defined with respect to the initial marking. For structured nets the *initial marking* is defined to be when only the augmented place has one token.

Theorem. Structured nets are live and safe.

Proof: Process transformations preserve liveness and safeness. A structured net can be obtained from a live loop by applying process transformations to one half of the loop. Therefore structured nets are live and safe. □

The process transformations gave us a top down approach to structured nets. We can equally have a bottom up recursive definition.

Definition. (Recursive definition for structured nets)

(a) A process is a structured process.

(b) If $x_1, x_2, \ldots, x_n$ are structured processes, then so are $\sigma(x_1, x_2, \ldots, x_n)$, $\pi(x_1, x_2, \ldots, x_n)$, $\phi(x_1, x_2, \ldots, x_n)$, and $\lambda(x_1, x_2, \ldots, x_n)$, where σ, π, ϕ and λ are the composite structures defined above, but instead of processes, structured processes occur as parameters. Note that the arguments may be permuted in all but σ.

This definition allows us to denote a structured process as a functional composition of the four operators σ, π, ϕ and λ.

Notation. We write $\chi(x_1, x_2, \ldots, x_n)$, for $n \geq 1$, to denote the fact that $x_1, x_2, \ldots, x_n$ are components of a χ-construct, where χ is one of σ, π, ϕ and λ.

Example.

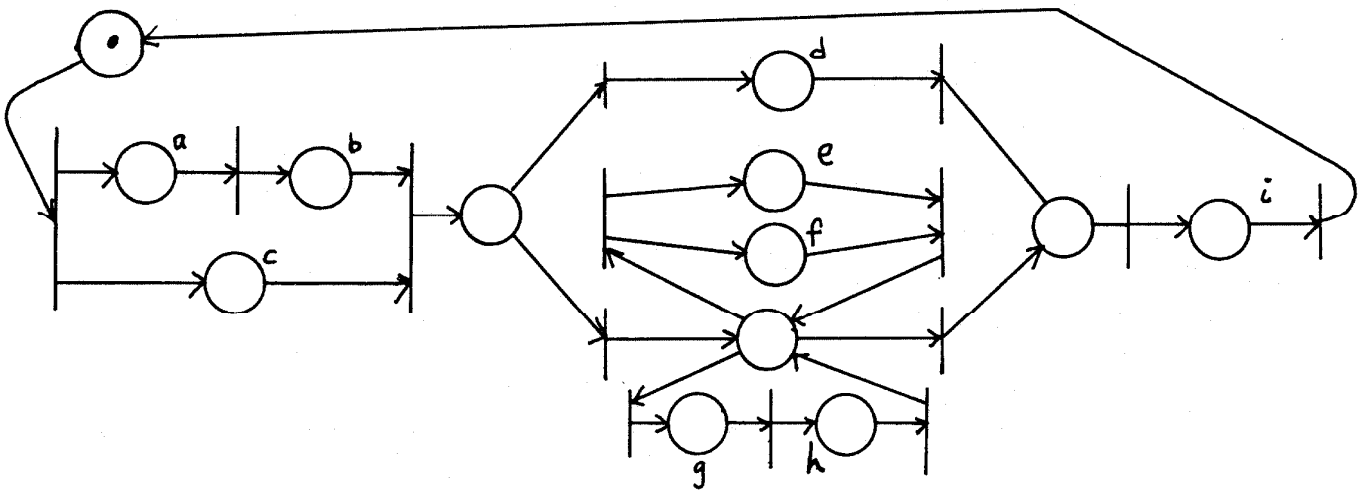


Figure 3–3. $\odot\sigma(\pi(\sigma(a,b),c),\phi(d,\lambda(\pi(e,f),\sigma(g,h))),i)$

Definition. A *net substitution*, written $[x/a]y$, is the operation of replacing the process a with the structured process x in the structured process or net y. This means that the input of x is identified with the input of a and the output of x is identified with the output of a, and ao is removed. For example,

$$[\sigma(a,\pi(b,c),d)/e]\ \pi(g,\phi(e,f)) = \pi(g,\phi(\sigma(a,\pi(b,c),d),f)).$$

Concurrency Relation

Definition. Let a and b be structured processes. a is a *sub-component* of b, written $a \subseteq b$, if either $a \equiv b$ or $a \subseteq c$ and $b = \chi(\ldots,c,\ldots)$ for some structured process c and structure χ.

Definition. Two processes a and b in a structured net are *concurrent* or in a *concurrency relation*, written $a \parallel b$, if there is a marking reachable from the initial marking in which both of the processes are marked. The concurrency relation is symmetric and reflexive. The complement of the concurrency relation is the *order relation*.

Definition. Two structured processes are *concurrent* or in a *concurrency relation* if every process of one is concurrent with every process of the other. The same symbol ($\|$) will be used as for processes. A set of processes is called *concurrent* if every process in it is concurrent with every other. Similarly for concurrent sets of structured processes.

Lemma. Let a, b be two processes in a structured net. $a \parallel b$ if and only if there exist c and d such that $a \subseteq c$ and $b \subseteq d$ and $\pi(c, d)$.

Proof: ($\Leftarrow$) Assume $\pi(c, d)$ and $a \subseteq c$ and $b \subseteq d$ for some c and d. By definition of π the two structured processes c and d can be simultaneously marked. Since structured processes are live and safe, a and b can be marked simultaneously after $\pi(c, d)^-$ fires. ($\Rightarrow$) Assume $a \parallel b$. This means that the two processes a and b can be marked simultaneously. The only process transformation that creates multiple tokens is π. Assume that a and b do not have an enclosing π construct. Then, since they are part of a strucutred net, there is a smallest construct that encloses them which is not π. Since all the other constructs do not produce multiple tokens, a and b cannot be concurrent. So, a and b are sub-components of some c and d, respectively, with $\pi(c, d)$. □

These notions of concurrency and concurrent sets for structured processes will be needed in subsequent chapters.

Chapter 4

Hierarchical Nets

The class of behaviors that structured nets can model is quite restrictive. They allow parallelism but not interaction between the parallel processes. In this chapter two forms of interaction between parallel processes, the ***mutual exclusion*** and the ***repetition***, are introduced.

Mutual exclusion is usually thought of as the sharing of a common resource with the constraint that the two processes cannot both be using the resource at the same time. Since messages can be left in the common resource, mutual exclusion can be thought of as a loosely coupled form of communication. Mutual exclusion resembles a message box; synchronization resembles a telephone conversation.

Repetition allows different processes to cause a common event. When using Petri nets to represent program behavior, the common transition can be thought of as the same communication channel that is being used at different points in a program.

Hierarchical nets are obtained when both the mutual exclusion and repetition are allowed between processes of a structured net. Since the use of mutual exclusions and repetitions can cause deadlock and unsafeness, the main result of this chapter are the necessary and sufficient conditions for their use to preserve liveness and safeness.

Mutual Exclusion Construct

The mutual exclusion construct is a Petri net structure that forces the places of a number of processes to be mutually exclusive about when they can be marked. If a marked place is interpreted as meaning the use of a resource then only one of the processes can read the resource at any one time.

Definition. The ***mutual exclusion construct*** between n processes $a_1, a_2, \ldots, a_n$,

written $\mu(a_1, a_2, \ldots, a_n)$, is a new place p with one token and structure

$$\{a_{1-}, a_{2-}, \ldots, a_{n-}\} \rightarrow p \rightarrow \{\bar{a_1}, \bar{a_2}, \ldots, \bar{a_n}\}.$$

The a_i's are the ***components*** of the mutual exclusion $\mu(a_1, a_2, \ldots, a_n)$. Note that the order of the processes is immaterial. When one of the component processes becomes marked and the place $p = \mu o$ becomes unmarked, the mutual exclusion is ***busy***, otherwise, it is ***free*** or ***available***.

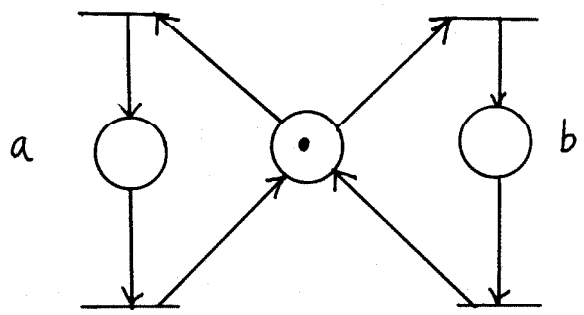


Figure 4–1. A mutual exclusion construct $\mu(a, b)$.

Lemma. An n-ary mutual exclusion construct can be modeled using $\binom{n}{2}$ binary mutual exclusion constructs.

Proof: $\mu(a_1, a_2, \ldots, a_n)$ means that whenever one of the processes is busy, none of the others are. If the processes are nodes and mutual exclusion between them is represented as arcs, then the n-ary mutual exclusion can be represented as a complete graph with n nodes and $\binom{n}{2}$ arcs (see Figure 4–2). So $\binom{n}{2}$ binary mutual exclusions are needed. □

This Lemma shows we can restrict our discussion to binary mutual exclusions without loss of generality.

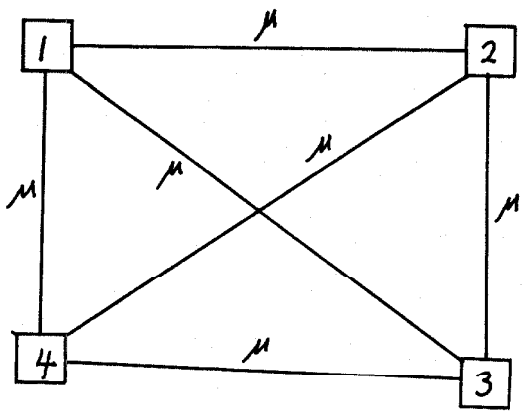
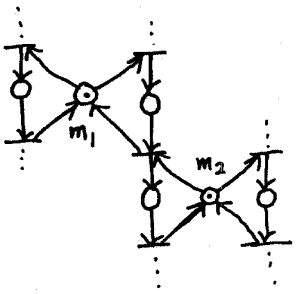


Figure 4–2. Simulation of an n-ary μ using $\binom{n}{2}$ binary μ's (for $n = 4$)

Notation. Let $m, m_1, m_2, \ldots$ denote mutual exclusions. m^- and m_- , called the input and output of the mutual exclusion, will denote the set of inputs and outputs of the component processes, respectively. The new place introduced in a mutual exclusion will be denoted by $m\circ$.

Definition. Let $m_1 \equiv \mu(a_1, a_2)$ and $m_2 \equiv \mu(b_1, b_2)$. m_1 ***directly precedes*** m_2, written $m_1 \vdash\!\!\!- m_2$, if $a_{i-} = b_j^-$ for some i or j. Note that this allows the possibility that each precedes the other (for example, if $a_{1-} = b_1^-$ and $b_{2-} = a_2^-$).



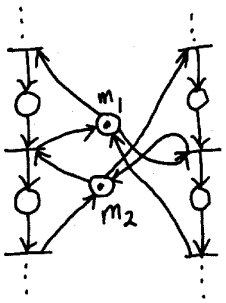


Figure 4–3. $m_1 \vdash\!\!\!-^* m_2$. $m_1 \vdash\!\!\!-^* m_2$ and $m_2 \vdash\!\!\!-^* m_1$.

Using the directly precedes relationship, a pre-order ***precedes*** is defined, written $\vdash\!\!\!-^*$, to be the transitive closure of directly precedes: If $m_1 \vdash\!\!\!- m_2$ and $m_2 \vdash\!\!\!-^* m_3$ then, $m_1 \vdash\!\!\!-^* m_3$.

Deadlock

Deadlock is a state where a set of processes are all blocked because they are all waiting for some resource which the others have and will not give up. Since μ-structured nets allow communication between parallel processes, deadlock is possible.

Definition. A set of mutual exclusions is said to be ***deadlocked*** or is in a ***state of deadlock*** if they are all busy and no transition in any of the outputs of the mutual exclusions is firable. A μ-structured net has ***danger of deadlock*** if it has a set of mutual exclusions with a reachable deadlocked marking.

Arbitrary application of the mutual exclusion construct to a structured net can easily cause deadlock. Some deadlocks can occur because of the way the mutual exclusion was applied, others can occur due to the interaction of several mutual exclusions. In this section certain restrictions necessary to prevent obvious deadlocks are considered.

Definition. A mutual exclusion construct is ***pathological*** if applying it creates danger of deadlock. Mutual exclusions that are not pathological are called ***normal.***

Lemma. Let a and b be processes of a structured net. The mutual exclusion $\mu(a, b)$ is pathological if one of the following conditions hold:

(a) $a^- = b^-$

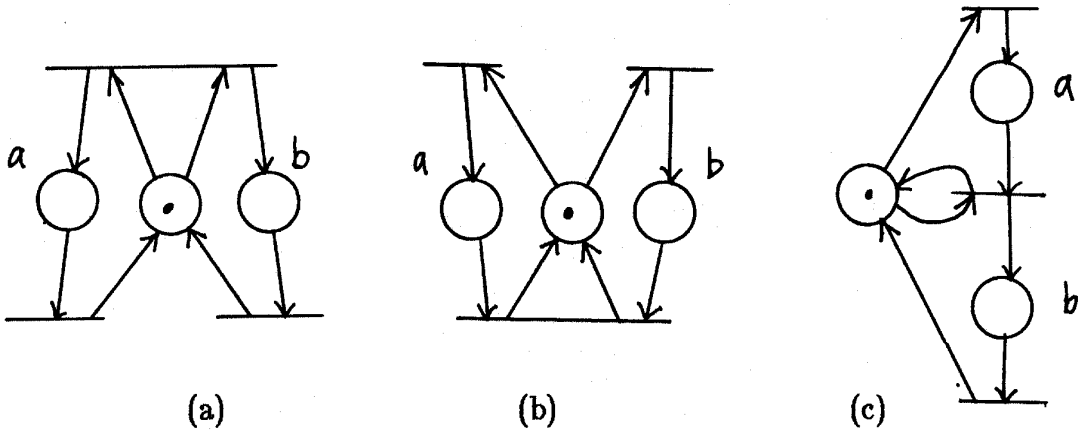
(b) $a_- = b_-$

(c) $a_- = b^-$

(d) $\pi(a, c)$ and $b \subset c$.

Proof:

For (a) and (b) (see Figure 4–4 (a) and (b)): If $a^- = b^-$, then the mutual exclusion cannot be entered because there are two arcs from μ_o to a^- but only one token in μ_o. If $a_- = b_-$, then the mutual exclusion cannot be left because to enable the output of the mutual exclusion the places in the two processes must be marked, but by the mutual exclusion construct only one of them is marked.

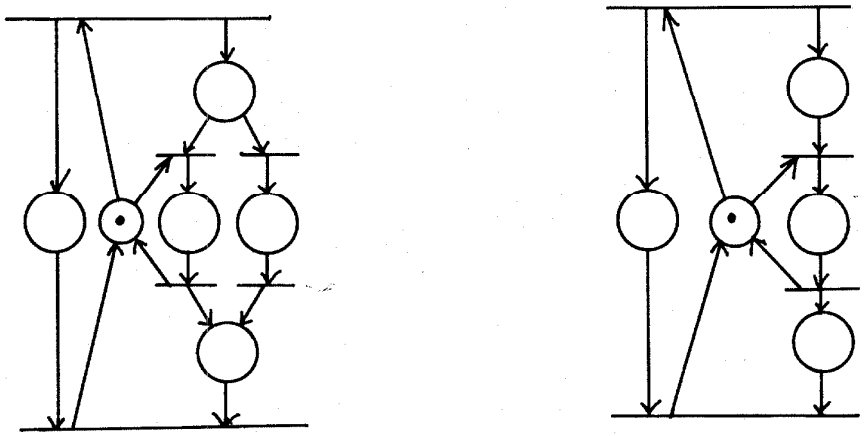


(a) (b) (c)

Figure 4–4. Pathological mutual exclusions.

For (c) (see Figure 4–4 (c)): If $a^- = b_-$ then this transition cannot be fired since the two input places $b\circ$ and $\mu\circ$ must each have tokens, but this cannot happen.

For (d) (see Figure 4–5 (a) and (b)): To enter process a, the mutual exclusion becomes busy. Within the structured process c, there will be deadlock when process b wants to enter the mutual exclusion because it is already busy. a cannot leave the mutual exclusion until all the inputs to the output transition are marked, but this cannot happen until all the internal processes of c are terminated. Figure 4–5 (a) is an example of a deadlock causing structure, while (b) has danger of deadlock.



(a) (b)

Figure 4–5. More pathological mutual exclusions.

Let us call the pathological mutual exclusions in the previous Lemma, type (a), (b), (c), and (d), respectively.

Lemma. All pathological mutual exclusions are of type (a), (b), (c), or (d).

Proof: Since structured nets are live and safe, the introduction of the mutual exclusion with the place m_o is the cause of the deadlock, so only the inputs and outputs of the mutual exclusion need be considered. Let $m = \mu(a, b)$ be a pathological mutual exclusion. Then there is a marking where none of the inputs or outputs of the mutual exclusion are firable. There are two possible cases: m_o is either marked or unmarked.

Case (1) If m_o is marked, then it means that both of the input transitions a^- and b^- are blocked. The only way these transitions can be blocked is because there are more than one arc from m_o to each of them but m_o has only one token. Since there are only two arcs from m_o, this situation can arise only when the two inputs are the same. This is pathological mutual exclusion of type (a).

Case (2) If m_o is empty, then one of the two component processes, say a, is busy and its output transition a_- is blocked. This implies that there is an empty input place, p, with $p \rightarrow a_-$. If this place belongs to the mutual exclusion, then we have type (c). If this place belongs to the other process, b, then we have type (b). If this place belongs to a process c which is not a component of the mutual exclusion, then c is blocked because it follows b. This implies that b is blocked while not having any common transitions with a, so it must be in a π relation to the busy process, giving us case (d). □

Structured Nets with Mutual Exclusion

Definition. The class of Petri nets obtained by allowing structured processes to have normal mutual exclusion constructs will be called *μ-structured processes.* *μ-structured nets* are obtained when the μ-structured processes are augmented . A μ-structured process is *live* and *safe* if its associated net is live and safe.

Notation. μ-structured nets will be denoted as a pair $\langle x(a_1, a_2, \ldots, a_n), M \rangle$ where x is a structured net and M is the set of mutual exclusions. For example, $\langle \pi(\sigma(a, b), \sigma(c, d)), \{ \mu(b, c) \} \rangle$.

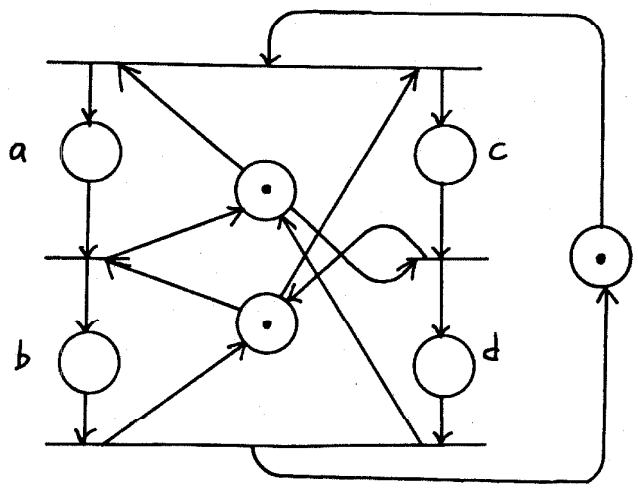


Figure 4–6. A μ-loop.

The pathological μ's cause deadlock by virtue of their being applied incorrectly. Another way for the mutual exclusions to cause deadlock is by mutual interaction at a global level. In Figure 4–6, there is a structured net $\odot\pi(\sigma(a,b),\sigma(c,d))$ with mutual exclusions $\mu(a,d)$ and $\mu(b,c)$. Deadlock occurs when processes a and c are busy. This comes about because the two mutual exclusions both precede the other and the two structured process are concurrent. This configuration of μ's will be called a ***deadly μ-loop***.

Definition. A set of mutual exclusions $\{m_1, m_2, \ldots, m_n\}$ is called a ***μ-loop*** if $m_i \vdash\!\!\!\dashv m_{i+1}$ for $1 \leq i < n$ and $m_n \vdash\!\!\!\dashv m_1$. By transitivity of $\vdash\!\!\!\dashv$, $m_i \overset{*}{\vdash\!\!\!\dashv} m_i$ for all i. A μ-loop is said to be ***deadly*** if all the μ's can be busy at a marking.

Theorem. A μ-structured net is live and safe if it has no deadly μ-loops.

Proof: The safeness is not affected by the mutual exclusion construct since it does not add tokens to any place in the net except the place in the mutual exclusion. So we must demonstrate that liveness is preserved.

($\Leftarrow$) Clearly, if a μ-structured net has a deadly μ-loop then we can reach a marking in which they are all busy and since it is a loop, none of the output transitions can fire, so it is not live.

($\Rightarrow$) Assume there are no deadly μ-loops but the net is not live, i.e. there is a marking in which some process is blocked. Since structured nets are live, the

process a must be a part of a mutual exclusion $\mu(a, b)$ that is not startable, i.e. the input to the process is not enabled. This means that b is busy. Ordinarily, b will become free unless its output transition is blocked. Because of the previous Lemma, this can happen only if there is a process c such that $\sigma(b, c)$ and c is in a mutual exclusion with another process which is busy. At each stage we have a process which is blocked by another which is in mutual exclusion with another. Since our nets are finite, this must stop at a finite number of mutual exclusions, yet the configuration must remain at an impasse, which implies the existence of the deadly μ-loop. □

Lemma. A μ-loop is deadly if and only if the component structured processes are all concurrent.

Proof: A μ-loop of n mutual exclusion constructs requires n σ structured processes. So in order for all of the mutual exclusions to be busy, all the structured processes must be concurrent. □

Knowing that deadly μ-loops are the cause of deadlocks in μ-structured nets, is it possible to recursively define a class of Petri nets which has mutual exclusion but is live and safe? Unfortunately, no. Deadlocks are global phenomenon and recursive definitions cannot distinguish between a single occurrence of a mutual exclusion and a deadly μ-loop.

However, testing for deadly μ-loops is not hard. An outline of an algorithm for determining deadly μ-loops is given below. Let us call a structured net that consists of a pair of processes where the input of one is the output of the other, a *sequential pair.*

An algorithm (for determining deadly μ-loops):

First, determine if there are any loops by constructing a directed graph where the nodes are the mutual exclusion constructs and the arcs represent the directly precedes relation on mutual exclusions. Next, test the graph for any cycles by scanning through the nodes and making a tree of nodes reachable from a start node checking for repetitions.

Second, if there are cycles, determine whether the loops are deadly by testing whether the sequential pairs are concurrent. This is done by checking whether they

occur as substructures of a π-construct.

Repetition Construct

Definition. The ***repetition construct*** between n processes $a_1, a_2, \ldots, a_n$, written $\rho(a_1, a_2, \ldots, a_n)$ is a Petri net structure with two new places and a transition with the following structure:

$$\{a_1^-, a_2^-, \ldots, a_n^-\} \rightarrow p_1 \rightarrow t \rightarrow p_2 \rightarrow \{a_{1-}, a_{2-}, \ldots, a_{n-}\}.$$

The new transition t will be called the ***shared*** transition (of the repetition construct).

When one of the component processes start, the shared transition becomes enabled to fire, and it must fire before the output of the process can fire. Figure 4-7 is an example of the repetition construct for two processes.

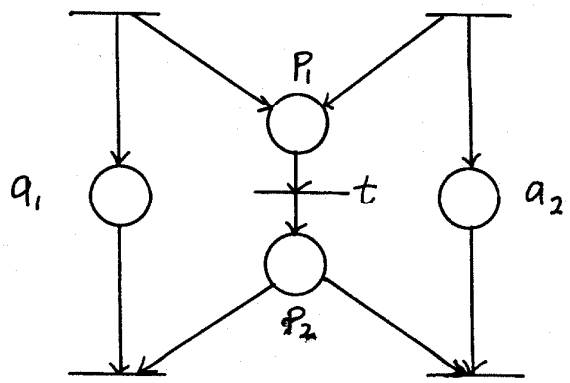


Figure 4–7. Repetition construct for two processes.

Lemma. The repetition construct preserves liveness, but not safeness.

Proof: (Not safe) If two or more of the component processes are concurrent, then the place p_1 of the construct can be marked simultaneously, so the net is not safe. (Live) Assume that each of the component processes are live, then after the

firing of the input of a process the shared transition will be enabled, and after it fires, the output of the process will be enabled. □

Definition. Structured processes with the repetition construct will be called *ρ-structured processes*. Augmented ρ-structured processes will be called *ρ-structured nets*.

Theorem. A ρ-structured net is live and safe if and only if the processes in each repetition construct are pairwise not concurrent.

Proof: Since they cannot be marked simultaneously, the shared places cannot be unsafely marked. Liveness is not affected by the construct. □

Liveness and Safeness of Hierarchical Nets

In μ-structured nets and ρ-structured nets, the mutual exclusion and repetition constructs were applied only between processes. The two constructs are mutually independent in their effect on the nets, so that can be simultaneous applied without introducing extra complications.

Definition. With simultaneous application of mutual exclusion and repetition construct between processes in a structured processes, *μ-ρ-processes* are obtained. Augmented μ-ρ-processes will be called *μ-ρ-nets*.

With the characterization for liveness of μ-structured nets and safeness for ρ-structured nets given, the μ-ρ-processes can be used as building blocks to form *hierarchical nets*. The construction has the same flavor as for structured nets.

Definition. A *hierarchical process* is either

(a) a live and safe μ-ρ-process, or

(b) a hierarchical process with a process replaced by another hierarchical process, i.e. a net substitution $[h_1/a]h_2$ where a is a process and h_1 and h_2 are hierarchical processes with no process names in common.

Definition. A *hierarchical net* is an augmented hierarchical process. $h, h_1, h_2, \ldots,$

will denote hierarchical processes, and $H, H_1, \ldots,$ will denote hierarchical nets.

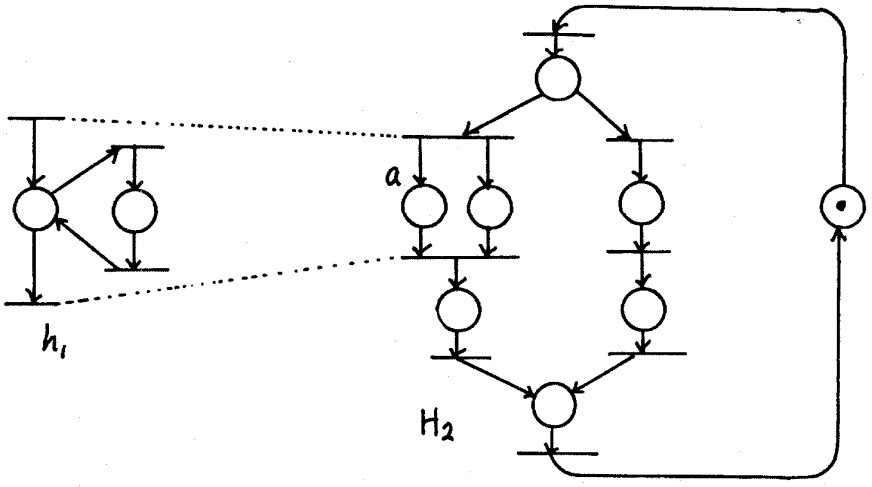


Figure 4–8. Example of a net transformation $[h_1/a]H_2$.

Theorem. Hierarchical nets are live and safe.

Proof: The proof is by induction on the construction of a hierarchical net.

(a) If the hierarchical net is an augmented μ-ρ-process, then, since the μ-ρ-net is assumed live and safe, it is too.

(b) Otherwise, assume that the hierarchical net is the result of a net substitution, $[h_1/a]H_2$, where a is a process in H_2. By induction hypothesis h_1 and H_2 are both live and safe. The net substitution replaces process a with h_1 by identifying the input and output of a with that of the input and output of h_1. Since h_1 is assumed live and safe, the output of h_1 will eventually fire after the input has fired, and so the whole net is live and safe. □

Hierarchical nets provide us with a way of synthesizing complex structures made of processes and mutual exclusions and repetitions. By restricting the use of the mutual exclusion and repetition constructs to one level at a time, in a μ-ρ-net, hierarchical nets are easily shown live and safe. We can go one step further in this hierarchy and allow mutual exclusion and repetition constructs to be applied to processes in different hierarchical nets.

Definition. A *distributed hierarchical net* is either

(a) a set of hierarchical nets where the mutual exclusion and repetition constructs are allowed between processes of the different nets, or

(b) a distributed hierarchical net where a process is replaced by a hierarchical process.

Notation. Distributed hierarchical nets are represented as an ordered pair

$$\langle \{H_1, \ldots, H_n\}, \{\mu(p_1, q_1), \ldots, \mu(p_m, q_m), \rho(a_1, b_1), \ldots, \rho(a_l, b_l)\}\rangle$$

where the processes in each μ or ρ construct are from different hierarchical nets .

Theorem. Distributed hierarchical nets are live and safe if the set of mutual exclusions does not contain deadly μ-loops and the repetition constructs are not formed between concurrent processes.

Proof: Since the different hierarchical nets can be viewed as concurrent processes the proof is identical to the liveness and safeness proof of μ-ρ-nets. □

Due to the way distributed hierarchical nets are constructed, there are certain structures that cannot be obtained. Figure 4–8 gives couple of Petri nets that are not obtainable by hierarchical nets construction.

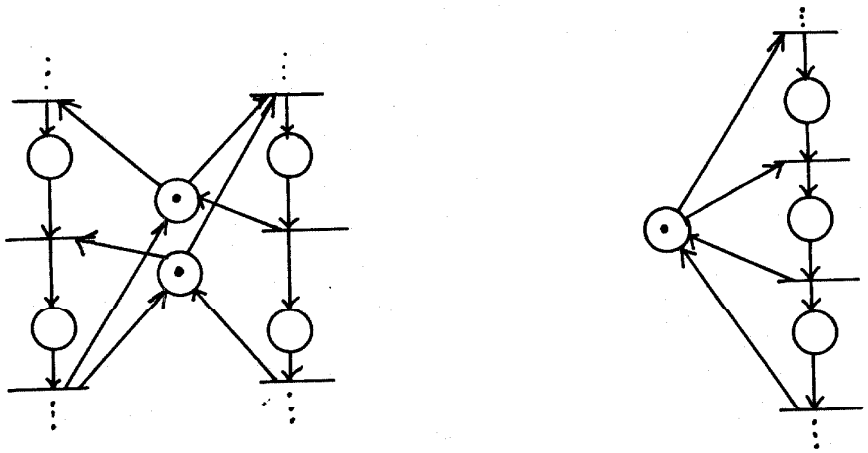


Figure 4–8. Petri nets not generatable as distributed hierarchical nets.

Examples of Applications

1. The Dining Chinese Philosophers

The problem. (Dijkstra [1968]) A group of Chinese philosophers are sitting around a round table. Each has a bowl in front of him. Between two adjacent bowls there is a chopstick. So there are only as many chopsticks as there are philosophers. A large bowl of food is at the center of the table. Our philosophers do one of two things: they either think, or they eat. Thinking can be done without any material aids, but eating requires the use of a pair of chopsticks. So, before a philosopher can eat, he must pick up the two chopsticks on each side of his bowl.

The Dining Chinese Philosophers problem was solved by the Petri net of Figure 1–6. The solution assumes that each philosopher can simultaneously pick up both his left and right chopsticks. This three-way synchronization may be considered to be begging the issue since it pushes the problem back one level to that of constructing multi-way synchronizations from simpler ones.

If only two-way synchronizations are assumed, then one solution would be Figure 4–10. Each philosopher picks up his right chopstick, then his left. Obviously, this solution can deadlock. The liveness and safeness condition of distributed hierarchical nets would exclude this solution because there is a deadly μ-loop.

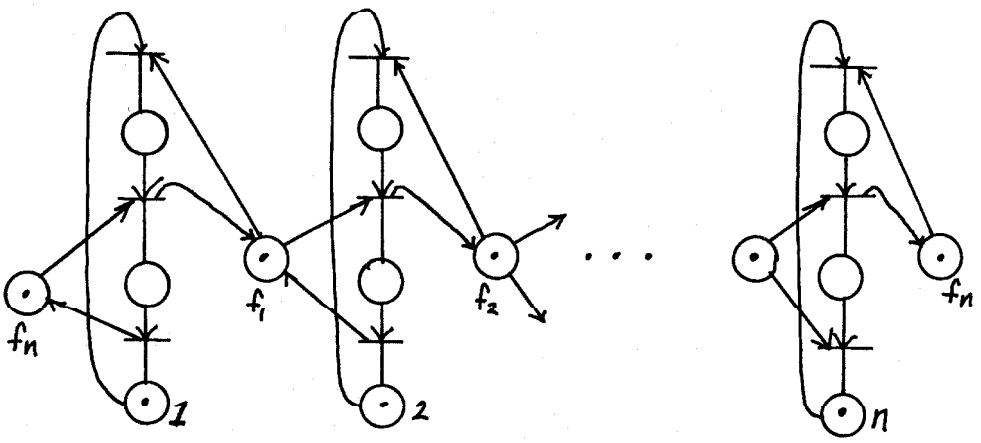


Figure 4–10. Dining Chinese Philosophers solution using only 2-way synchronizations.

By removing the deadly μ-loop a live and safe solution is obtained in Figure 4–11. Here one of the philosophers is left handed and picks up his left chopstick before his right.

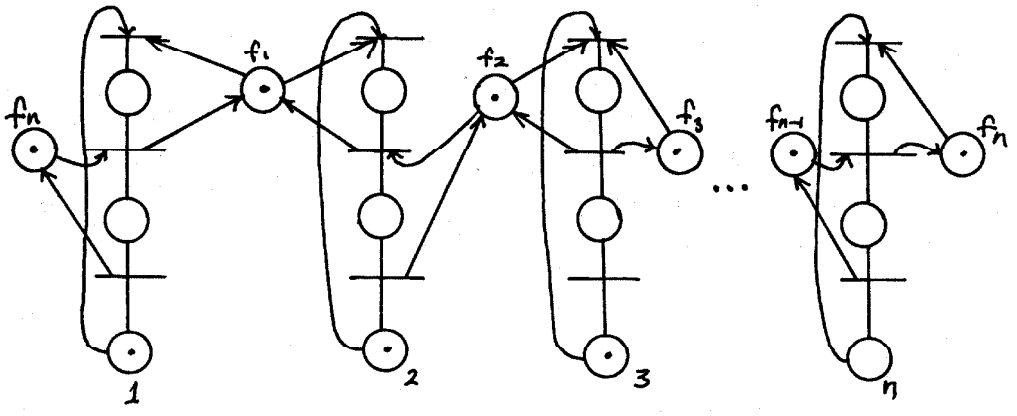


Figure 4–11. A live and safe solution.

Now, some may consider this solution inelegant because of lack of symmetry. The solution of Figure 4–12 is more symmetrical where all the even numbered philosophers pick up their left chopsticks first. Given only two-way synchronizations and no global mutual exclusion asymmetry is inevitable.

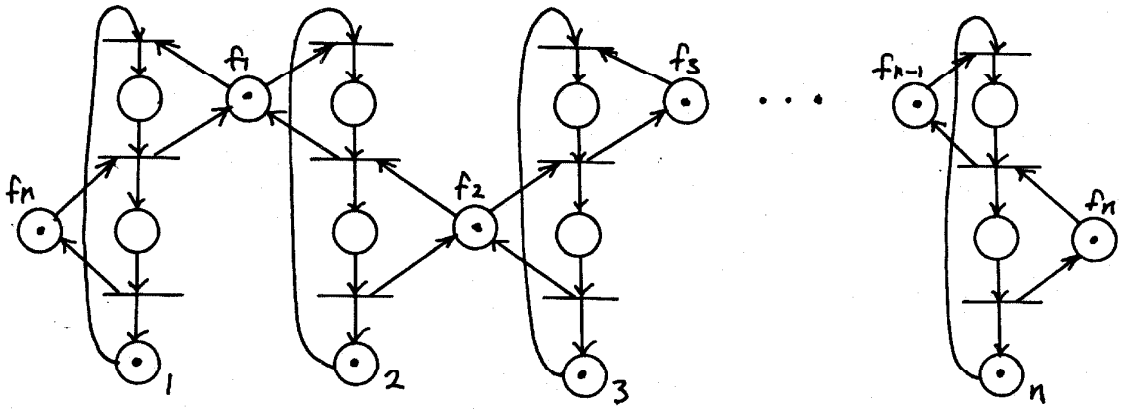


Figure 4–12. A more symmetric solution.

The standard solutions using the **P** and **V** operations essentially simulates three-way synchronizations using one global critical section.

2. The Smokers Problem

The problem. (Patil [1971]) Three ingredients, paper, tobacco, and matches, are required to make a cigarette and smoke it. There are three smokers around a table each already possessing different one of the three ingredients. On the table two of the three ingredients will be placed. The smoker with the necessary third ingredient may take the ingredients from the table and smoke. The others may not interfere. No new ingredients will be placed on the table until the previous ones have been consumed.

The Petri net solutions to the smokers problem presented by Campbell and Lauer [1975] are recast into distributed hierarchical nets solutions.

The sequential solution looks like Figure 4–13. Essentially, the solution uses a scheduler that keeps track of the ingredients and then allows the right smoker to go ahead. The places p_i inside a ρ-construct indicate the appearance of paper, etc. The places $\overline{p_i}$ in a ρ-construct means that the smoker with the paper can smoke.

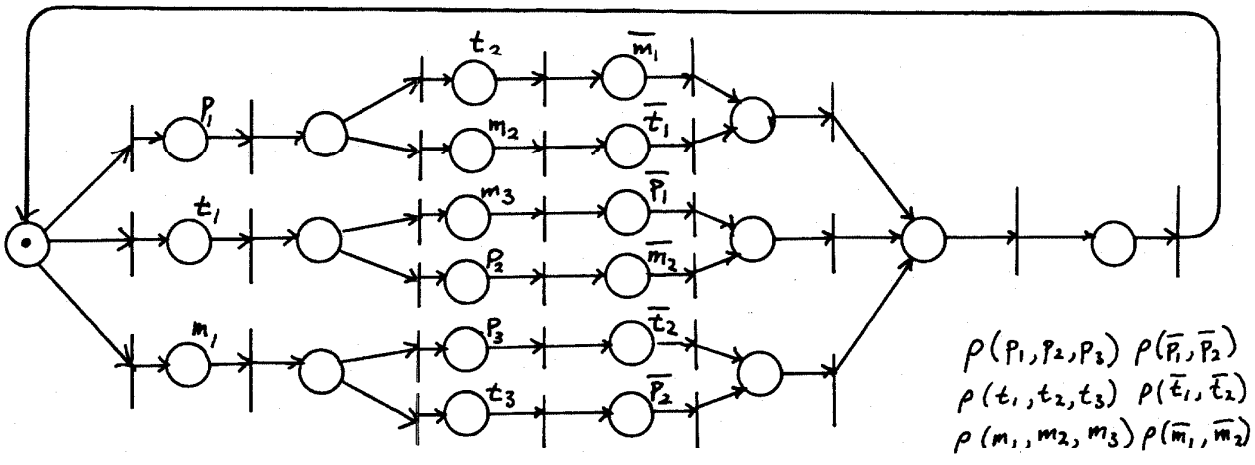


Figure 4–13. Sequential solution to the smokers problem.

The concurrent version looks like Figure 4–13. After the pruning is done to the path expression solution, it closely resembles Figure 4–13.

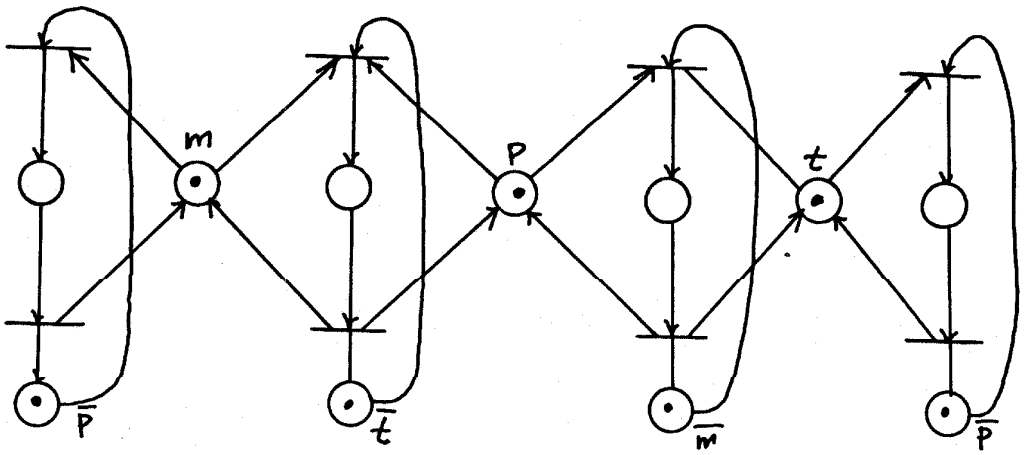


Figure 4–13. Concurrent solution.

These two solutions are live and safe since they satisfy the requirements on the use of mutual exclusion and repetition constructs.

The use of three-way synchronization makes the concurrent solution quite trivial, and so the question arises of what primitives we will allow in our various solutions to distributed synchronization problems. For these considerations, hierarchical nets seem to be a useful tool.

Chapter 5

General Net Transformations

In this chapter, more general types of transformations are considered which do not necessarily preserve liveness or safeness. We formally define these transformations, give examples of when they are not nicely behaved, and then explore their use by formulating suitable restrictions on their applications. These transformations have a certain symmetry which affects the behavior suggesting certain notions of duality.

Transformations on Transitions

General Transition Refinement (GTR). Consider Figure 5–1 where the transition t is refined but the inputs and outputs are split between the two new transitions.

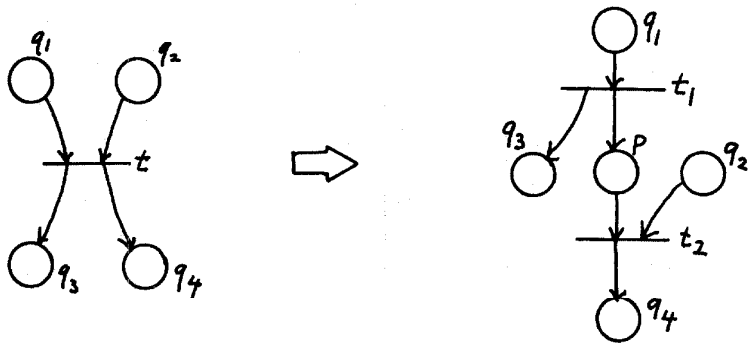


Figure 5–1. $\{q_1, q_2\} \rightarrow t \rightarrow \{q_3, q_4\} \Rightarrow \{q_1 \rightarrow t_1 \rightarrow \{q_3, p\}, \{p, q_2\} \rightarrow t_2 \rightarrow q_4\}$

In general we can have:

$$A \rightarrow t \rightarrow B \Rightarrow \{A_1 \rightarrow t_1 \rightarrow B_1, A_2 \rightarrow t_2 \rightarrow B_2\}$$

where $A_1 \cup A_2 = A \cup \{p\}$, p is in A_2, and A_1 and A_2 are disjoint, and $B_1 \cup B_2 = B \cup \{p\}$, p is in B_1, and B_1 and B_2 are disjoint, and neither A_1 nor B_2 are empty.

Proposition. General transition refinement preserves liveness, but not safeness.

Proof: If the original net is live, then each input place of t will be filled with a token. Then in the resulting net, the two new transitions will also be enabled, so the new net is also live. To show that the transformation does not preserve safeness, see Figure 5–2, where p is no longer safe after transition t is refined, although the original net is clearly live and safe. □

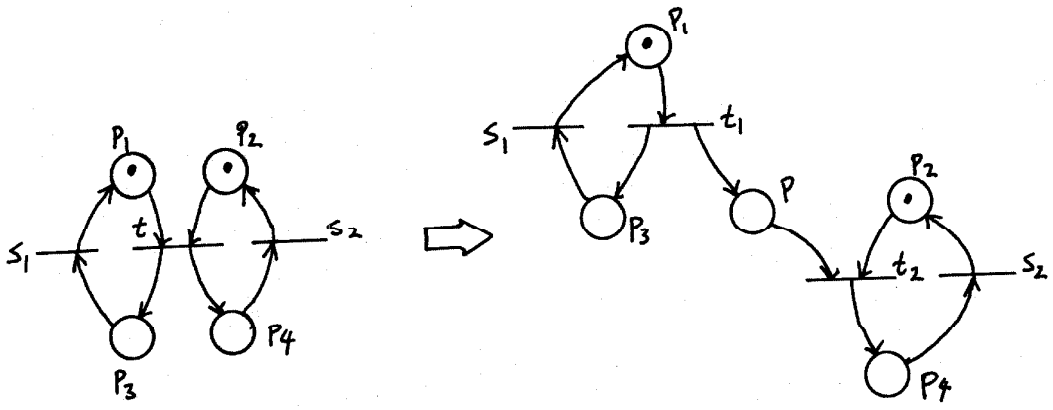


Figure 5–2. An example of **GTR** which does not preserve safeness.

General Transition Splitting (GTS). In the transition splitting transformation, $p_1 \rightarrow t \rightarrow p_2 \Rightarrow p_1 \rightarrow \{t_1, t_2\} \rightarrow p_2$, the transition has unique input and output places. In general, the transition can have multiple inputs and outputs (see Figure 5–3).

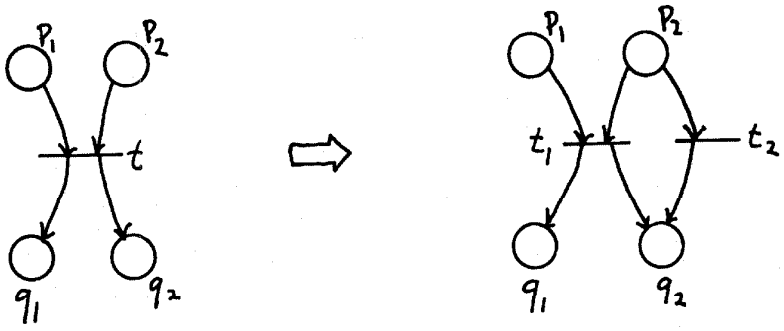


Figure 5–3. $\{p_1, p_2\} \rightarrow t \rightarrow \{q_1, q_2\} \Rightarrow \{\{p_1, p_2\} \rightarrow t_1 \rightarrow \{q_1, q_2\}, p_1 \rightarrow t_2 \rightarrow q_1\}$

General transition splitting distributes the input and output places among the new transitions:

$$\bullet t \rightarrow t \rightarrow t\bullet \Rightarrow \{\bullet t_1 \rightarrow t_1 \rightarrow t_1\bullet, \bullet t_2 \rightarrow t_2 \rightarrow t_2\bullet\}$$

where $\bullet t_1 \cup \bullet t_2 = \bullet t$ and $t_1\bullet \cup t_2\bullet = t\bullet$ and neither $\bullet t_1$ nor $\bullet t_2$ can be empty.

Proposition. General transition splitting preserves safeness, but not liveness.

Proof: Safeness is preserved since the new transition which is created does not add any extra tokens into the net. Liveness is not preserved as can be seen from Figure 5–4 where the firing of t_2 will cause deadlock. □

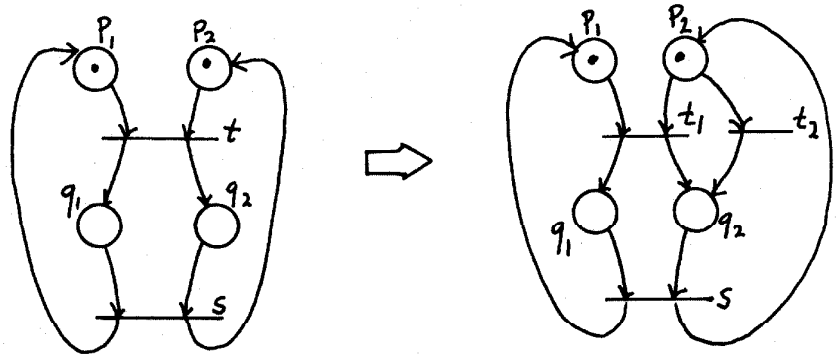


Figure 5–4. General transition splitting with danger of deadlock.

Transition Merge (TM) - Synchronization. This is one of the fundamental operations that is used to compose smaller Petri nets into larger ones by joining identifying two transitions as one with the resulting transition having all the inputs and outputs of the components:

$$\{ \bullet t_1 \rightarrow t_1 \rightarrow t_1 \bullet, \bullet t_2 \rightarrow t_2 \rightarrow t_2 \bullet \} \Rightarrow \bullet t_1 \cup \bullet t_2 \rightarrow t \rightarrow t_1 \bullet \cup t_2 \bullet.$$

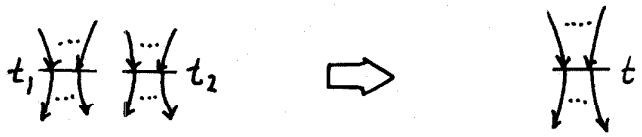


Figure 5–5. Synchronization.

Proposition. Synchronization preserves safeness, but not liveness.

Proof: No new tokens are produced, therefore safeness is preserved. To see how liveness is not preserved see the example in Figure 5–6. □

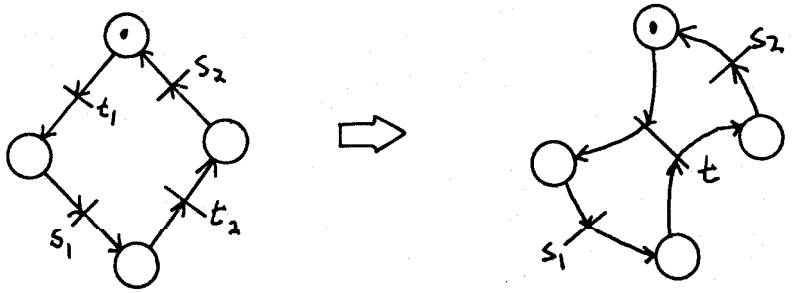


Figure 5–6. Synchronization has danger of deadlock.

Definition. Two transitions are said to be ***independent***, if when one is enabled, it is possible to reach a marking where the other is also enabled without the firing of the first enabled transition. In general, testing for independence is hard.

Proposition. Synchronization preserves liveness if applied between independent transitions.

Proof: Clear from definition of independent transitions. □

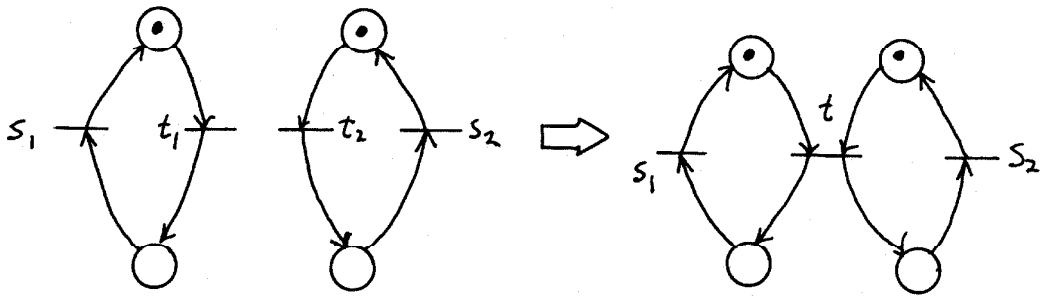


Figure 5–7. Synchronization of two independent transitions.

Transition Division (TD). The opposite of synchronization is to divide one transition into two, dividing the input and output arcs between them, making sure that each has at least one input arc:

$$\bullet t \rightarrow t \rightarrow t\bullet \Rightarrow \{ \bullet t_1 \rightarrow t_1 \rightarrow t_1\bullet, \bullet t_2 \rightarrow t_2 \rightarrow t_2\bullet \}$$

where $\bullet t = \bullet t_1 \cup \bullet t_2$ and $t\bullet = t_1\bullet \cup t_2\bullet$ and none of the sets of inputs or outputs are empty.

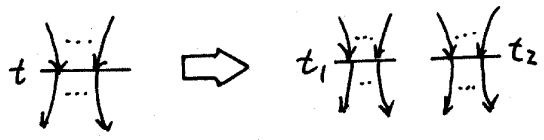


Figure 5–8. Transition division.

Proposition. Transition division preserves liveness, but not safeness.

Proof: Clearly if the original transition was live, then the resulting transitions are also. To see that safeness is not preserved, see the following example in Figure 5–9. The place p was safe, but after separating t into t_1 and t_2 it is not. □

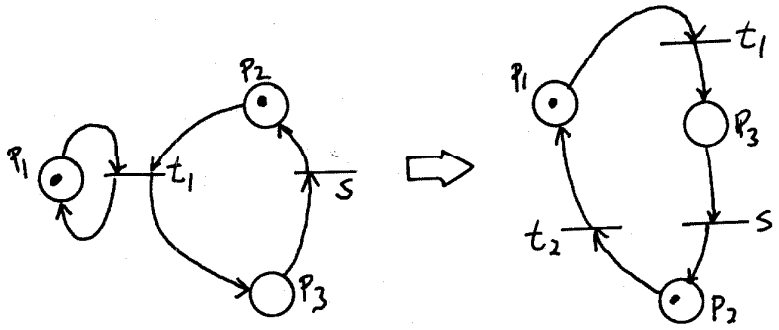


Figure 5–9. An unsafe transition division.

Transformations on Places

Same types of transformations can be defined on the places.

General Place Refinement (GPR). Analogous to the general refinement for transitions, the inputs and outputs can be split between the two new places.

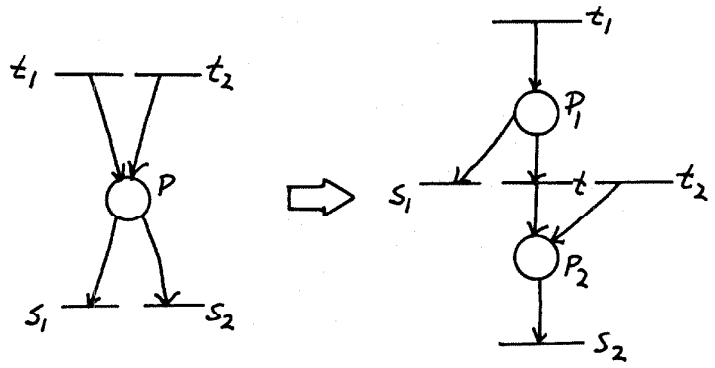


Figure 5–10. General place refinement with two inputs and outputs.

In general we can have:

$$A \rightarrow p \rightarrow B \Rightarrow \{ A_1 \rightarrow p_1 \rightarrow B_1, A_2 \rightarrow p_2 \rightarrow B_2 \}$$

where $A_1 \cup A_2 = A \cup \{t\}$, p is in A_2, and A_1 and A_2 are disjoint, and $B_1 \cup B_2 = B \cup \{t\}$, p is in B_1, and B_1 and B_2 are disjoint, and neither A_1 nor B_2 are empty.

Proposition. General place refinement preserves safeness, but not liveness.

Proof: Since the new transition does not add any new tokens to the net, it preserves safeness. It does not preserve liveness, as can be seen in Figure 5–11 where t is no longer live after p is refined. □

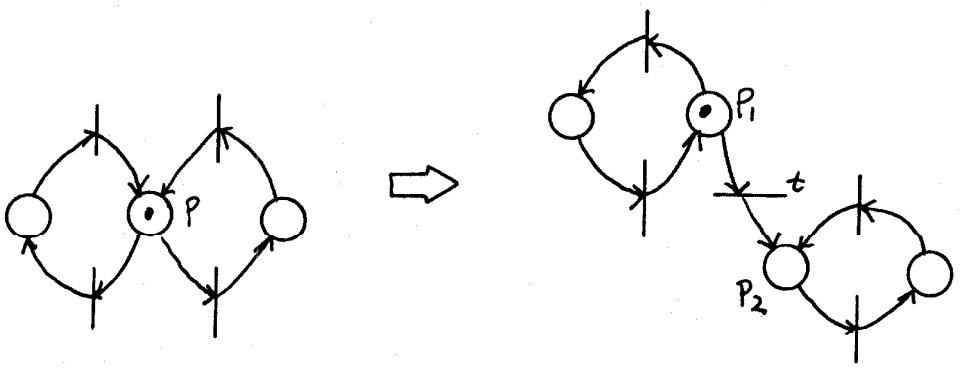


Figure 5–11. General place refinement does not preserve liveness.

General Place Splitting (GPS). The place, which is being split, can have multiple input and output transitions:

$$\bullet p \rightarrow p \rightarrow p\bullet \Rightarrow \{ \bullet p \rightarrow p \rightarrow p\bullet, t \rightarrow q \rightarrow s \}$$

where $t \in \bullet p$ and $s \in p\bullet$.

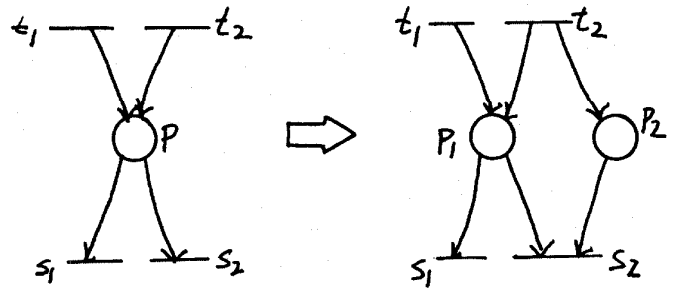


Figure 5–12. An example of general place splitting.

Proposition. General place splitting preserves liveness, but not safeness.

Proof: If all the input transitions are live, then all the output transitions are also live. After the splitting of the place, the transition s requires two places to be

marked to fire. Suppose the firing of another transition in $p\bullet$ disables s. Since the net was assumed live and safe before the transformation, the transition t must be live, so that s is also live. The place q that is introduced is now unsafe because it accumulates as many tokens as the number of times t fires but not s. See Figure 5–13. □

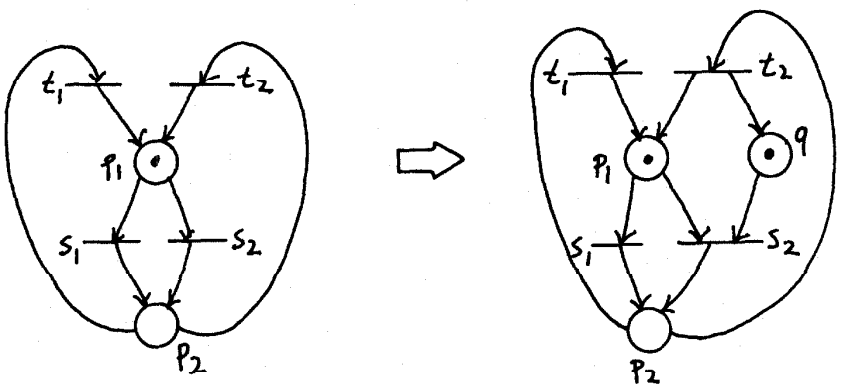


Figure 5–13. After the firing of s_1 and t_2, q is not safe.

Place Merge (PM). Two places are merged into one so that the new place has the combined input and output of the two:

$$\{ \bullet p_1 \rightarrow p_1 \rightarrow p_1\bullet, \bullet p_2 \rightarrow p_2 \rightarrow p_2\bullet \} \Rightarrow (\bullet p_1 \cup \bullet p_2) \rightarrow p \rightarrow (p_1\bullet \cup p_2\bullet).$$

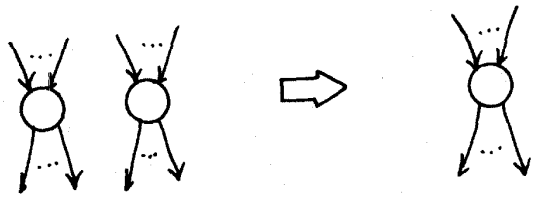


Figure 5–14. Place merge.

Proposition. Place merge preserves liveness, but not safeness.

Proof: Clearly, it cannot introduce danger of deadlock. However, if we merge two places that can be simultaneously marked, then the new place is no longer safe. □

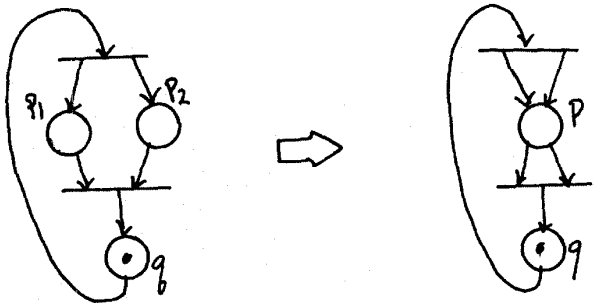


Figure 5–15. Merge of places does not preserve safeness.

Proposition. Place merge preserves liveness and safeness if it is applied to two places which can never be simultaneously marked.

Proof: Obvious. □

Place Separation (PS). One place can be made into many places and the inputs and outputs divided between them. Without loss of generality, we define the place separation producing two places:

$$\bullet p \rightarrow p \rightarrow p\bullet \Rightarrow \{ \bullet p_1 \rightarrow p_1 \rightarrow p_1\bullet, \bullet p_2 \rightarrow p_2 \rightarrow p_2\bullet \}$$

where $\bullet p = \bullet p_1 \cup \bullet p_2$ and $p\bullet = p_1\bullet \cup p_2\bullet$:

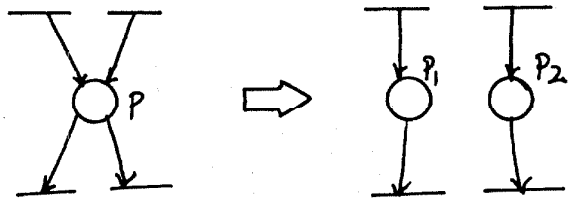


Figure 5–16. An example of place separation.

Proposition. Place separation preserves safeness, but not liveness.

Proof: No tokens are introduced that does not already exit in the net. Liveness is not preserved, as can be seen in Figure 5–8.

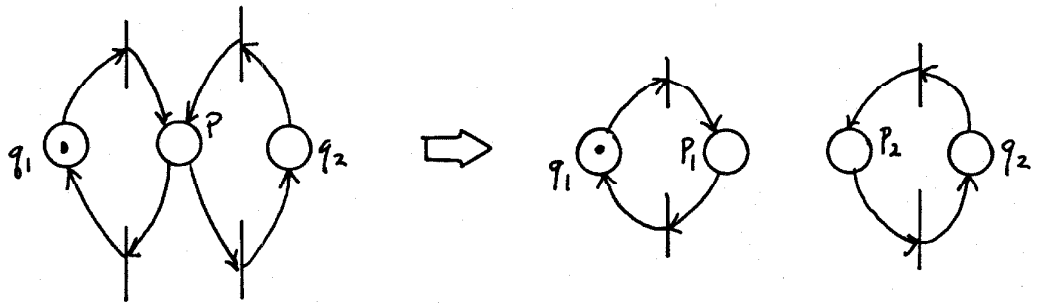


Figure 5–17. Place separation does not preserve liveness.

Symmetric Aspects of Net Transformations

The net transformations fall into two main classes. The first class consists of the transformations which do not add new arcs to the net: the merges and

divisions of places and transitions (**PM, TM, PD, TD**). The second class of transformations add new arcs: the splittings and the refinements (**GPS, GTS, GPR, GTR**). In each of the net transformations, one of safeness or liveness is not preserved. Transformations **PM, TD, GPS** and **GTR** do not preserve safeness, while **TM, PD, GTS** and **GPR** do not preserve liveness.

To see the symmetry, a notion of duality for Petri nets is needed.

Definition. The *dual* of a Petri net is another net with the same structure, but with the places and the transitions interchanged. Duality is a symmetric relation.

Since the net transformations are transformations of Petri nets into other Petri nets, each net transformation has a corresponding dual net transformation. For example, **PM** is the dual of **TM**, and **PD** is the dual of **TD**, since those are obtained when the places and transitions are interchanged.

Definition. A *characteristic* of a net transformation is either the property "not safeness preserving" or its *dual*, "not liveness preserving."

If n denotes a net transformation, then let n' be its dual. Also, if z denotes a characteristic, then z' is its dual. Let $\phi(n, z)$ be the sentence "net transformation n is z." For example, if n is **GTR** and z is "not safeness preserving" then $\phi(n, z)$ means "General transition refinement is not safeness preserving."

Theorem. $\phi(n, z)$ implies $\phi(n', z')$.

Proof: By collecting the results of all the previous propositions. □

Since the net transformations are made up of two components, whether they act on places or transitions and some operation, let us represent a net transformation by $n(x, y)$, where x can be either "place" or "transition" and y can be one of the four operations: "merge," "division," "splitting," or "refinement." So if x is "place" and y is "splitting," then $n(x, y)$ denotes the net transformation **GPS**.

Definition. Let merging (**M**) be the dual of division (**D**), and splitting (**S**) the dual of refinement (**R**).

Define $\psi(x, y, z)$ to mean "the net transformation $n(x, y)$ is z."

Theorem. $\psi(x, y, z)$ implies $\psi(x', y, z')$ and $\psi(x, y', z')$ and $\psi(x', y', z)$.

Proof: Again by inspection of the characteristics of each net transformation. □

For the net transformations in the first class, as far as the characteristics are concerned, the merging of two places has the same effect as the division of a transition into two and the merging of two transitions the same effect as the division of a place.

The net transformations of the second class can be viewed as introductions of places (**GPS**, **GTR**) or introductions of transitions (**GTS**, **GPR**). Introductions of places do not preserve safeness while introductions of transitions do not preserve liveness.

Notions of duality have been around, but not much has been done because it was not clear what to do about the tokens in places when the places where changed to transitions. The net transformations introduced in this chapter suggest a way of reasoning about duality without the need for explicit markings, and still say something about the behavior of Petri nets.

Chapter 6

Conclusions

Petri Nets as Design Tool

Although Petri nets provide only a small number of primitives with which to model systems, by suitable abstraction, any level of a finite system can be modeled. Since the firings of transitions are the only actions possible, not all activities can be modeled easily, and some require complex Petri net structures.

To use Petri nets for more complex computations, some researchers have interpreted the transitions as representing assignment statements. The net represents the flow of control and its topology, but hides the flow and spatial location of data by assuming it is independent of the net.

In concurrent computation, the distinction between data and control is undesirable. Concurrent computation has spatial as well as temporal components, and by interpreting the transitions as functions controlling the flow of information — both data and control — the topological aspects of a computation become clearer. For an operation to happen, all the necessary data must be available, and this requires that arcs link the different parts of a net.

The mutual exclusion construct is a good example of the actual communications that are required to implement mutual exclusion. Mutual exclusion between spatially distant processes is costly, and simulating synchronization "primitives" may require lots of communications. The topology of the Petri net also tells us what the topology of a physical implementation must be. If the Petri net is not planar, then any implementation won't be either.

The transitions and places of a Petri net are too small as units in the design process. In this work, the unit of action was defined as a *process*, and the control structures (mutual exclusion and repetition constructs) and net transformations

were defined in terms of it. The characterization for the safe use of the mutual exclusion construct would have been very complex without this higher level view.

The hierarchical approach to Petri nets gives the designer a formalism that combines the control structures of regular programming notations and the topological information to reason about concurrency. It is not meant to represent a complete programming language dealing with values of computations.

Not all control structures can be modeled using Petri nets without fundamental modifications. For example, the busy waiting, where one process tests a condition repeatedly until the condition becomes true, and then executes an action, cannot be modeled. In the Petri net version, even after the condition becomes true, there is nothing to force the net to execute that action.

Petri Nets and Verification

Once a system has been modeled as a Petri net, the task of verifying remains. For general questions like liveness, safeness, and reachability, procedures exist so that they can be done automatically. Hierarchical nets have been defined to make testing for liveness and safeness easy, so that verification can be concentrated on the more specific behaviors of a system.

Proving a given property of a net will usually require simulating the net, which is quite hard for general nets, but hierarchical nets have structural information that can be utilized. The processes at different levels of abstraction all begin and end at unique transitions. The parallel construct can be separately considered from the mutual exclusion and repetition constructs that are added later.

Because Petri nets are usually considered graphically, they are hard to manipulate (by hand). What would be nice is a symbolic or an algebraic notation with rules for forming and transforming Petri nets and their behavior. The transformations presented in this work may be a part of such a framework. A work done along these lines is by Kotov [1978] on "An Algebra for Parallelism Based on Petri Nets." In his paper, however, the main operator is that of identifying of transitions or places named the same, so that it is difficult to have any sort of hierarchy.

Before we can have a nice formalism for Petri nets, not only restrictions but

also more structured approaches are needed. The spatial nature of Petri nets is the main cause of the difficulty. As yet, we do not have a very clear understanding of how to manipulate general graphical structures.

Issues in Concurrent Computation

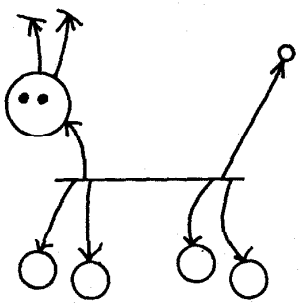
The essential feature of concurrent computation is that different computing elements are distributed in space and must communicate with each other to perform a computation. Communication requires synchronization of two or more processes. In hierarchical nets, synchronizations are done two different ways. The π-construct provides the structured synchronization where multiple processes are created and merged. The mutual exclusion and repetition constructs impose synchronizations among arbitrary processes, some of whom are concurrent with each other.

Concurrency manifests itself in two distinct forms. The first, which I call "parallelism," is when the processes are part of an enclosing process, as in the parallel construct. The second, "independence," is when the processes are totally unrelated except for special control structures like mutual exclusion that are imposed.

Applying multiple mutual exclusions to a process has the effect of making multiway synchronizations. As we saw in the Dining Philosophers problem, by having each philosopher in mutual exclusion with both his left and right neighbors, we are using three-way synchronization, and the solution is obvious. The graphical representation of Petri nets together with explicit structures that embody the notions of mutual exclusion and repetition help clarify what is happening in distributed synchronization problems.

Crucial in the design of large systems is the ability to compose smaller parts to make a bigger part. The primary method of composition in hierarchical nets has been the identification of transitions — the calling and returning of subprocesses. The mutual exclusion and repetition constructs have been used to link different hierarchical nets, but the resulting net remains very loosely coupled. There remain many other possibilities that may offer insight into concurrent systems: buffered communication channels, one way flow of information, identification of places etc.

The general questions of manipulation and composition of structured objects exhibiting dynamic behavior remain for future work.



References

E. Dijkstra,

[1968] Cooperating Sequential Processes, in F. Genuys (Editor), ***Programming Languages,*** New York: Academic Press, (1968), pages 43-112.

Michael Hack,

[1972] Analysis of Production Schemata by Petri Nets, Technical Report 94, Project MAC, MIT, Cambridge, Massachusetts, (February 1972), 119 pages.

A. Holt and F. Commoner,

[1970] Events and Conditions, ***Record of the Project MAC Conference on Concurrent Systems and Parallel Computation,*** New York: ACM (June 1970), pages 1-52.

V.E. Kotov,

[1978] An Algebra for Parallelism Based on Petri Nets, ***Mathematical Foundations of Computer Science,*** 7th Symposium, J. Winkowski (Ed.), Springer-Verlag, pages 39-55.

P.E. Lauer and R.H. Campbell,

[1975] Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes, ***Acta Informatica 5,*** (1975), pages 297-332.

R. Lipton,

[1976] The Reachability Problem Requires Exponential Space, Research Report 62, Dept. of Computer Science, Yale University, New Haven, Connecticut, (January 1976), 15 pages.

Ernst W. Mayr,

[1981] An Algorithm for the General Petri Net Reachability Problem, *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, Milwakee, Wisconsin, May 11-13, 1981, pages 238-246.

James L. Peterson,

[1981] *Petri net Theory and the Modeling of Systems*, Prentice-Hall, (Englewood Cliffs, 1981).

C. A. Petri,

[1962] Kommunikation mit Automaten, Ph.D. Dissertation, University of Bonn, West Germany, (1962), (in German); also MIT Memorandum MAC-M-212, Project MAC, MIT, Cambridge, Mass.

R. Valette,

[1979] Analysis of Petri Nets by Stepwise Refinements, *Journal of Computer and System Sciences* 18, (1979), pages 35-46.