Analysis of Parameterized Networks

by

Siamak Nazari

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.
Siamak Nazari

Abstract

In particular, the thesis will focus on parameterized networks of discrete-event systems. These are collections of interacting, isomorphic subsystems, where the number of subsystems is, for practical purposes, arbitrary; thus, the system parameter of interest is, in this case, the size of the network as characterized by the number of subsystems. Parameterized networks are reasonable models of real systems where the number of subsystems is large, unknown, or time-varying: examples include communication, computer and transportation networks. Intuition and engineering practice suggest that, in checking properties of such networks , it should be sufficient to consider a "testbed" network of limited size. However, there is presently little rigorous support for such an approach.

In general, the problem of deciding whether a temporal property holds for a parameterized network of finite-state systems is undecidable; and the only decidable subproblems that have so far been identified place unreasonable restrictions on the means by which subsystems may interact. The key to ensuring decidability, and therefore the existence of effective solutions to the problem, is to identify restrictions that limit the computational power of the network. This can be done not only by limiting communication but also by restricting the structure of individual subsystems. In this thesis, we take both approaches, and also their combination on two different network topologies: ring networks and fully connected networks.

Acknowledgements

First, I would like to thank my supervisor Professor John Thistle for his great guidance, support and patience. His advice and support were always greatly appreciated. I also want to thank all my committee members: Prof. Mark Aagaard, Prof. Richard Trefler, Prof. Paul Ward, and Prof. Stephane Lafortune.

Most importantly, I'd like to thank my family for their supporting and encouraging comments, their love and faith in me.

Contents

List of Figures **vii**

1 Introduction **1**

1.1 Related Work 5

2 Ring Networks of Isomorphic Processes **12**

2.1 Preliminaries 14

2.1.1 Process Model 14

2.1.2 Observable Modal Logic 18

2.1.3 Process Equivalences 19

2.2 Computation Model 24

2.3 Undecidability Results 31

2.3.1 Ring Networks Equivalence Classes 31

2.3.2 Component and Network Blocking 36

2.3.3 Ring-Segments Equivalence Classes 37

2.4 Termination of $\mathcal{PROC}$ 41

2.4.1 Piecewise Recognizable Processes 41

2.4.2 Shuffled processes 51

3 Infinite State Modelings and Fully-Connected Networks **56**

3.1 Petri Net Preliminaries 57
3.1.1 Petri Net Models 57
3.1.2 Reachability Tree and Coverability Tree 60
3.1.3 Linear Temporal Logic 67
3.1.4 Product of Petri Nets 68
3.1.5 Variants of Ordinary Petri nets 69
3.2 Model Checking of Ordinary Petri Nets 74
3.2.1 A Decidable Fragment of Linear Temporal Logic 75
3.2.2 Model-Checking Under Fairness 83
3.2.3 Factory Example 85
3.3 Networks of Identical Processes 87
3.3.1 Computation Model 87
3.3.2 Petri Nets Modelling of Networks with Rendezvous Templates . . . 88
3.3.3 Component and Network Blocking 92
3.4 Networks of Isomorphic Processes 97
3.4.1 A Generic Template 101

4 Conclusion and Future Work **109**

Bibliography **113**

List of Figures

2.1 PR Processes 15
2.2 A Template Process Example 25
2.3 Procedure $\mathcal{PROC}$ 27
2.4 template process P 29
2.5 Weak Bisimilarity of Rings 34
2.6 segment of size N $(\mathcal{S}_N)$ 45
2.7 Process T_{max} 46
2.8 template T 48
2.9 Con-discon example 53
2.10 Processes P_ℓ and P_r 53
2.11 A ring segment of size $2m-2$ 54

3.1 Producer and Consumer with an infinite size Buffer 59
3.2 Producer and Consumer with a Buffer of size 3 59
3.3 Mutual Exclusion example 60
3.4 different Petri nets with the same coverability tree 64
3.5 Petri Net $\mathcal{P}$ 66
3.6 Coverability Tree of $\mathcal{P}$ 66
3.7 FSM corresponding to coverability tree of $\mathcal{P}$ 67
3.8 Mutual Exclusion example with assigned priority 70

3.9 Elevator System 70
3.10 One Dining Philosopher 72
3.11 A Ring of 5 Dining Philosophers 74
3.12 Colored Petri Net Model of the Dining Philosophers 75
3.13 Petri nets in Chomsky Hierarchy 76
3.14 Büchi net construction steps 79
3.15 Petri net model of a factory 86
3.16 Token Generator in Petri Net $\mathcal{P}_e$ 89
3.17 Template Process of Network $\mathcal{N}$ 90
3.18 Petri Net Model of Network $\mathcal{N}$ 91
3.19 Petri Net Model of Network $\mathcal{N}$ with a Distinguished Process 92
3.20 Template Process $\mathcal{T}$ 93

Chapter 1

Introduction

Digital technology continues to create new opportunities and challenges for control engineers. Inexpensive computing power has made computer control almost ubiquitous, to the point where it is commonplace in everyday household appliances – let alone in disk drives, automobiles, aircraft and chemical plants. The abundance of computing power has been coupled with an explosion in digital communications and internet infrastructure, and this combination has produced automated systems of unprecedented scale and complexity.

To meet this challenge, control scientists have been developing appropriate new control paradigms since the early eighties. The field of control of discrete event systems combines the control engineer's outlook and methodology with models borrowed from computer engineering and computer science. Featuring abrupt, event-driven transitions among discrete states, such "discrete-event" models are more appropriate for high-level coordination problems that arise in the control of complex systems.

Thus far, the models employed have primarily been unstructured finite automata. Given intelligent means of problem decomposition such as those developed within discrete-event control, such models represent a useful means of synthesis and analysis of specific control systems. However, their unstructured nature often obscures the logical essence of a prob-

lem. Consider, for example, the control of a manufacturing system, where buffer overflows must be avoided. The appropriate control logic should in essence be independent of specific buffer capacities, yet current standard approaches provide no means of abstracting away such details. An alternative approach would consider "parameterized" models – these, in fact, are families of models, each member corresponding to different values of key system parameters (such as buffer capacities). The overall objective of this research is to extend methods for the analysis and synthesis of control logic to these more structured models.

In particular, the thesis will focus on parameterized networks of discrete-event systems. These are collections of interacting, isomorphic subsystems, where the number of subsystems is, for practical purposes, arbitrary; thus, the system parameter of interest is, in this case, the size of the network as characterized by the number of subsystems. Parameterized networks are reasonable models of real systems where the number of subsystems is large, unknown, or time-varying: examples include communication, computer and transportation networks. Specific motivation for the research will be provided by prior studies of the development of call-processing services in telecommunications networks within the formal framework of discrete-event control [2]. A key stumbling block in the development of such services is "feature interaction" – the unforeseen and undesirable interaction of different call-processing features or services. Within the discrete-event control framework, such interaction manifests itself as a form of "blocking" – the prevention of one subsystem from ever reaching a set of prespecified "goal" states.

Intuition and engineering practice suggest that, in checking for blocking or other properties, it should be sufficient to consider a "testbed" network of limited size. However, there is presently little rigorous support for such an approach. In general, the problem of deciding whether blocking occurs in parameterized networks of finite-state systems is undecidable [20]; and the only decidable subproblems that have so far been identified place unreasonable restrictions on the means by which subsystems may interact. The key to ensuring decidability, and therefore the existence of effective solutions to the problem, is

to identify restrictions that limit the computational power of the network. This can be done not only by limiting communication but also by restricting the structure of individual subsystems. In this thesis, we take both approaches, and also their combination on two types of network topologies: ring networks and fully connected networks.

The general methodology for ring networks will be to compare networks of different size, using process-algebraic equivalence relations. This methodology provides semidecision procedures for establishing that all networks are equivalent to networks of bounded size. We shall extend these results by identifying restrictions on subsystem structure that yield decision procedures. We have showed a decidability result for the case where subsystems have the form of so-called "piecewise" automata, and the equivalence relation employed is weak trace equivalence. The key idea is roughly that the number of equivalence classes of networks of arbitrary size can be uniformly bounded on the basis of the piecewise structure alone. This result will be extended to other equivalence relations, such as weak failures equivalence, weak possible-futures equivalence (which suffices to handle blocking). Another decidability result for ring networks is shown when a ring segment is a "shuffled" process; this guarantees the ring segments of arbitrary size to fall into a finite number of bisimulation classes. Then, a specific framework has been presented which limits the communication to a limited number, say m, of processes, and results in a ring segment of size $2m - 2$ to be a shuffled process. This framework enforces restriction on both communication and structure of processes in the ring.

Fully connected networks are investigated in two categories:

1. fully connected networks of identical processes in which processes are the exact same copies of a network template, and cannot distinguish between one another;

2. fully connected networks of isomorphic processes in which every process is obtained from the network template by appropriate relabeling of actions; has a distinguished identity, and can precisely determine the process that he intends to communicate

with.

Petri nets are introduced as a mathematical tool to model infinite-state and parameterized systems. More specifically, we will show how Petri nets can be used to model fully connected networks of identical processes. A linear temporal logic $\mathcal{L}$ on Petri nets is introduced, and the already-known decidable fragments of this logic are discussed. Furthermore, we define two fragments of this logic: $\mathcal{L}^{\mathcal{E}}$ and $\mathcal{L}^{\mathcal{O}}$, and show that the problem of deciding whether a given Petri net existentially (globally) satisfies a formula of $\mathcal{L}^{\mathcal{E}}$ ($\mathcal{L}^{\mathcal{O}}$) is decidable. Component and network blocking as some specific properties of our interest are investigated on fully connected networks of identical processes. It is proved that the problem of checking blocking for networks with a general broadcast template is undecidable. Such templates allow for communication among processes by means of rendezvous and broadcast actions; however, if templates are restricted to only allow for rendezvous actions, the problem becomes decidable. This implies a restriction on the means of communication, and not the structure of network processes.

Finally, fully connected networks of isomorphic processes are introduced. It is proved that blocking problem for such networks is undecidable even if the template process only allows for rendezvous actions. A general template is then proposed which limits the total number of processes communicating at any time. This template is expressive enough to model many real-life networks, and at the same time makes the model-checking problem for CTL$^*\backslash$X, and therefore, blocking problem decidable. In fact, it implies restriction on both communication and structure of network processes.

The results of the research will permit rigorous approaches to the analysis and design of complex, distributed control systems, and should enable further progress toward the goal of addressing control of parameterized systems in a more general context.

This thesis is organized as follows: In the following section, we will survey some of the work done in the literature which is similar in nature to our work.

In chapter 2, ring networks consisting of an arbitrary number of processes are in-

vestigated. Semialgorithms are introduced to automatically model-check such networks. Sufficient conditions are then presented on ring templates which guarantee the termination of the proposed semialgorithms. Chapter 3 discusses Petri nets as a model for infinite-state systems, and parameterized systems. An expressive linear temporal logic on Petri nets is defined, and the decidability of a large fragment of this logic is proven. Some applications of such results on infinite-state manufacturing systems, and fully-connected networks consisting of an arbitrary number of identical processes are then discussed. Fully connected networks of isomorphic processes are investigated next in this chapter. Blocking as a special property of our interest is discussed in both chapters 2,3 on ring networks and fully connected networks.

Finally in chapter 4, conclusions are given and future work is discussed.

1.1 Related Work

As mentioned before, the abundance of networks which consist of an essentially arbitrary number of isomorphic subsystems, and the criticality of safety and security in many such networks makes the parameterized model checking problem (PMCP) vitally important. This problem is known to be undecidable in general [42].

In an early work [10], Clarke, Grumberg and Browne defined a new logic called indexed CTL$^*\backslash$X to express the properties of networks consisting of arbitrary number of identical processes. In such networks, a natural number is assigned to every process by which all the atomic propositions of that process are subscripted. The formula $\bigwedge_i \phi(i)$ is then used to imply that $\phi(i)$ holds for every process P_i in the network, and similarly, $\bigvee_i \phi(i)$ is used to imply at least one process P_i satisfies $\phi(i)$ where $\phi(i)$ is a CTL^* formula in which the X operator is not allowed, and whose atomic propositions are all subscripted by i. A new version of bisimulation equivalence between two processes is also introduced which guarantees that both processes satisfy the same formulas of indexed CTL$^*\backslash$X. For

a distributed mutual exclusion example, it is shown that a network N_n of size $n \geq 2$ is equivalent with a network N_2 of size two. The properties of a network of arbitrary size can therefore be checked on N_2. This technique may be applied in other examples, but the main problem with this method is that the construction of the equivalence relation must be done by ad hoc means. The method proposed in [11] is a more systematic way of constructing the correspondences between the processes. A form of induction is used on the number of processes in the network. Since this version of bisimulation equivalence is not a congruence for the composition of processes, this requires ad hoc introduction of a new process that serves as a process closure. Therefore, this method again needs human intervention, and cannot be done automatically.

New induction methods are presented in [12, 13]. According to these methods, the specification is modelled as a further process, and the satisfaction relation is that of "implementation" of the specification by the network [12], or some other preorder relation such as language containment[13]. Sometimes we need to replace the specification model with a logically-stronger "network invariant". These invariants are typically constructed in an ad hoc manner.

Significant related studies have been reported that consider systems with special topologies. In [43] asynchronous networks consisting of classes of homogenous processes are investigated. The processes of a class are considered to be instances of a template. The overall network can be represented by $(U_1, ..., U_k)^{n_1,...,n_k}$ where U_i denotes the template of class i, and n_i denotes the number of instances in that class. So the processes in this class are numbered $1, 2, \cdots, n_i$. The size of this system depends on the number of instances in each class – one may think of n_1 to n_k as the parameters of this system. The transitions of a template process are of the form $s \xrightarrow{g} t$, indicating that the process can go from state s to state t provided that the guard g is true. The guards must have exactly one of the following two forms:

1. Disjunctive guards for the i^{th} instance of template l have the general form

$$\bigvee_{r\neq i}(a_l^r \vee ... \vee b_l^r) \vee \bigvee_{j\neq l}(\bigvee_{k\in[1..n_j]} (a_j^k \vee ... \vee b_j^k))$$

where a_i^j is true when instance j of template U_i is in local state a_i.

2. Conjunctive guards for the i^{th} instance of template l have the general form

$$\bigwedge_{r\neq i}(i_l^r \vee a_l^r \vee ... \vee b_l^r) \wedge \bigwedge_{j\neq l}(\bigwedge_{k\in[1..n_j]} (i_j^k \vee a_j^k \vee ... \vee b_j^k))$$

where i_l denotes the initial state of template l.

On the other hand, the correctness properties are expressed using a fragment of indexed CTL$^*\backslash$X. They are of one of the following three forms:

1. the properties over all the processes of a single class U_l:

   $\bigwedge_{i_l} Ah(i_l)$ and $\bigwedge_{i_l} Eh(i_l)$ where i_l ranges over the indices of U_l.

2. the properties over pairs of different processes in a single class U_l:

   $\bigwedge_{i_l,j_l} Ah(i_l,j_l)$ and $\bigwedge_{i_l,j_l} Eh(i_l,j_l)$ where i_l, j_l ranges over all the indices of U_l.

3. the properties over pairs of processes from two distinct classes U_l,U_m:

   $\bigwedge_{i_l,j_m} Ah(i_l,j_m)$ and $\bigwedge_{i_l,j_m} Eh(i_l,j_m)$ where i_l ranges over all the indices of U_l and j_m ranges over all the indices of U_m.

where h is an LTL$\backslash$X formula.

Various examples of networks can be modelled in the framework of [43]. For such systems a bound on the number of instances of each class has been found which is sufficient to capture all the possible computations in the parameterized system.

For the network $(U_1, ..., U_k)^{n_1,...,n_k}$, two cutoffs $(c_1, c_2 \cdots c_k)$ and $(d_1, d_2 \cdots d_k)$ are defined such that $c_i = |U_i| + 3$, $d_i = 2|U_i| + 1$ and $|U_i|$ is the number of states of template U_i. It is

then shown that they are sufficient cutoffs for checking all three types of properties of the networks defined in disjunctive and conjunctive forms respectively. For example, in order to show that all the instances of a given network $(U_1, ..., U_k)^{n_1,...,n_k}$(defined in disjunctive framework for instance) satisfy a given property ψ (one of the three types), it suffices to check ψ for those instances which we have $n_i \leq c_i$ for $1 \leq i \leq k$. This can also be denoted by $(n_1, ..., n_k) \leq (c_1, c_2 \cdots c_k)$. This is a significant result in parameterized model checking.

In [14] two different types of network structures are investigated. First, networks consisting of a unique control process, and an arbitrary number of identical processes. The processes communicate using CCS actions. A decision algorithm for checking whether all the computations of such a network satisfy a given temporal property is presented. The second one is a special case of the first when the network does not have a distinguished control process. It is shown that a more efficient algorithm can be applied for such networks. This framework is too restrictive, and is not appropriate for modelling a complicated network such as a telephone network because processes cannot distinguish among one another, and a transition of one process can be synchronized with any other process capable of executing its complement transition.

In [17], networks with a ring topology have been investigated. The processes are all copies of a template process T. So a ring of size n consists of processes $P_1, P_2, \cdots, P_{n-1}$ where P_i is obtained from T by indexing all its propositions and actions with i. P_{i+1} is the immediate right-hand neighbor of P_i, and P_{i-1} is the immediate left-hand neighbor of P_i where the index calculations are modulo n. Every process in the ring can communicate with its immediate neighbors, and this communication is done by means of a single token which is passed between neighbor processes. Every process that has the token will eventually pass it to its right neighbor. Initially, the token is given to one of the processes nondeterministically. The token cannot carry any information with it. Processes are assumed to have two sets of actions: free actions, and token-dependent actions. A process

cannot perform a token dependent action unless it has the single token of the ring, but free actions can be performed at any time. For such a system, it has been shown that the problem of checking many correctness properties for every size instance of the ring can be reduced to checking them on rings of bounded size where the bound (cutoff) depends on the kind of property for which we check the system. The properties are expressed using indexed CTL$^*\backslash$X. An example of an acceptable formula is $\forall i :: g(i)$ where i ranges over all the process indices, and $g(i)$ is a CTL$^*\backslash$X formula expressed on the propositions and actions of process i. A given ring network satisfies this property iff every process P_i in the ring satisfies $g(i)$. Another example is $\forall i, j : i \neq j : g(i, j)$; this property holds iff every distinct pair of processes P_i, P_j satisfies $g(i, j)$. The following results have been proved for four different types of properties:

- properties of the form $\forall i :: g(i)$ have a cutoff of 2.
- properties of the form $\forall i :: g(i, i + 1)$ have a cutoff of 3.
- properties of the form $\forall i, j : i \neq j : g(i, j)$ have a cutoff of 4.
- properties of the form $\forall i, j : i \neq j : g(i, i + 1, j)$ have a cutoff of 5.

Using similar techniques as in [17], these results can be extended to other types of properties. The major restriction in this work is on the communication of processes. As said before, the token does not carry any information; if it did so, the problem would become undecidable.

The authors of [19] generalize the work of [17] to other network topologies consisting of isomorphic processes which communicate by passing a single token among them. A directed graph G(V,E), where $V = \{1, 2, \cdots, |V|\}$ and $E \subseteq V \times V$, is defined to represent the way the processes can transfer the unique token. Each vertex i corresponds to a process P_i in the network, and a directed edge (i, j) means that process P_i can transfer its token to process P_j. Therefore a network of n processes can be represented by a network graph

$G = (V, E)$ such that $|V| = n$, and a template process T. The processes are copies of T obtained by appropriate renaming. There are two major differences between [19] and [17]. First, the results only hold for LTL\X properties, and second, a more refined definition of cutoff is used.

Consider the quantifier-free LTL\X formula $\phi(x, y)$ where x, y are two indexed variables. Then the closed formula $\forall x \exists y.\phi(x, y)$ known as a *2-indexed* specification implies "for every process P_i in the network, there exits another process P_j such that $\phi(i, j)$ holds". In a similar way, a *k-indexed* specification is defined as a formula whose quantifier-free part, also known as its *matrix*, refers to k processes.

Assume that we want to model-check a parameterized network N for a property ψ; we reduce the problem as follows: model check c smaller networks of size less than or equal to s. The final result of the verification on the original network is a boolean expression on the collected results on smaller ones. This is called a (c, s)-bounded reduction. It is shown that for a class of networks and a k-indexed specification, a (c, s)-bounded reduction exists such that c, s only depend on k. In particular, for a 2-indexed specification, it is enough to model check at most 36 networks of size 4. Although the results of [19] is more general than [17], the communication among processes is still restricted to a great extent.

In [18], Emerson and Kahlon address the state explosion in the context of resource allocation systems. In such systems the processes share tokens representing their common resources. Each token is shared between at most two processes, and is possessed by one of them, or none at a time (free resource). Every process needs to possess all its shared tokens to perform the token dependent actions. A method is proposed in which the model checking of a network comprising a fixed number n of possibly heterogeneous processes is reduced to the model checking of a smaller system. The smaller system is constructed with respect to the original system and the correctness property to be verified (expressed in LTL\X). In the special case when the network is symmetric and the processes are homogenous, this method can be extended to the model checking of parameterized networks where the

number of processes in the network is arbitrary. The dining philosophers problem is studied as an application of this method.

Chapter 2

Ring Networks of Isomorphic Processes

In this chapter, we investigate ring networks consisting of isomorphic processes. Processes in the network are obtained by appropriate relabelling of a template process. They interact through event-sharing. Furthermore, the number of processes in the network is assumed to be arbitrary. This is an instance of parameterized model-checking problem. In fact, the number of processes is the only parameter of such networks. The model-checking of a ring network is then to check whether a given property holds for every fixed size ring. This problem is known to be undecidable [17, 42]. However, one may achieve decidability results by imposing restrictions on the structure of the template process, or the mechanism whereby processes interact with one another. In this work, we apply restrictions on the structure of subsystems rather than the means of interaction. Within our framework, semialgorithms have been proposed, based on various process equivalences. We show that a procedure based on *weak trace equivalence* is guaranteed to terminate in the case where individual subsystems are *piecewise recognizable*; this can be extended to a procedure based on the finer *completed trace equivalence*, *weak failure equivalence* or *weak possible futures*

equivalence. Consequently, a subset of *observable modal logic* (which is preserved under the above equivalence relations) as well as LTL\X properties can be model-checked for such systems.

We then consider networks in which actions occurring in a subsystem affect only a bounded number of other subsystems. This property is formalized through the notion of *shuffled processes*. When a segment of a ring is a shuffled process, it is shown that networks of all sizes fall into a finite number of weak bisimilarity classes. In this case, the whole set of observable modal logic and also CTL*\X properties can be model-checked.

We also show how such parameterized networks can be checked for *deadlock* and *blocking* of their subsystems whenever the proposed procedures terminate.

This chapter is organized as follows. In the first section, we discuss some preliminaries of process algebra. A process model is defined, and several operations on processes such as synchronous product, hiding, and renaming are explained. We also introduce a modal logic to express the properties of processes. Several equivalence relations are defined, and the classes of modal formulas preserved under the respective equivalence relations are discussed. In section 2.2, the computation model of a ring network is defined, and a general semialgorithm is introduced to check such networks against modal properties. It is also shown how the results can be extended to temporal logic. However, the proposed semialgorithm is not guaranteed to terminate. It is shown in section 2.3 that the termination of this semialgorithm is undecidable for every equivalence relation which is finer than weak trace equivalence and coarser than weak bisimulation. Finally in section 2.4, we introduce some sufficient conditions on the structure of rings templates which guarantees the termination of the proposed semialgorithm for some equivalence relations.

2.1 Preliminaries

2.1.1 Process Model

A finite state process P is a tuple of the form (S, Σ, R, s_0) where S is a finite set of reachable states, Σ is a finite set of visible actions, $R \subseteq S \times (\Sigma \uplus \{\tau\}) \times S$ is a transition relation[1], and $s_0 \in S$ is the initial state. Sometimes we write $s \xrightarrow{\sigma} r$ to show $(s, \sigma, r) \in R$. In a given transition $t = (s, \sigma, r)$, s is known as its source state ($Src(t)$), r as the destination state ($Dst(t)$), and σ as its action ($Act(t)$).

For a given sequence of transitions $\overline{t} = t_1 t_2 \cdots t_n$, if $Src(t_1) = s$, $Dst(t_n) = r$, and $Dst(t_i) = Src(t_{i+1})$ for $1 \leq i \leq n-1$, we say s leads to r via $\overline{t}$ and denote this by $s \overset{\overline{t}}{\rightsquigarrow} r$. The sequence $\overline{t}$ is called a computation path from s to r. The reachable state of $\overline{t}$, denoted by $reach(\overline{t})$, is the destination state r of its last transition t_n. The definition of $reach$ can be extended to sets of computation paths as follows: $reach(A) = \{reach(\overline{t}) : \overline{t} \in A\}$. We also denote by $Act(\overline{t})$ the sequence of actions of transitions in $\overline{t}$: $Act(t_1)Act(t_2)\cdots Act(t_n)$. The projection of $Act(\overline{t})$ onto the visible actions is called $\overline{t}$'s sequence of visible actions, and is denoted by $Act_v(\overline{t})$. The set of all the computation paths of P starting from s_0 is denoted by $\mathcal{C}(P)$.

The notation $s \overset{\epsilon}{\Longrightarrow} r$ means that there exists a sequence of states $u_1 u_2 \cdots u_n$ such that $u_1 = s$ and $u_n = r$ and $u_i \xrightarrow{\tau} u_{i+1}$ for $1 \leq i \leq n-1$. As a special case, $s \overset{\epsilon}{\Longrightarrow} s$ for every state $s \in S$. We also use the notation $s \overset{\sigma}{\Longrightarrow} r$ for a visible action σ when there exist states v and w such that $s \overset{\epsilon}{\Longrightarrow} v$, $v \xrightarrow{\sigma} w$, and $w \overset{\epsilon}{\Longrightarrow} r$. This can easily be generalized to strings of visible actions. Given a string $\overline{\sigma} = \sigma_1 \sigma_2 \cdots \sigma_n$ where $\sigma_i \in \Sigma$, the notation $s \overset{\overline{\sigma}}{\Longrightarrow} r$ implies that there exists a sequence of states $u_1 u_2 \cdots u_{n+1}$ such that $u_1 = s$ and $u_{n+1} = r$ and $u_i \overset{\sigma_i}{\Longrightarrow} u_{i+1}$ for $1 \leq i \leq n$. Sometimes we just write $s \overset{\overline{\sigma}}{\Longrightarrow}$ to show that there exists some state r such that $s \overset{\overline{\sigma}}{\Longrightarrow} r$. Define the visible language of a process P as the set of all the strings executable from its initial state: $L_v(P) = \{\sigma \in \Sigma^* : s_0 \overset{\sigma}{\Rightarrow}\}$.

[1] τ is the invisible action.

Piecewise Recognizable Processes

Process P is called *piecewise recognizable* (PR) iff there exists a partial order relation $\leq$ on S such that if $(s_1, \sigma, s_2) \in R$, then $s_1 \leq s_2$ [7]. An example of a PR process is a program whose variables have all finite domains, and the values assigned to any of them, during a run of that program, is non-decreasing or non-increasing. A *thin piecewise* process is a PR process whose states can have at most one successor state other than itself. A *tree* process is a special PR process in which each state has at most one predecessor other than itself. A *leaf* state is one which does not have any successor. Figure 2.1 depicts some examples of PR processes. All three processes in the Figure are PR; the one on the left is thin piecewise and the one in the middle is a tree.

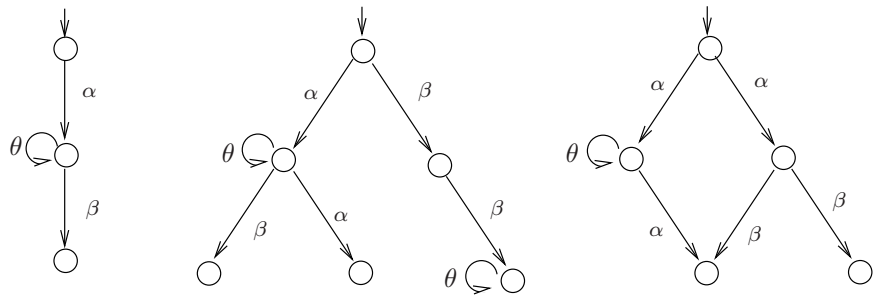


Figure 2.1: PR Processes

Given a tree process $P = (S, \Sigma, R, s_0)$, a subtree of P is a process (S', Σ, R', s_0) where $S' \subseteq S$, $R' \subseteq R$, and if $(s, \sigma, s) \in R$ for some $s \in S'$, then $(s, \sigma, s) \in R'$. Note that a subtree of P has the same initial state as P. Denote the set of all subtrees of P by $sub(P)$. One way of constructing a subtree of P is to remove a subset of its non-self-loop transitions, and then trim the obtained process by removing all unreachable states. Therefore, if P has m non-self-loop transitions, there are 2^m different possibilities for subtrees of P, some of which become the same after trimming; 2^m is, therefore, only an upper bound on the number of such subtrees: $|sub(P)| < 2^m$.

Hiding and Renaming

Let σ be a visible string in Σ^*. We denote by $\sigma\backslash\Sigma_1$ string σ with all the actions in Σ_1 deleted (hidden). This operation can be defined inductively as follows:

$$\epsilon\backslash\Sigma_1 = \epsilon$$

$$\alpha\backslash\Sigma_1 = \begin{cases} \epsilon & : \quad \alpha \in \Sigma_1 \\ \alpha & : \quad \alpha \notin \Sigma_1 \end{cases}$$

$$\sigma\alpha\backslash\Sigma_1 = (\sigma\backslash\Sigma_1)(\alpha\backslash\Sigma_1)$$

where $\alpha \in \Sigma$ and $\sigma \in \Sigma^*$. Similarly, $\sigma\backslash\overline{\Sigma}_1$ is defined as σ with all the actions not in Σ_1 deleted. For a language $L \subseteq \Sigma^*$, define $L\backslash\Sigma_1 = \{\sigma\backslash\Sigma_1 : \sigma \in L\}$ and $L\backslash\overline{\Sigma}_1 = \{\sigma\backslash\overline{\Sigma}_1 : \sigma \in L\}$.

We can also extend the notion of action hiding to processes which is a standard operation in Milner's process algebra [5]. We denote by $P\backslash\Sigma_1$, process P with all his actions in Σ_1 hidden. More formally, for a given process $P = (S, \Sigma, R, s_0)$, $P\backslash\Sigma_1$ is defined as process P in which the action $Act(t)$ of every transition $t \in R$ is renamed to τ if $Act(t) \in \Sigma_1$. Similarly, process $P\backslash\overline{\Sigma_1}$ is defined as P with all the actions not in Σ_1 hidden.

Proposition 1 *For a given process $P = (S, \Sigma, R, s_0)$ and a set of actions Σ_1, we have $L_v(P)\backslash\Sigma_1 = L_v(P\backslash\Sigma_1)$.*

Renaming, another standard operation of Milner's process algebra [5], is a more general form of hiding where every action of a process is renamed (not necessarily to τ) according to a given function. Given a process $P = (S, \Sigma, R, s_0)$ and a function $f : \Sigma \longrightarrow \Sigma'$, define $P[f]$ to be the tuple (S, Σ', R_1, s_0) where $R_1 = \{(s_1, \beta, s_2) | (s_1, \alpha, s_2) \in R \ \& \ \beta = f(\alpha)\}$. If f is not defined for some action, then that action won't be renamed.

Interleaving of Strings

Given any two strings of actions (transitions) x, y, define $inter(x, y)$ as the set of all strings obtained by interleaving the actions (transitions) of x and y [6]. For instance,

$$inter(\beta_1\beta_2, \alpha_1\alpha_2) = \{\beta_1\beta_2\alpha_1\alpha_2, \beta_1\alpha_1\beta_2\alpha_2, \beta_1\alpha_1\alpha_2\beta_2, \alpha_1\beta_1\beta_2\alpha_2, \alpha_1\beta_1\alpha_2\beta_2, \alpha_1\alpha_2\beta_1\beta_2\}.$$

The interleaving of two sets of strings is defined as the set of all interleavings of all pairs of strings belonging to those sets:

$$inter(A, B) = \bigcup_{x \in A, y \in B} inter(x, y)$$

Synchronous Product of Processes

Given two processes $P_1 = (S_1, \Sigma_1, R_1, s_{01})$, $P_2 = (S_2, \Sigma_2, R_2, s_{02})$, define their synchronous product $P_1 \times P_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, R, (s_{01}, s_{02}))$ where $((s_1, s_2), \alpha, (r_1, r_2)) \in R$ iff one of the following conditions holds:

- $\alpha \in \Sigma_1 \cap \Sigma_2, (s_1, \alpha, r_1) \in R_1, (s_2, \alpha, r_2) \in R_2$;
- $\alpha \in (\Sigma_1 \setminus \Sigma_2) \cup \{\tau\}, (s_1, \alpha, r_1) \in R_1, s_2 = r_2$;
- $\alpha \in (\Sigma_2 \setminus \Sigma_1) \cup \{\tau\}, (s_2, \alpha, r_2) \in R_2, s_1 = r_1$.

In the special case when $\Sigma_1 \cap \Sigma_2 = \emptyset$ the synchronous product of P_1 and P_2 is known as their shuffle product and is denoted by $P_1 \circledcirc P_2$.

Proposition 2 *Given two processes $P_1 = (S_1, \Sigma_1, R_1, s_{01})$ and $P_2 = (S_2, \Sigma_2, R_2, s_{02})$, the following statements hold:*

- $L_v(P_1 \times P_2)\backslash\overline{\Sigma}_1 \subseteq L_v(P_1)$
- $L_v(P_1 \circledcirc P_2)\backslash\overline{\Sigma}_1 = L_v(P_1)$
- $L_v(P_1 \times P_2)\backslash\overline{\Sigma} \subseteq L_v(P_1)\backslash\overline{\Sigma} \cap L_v(P_2)\backslash\overline{\Sigma}$

2.1.2 Observable Modal Logic

We use the observable modal logic M^o from [5] to express the properties of the processes. The formulas of M^o are inductively defined as follows:

$$\Psi := \mathbb{T} \mid \mathbb{F} \mid \Psi_1 \wedge \Psi_2 \mid \Psi_1 \vee \Psi_2 \mid \langle\langle K \rangle\rangle \Psi \mid \langle\langle\rangle\rangle \Psi \mid [[K]]\Psi \mid [[\ ]]\Psi$$

where K is a subset of visible actions.

For a given process $P = (S, \Sigma, R, s_0)$, state $s \in S$, and modal formula Ψ, we write $(P, s) \models \Psi$ to show that state s of P satisfies Ψ, and if that does not hold $(P, s) \not\models \Psi$. The semantics of this satisfaction relation is defined inductively on the structure of the formulas as follows:

$$\begin{array}{lll}
(P,s) \models \mathbb{T} & & \\
(P,s) \not\models \mathbb{F} & & \\
(P,s) \models \Psi_1 \wedge \Psi_2 & \text{iff} & (P,s) \models \Psi_1 \text{ and } (P,s) \models \Psi_2 \\
(P,s) \models \Psi_1 \vee \Psi_2 & \text{iff} & (P,s) \models \Psi_1 \text{ or } (P,s) \models \Psi_2 \\
(P,s) \models \langle\langle K \rangle\rangle \Psi & \text{iff} & \exists t \in \mathcal{R}ch(s,K).\ (P,t) \models \Psi \\
(P,s) \models \langle\langle\rangle\rangle \Psi & \text{iff} & \exists t \in \mathcal{R}ch(s,\epsilon).\ (P,t) \models \Psi \\
(P,s) \models [[K]]\Psi & \text{iff} & \forall t \in \mathcal{R}ch(s,K).\ (P,t) \models \Psi \\
(P,s) \models [[\ ]]\Psi & \text{iff} & \forall t \in \mathcal{R}ch(s,\epsilon).\ (P,t) \models \Psi
\end{array}$$

where $\mathcal{R}ch(s,K)$, $\mathcal{R}ch(s,\epsilon)$ denote the set of states reachable from s by performing an action in K or ϵ respectively; more formally, $\mathcal{R}ch(s,K) := \{r \in S : s \stackrel{\alpha}{\Longrightarrow} r, \alpha \in K\}$, $\mathcal{R}ch(s,\epsilon) := \{r \in S : s \stackrel{\epsilon}{\Longrightarrow} r\}$.

Proposition $\mathbb{T}$ holds in every state s of a process P, and $\mathbb{F}$ does not hold in any state. The formula $\Psi_1 \wedge \Psi_2$ ($\Psi_1 \vee \Psi_2$) holds in state s if Ψ_1 and (or) Ψ_2 holds in s. The formula $\langle\langle K \rangle\rangle \Psi$ holds in s if P can perform an action in K from state s, and reach a state which satisfies Ψ. Similarly, $\langle\langle\rangle\rangle \Psi$ holds in s iff P can reach another state from s which satisfies

Ψ by only taking invisible actions. The formula $[[K]]\Psi$ holds in state s if all the actions in K from s evolve to states which satisfy Ψ. Similarly, $[[\ ]]\Psi$ holds in state s if all the reachable states from s by performing invisible strings of actions satisfy Ψ.

We say process P satisfies an observable modal formula Ψ, and write $P \models \Psi$ if $(P, s_0) \models \Psi$. For notation simplicity, we use $[[\alpha_1, \alpha_2, \cdots \alpha_n]]$ instead of $[[\{\alpha_1, \alpha_2, \cdots \alpha_n\}]]$, and $\langle\langle \alpha_1, \alpha_2, \cdots \alpha_n \rangle\rangle$ instead of $\langle\langle \{\alpha_1, \alpha_2, \cdots \alpha_n\} \rangle\rangle$. This logic can be used to express local capabilities and necessities of processes [5]. For instance, $P \models [[\alpha_1, \alpha_2]]\langle\langle \beta \rangle\rangle T$ implies that the visible action β can be performed from any state reachable from the initial state of P by performing an visible action α_1 or α_2.

2.1.3 Process Equivalences

When do we call two processes equivalent? This is a fundamental question in process algebra. Two processes are said to be equivalent when they both show the same "behavior". However, their behavior can be seen from different perspectives. Let $\mathcal{P}$ be the set of all processes. Each equivalence relation partitions $\mathcal{P}$ into classes of processes with equivalent behaviors. Equivalence relation E_1 is said to be more refined than E_2 ($E_1 \leq E_2$) if any two processes which are equal according to E_1 are also equal according to E_2.

$$E_1 \leq E_2 \quad \text{iff} \quad \forall P_1, P_2 \in \mathcal{P}.\ [(P_1, P_2) \in E_1 \Rightarrow (P_1, P_2) \in E_2]$$

One way of defining an equivalence relation on $\mathcal{P}$ is by presenting a set of properties Γ. Then, we say two processes P_1 and P_2 are equivalent with respect to Γ, and denote it by $P_1 \equiv_\Gamma P_2$, iff they both satisfy the same properties from Γ. In other words, $\equiv_\Gamma$ preserves the set of properties indicated by Γ. An extreme case is when $\Gamma = \emptyset$ which results in the equivalence of any two processes. As the set of properties, one may choose a subset of formulas of the modal logic M^o. In the remainder of this subsection, a number of important equivalence relations will be defined, and their respective preserved subsets

of M^o are stated. We will also investigate whether they can be ordered according to the partial order relation $\leq$ [5].

Weak Trace Equivalence

We say two processes P_1, P_2 are *weakly trace equivalent*, and denote it by $P_1 \equiv_{wtr} P_2$, iff they both have the same visible languages [5].

$$P_1 \equiv_{wtr} P_2 \quad \text{iff } L_v(P_1) = L_v(P_2)$$

Proposition 3 *Let Γ_{wtr} be the set of all modal formulas of the form $\langle\langle\alpha_1\rangle\rangle\langle\langle\alpha_2\rangle\rangle \cdots \langle\langle\alpha_n\rangle\rangle\mathbb{T}$ where $\alpha_i \in \Sigma$.* [2] *Then, the equivalence relations $\equiv_{wtr}$ and $\equiv_{\Gamma_{wtr}}$ are identical.*

String $\sigma \in \Sigma^*$ is called a *deadlock* for process $P = (S, \Sigma, R, s_0)$ iff there exists some state $s \in S$ such that $s_0 \stackrel{\sigma}{\Longrightarrow} s$, and no visible action can be performed from s. Let's denote the set of all deadlocks of P by $D(P)$. *Completed trace equivalence* is a finer version of weakly trace equivalence which guarantees the processes to have the same set of deadlocks as well [5]:

$$P_1 \equiv_{ctr} P_2 \quad \text{iff } L_v(P_1) = L_v(P_2) \ \&\ D(P_1) = D(P_2)$$

Proposition 4 *Let Γ_{ctr} be the union of Γ_{wtr} and the set of all modal formulas of the form $\langle\langle\alpha_1\rangle\rangle\langle\langle\alpha_2\rangle\rangle \cdots \langle\langle\alpha_n\rangle\rangle[[\Sigma]]\mathbb{F}$ or $\langle\langle\ \rangle\rangle[[\Sigma]]\mathbb{F}$ where $\alpha_i \in \Sigma$. Then, the equivalence relations $\equiv_{ctr}$ and $\equiv_{\Gamma_{ctr}}$ are identical.*

Weak Failure Equivalence

There may be a certain subset of visible actions X of a process P which cannot occur (be rejected) after a specific run of visible actions $\overline{\sigma}$; such visible actions are known as *rejections* of P after performing $\overline{\sigma}$, and the pair $(\overline{\sigma}, X)$ is known as a *failure* of P. More

[2] when not specified Σ is the set of all visible actions.

formally, for a given process $P = (S, \Sigma, R, s_0)$ and a state $s \in S$, define $\mathcal{R}jct_v(s)$ as the set of visible actions which cannot be executed from s:

$$Rjct_v(s) := \Sigma \setminus \{\alpha \in \Sigma \mid s \stackrel{\alpha}{\Longrightarrow}\}$$

Then, $Failures(P)$ is defined as follows:

$$Failures(P) = \{(\overline{\sigma}, X) \in \Sigma^* \times 2^{\Sigma} : \ \exists s \in S.\ s_0 \ \stackrel{\overline{\sigma}}{\Longrightarrow} s \ \ \& \ X \subseteq Rjct_v(s)\}$$

We say two processes P_1, P_2 are *weakly failure equivalent*, and denote it by $P_1 \equiv_{wf} P_2$, iff $Failures(P_1) = Failures(P_2)$.

Proposition 5 *Let Γ_{wf} be the union of Γ_{ctr} and the set of all modal formulas of the form $\langle\langle\alpha_1\rangle\rangle\langle\langle\alpha_2\rangle\rangle \cdots \langle\langle\alpha_n\rangle\rangle[[K]]\mathbb{F}$ where $\alpha_i \in \Sigma$ and $K \subseteq \Sigma$. Then, the equivalence relations $\equiv_{wf}$ and $\equiv_{\Gamma_{wf}}$ are identical.*

A string σ belongs to the visible language $L_v(P)$ of a process P iff $(\sigma, \emptyset) \in Failures(P)$. Similarly, a deadlock σ belongs to $D(P)$ iff $(\sigma, \Sigma) \in Failures(P)$. Therefore, when the $Failures$ sets of two processes are equal, so are their visible languages and deadlocks. Consequently $\equiv_{wf} \leq \equiv_{ctr}$.

The weak failure equivalence is specially of interest for deadlock detection. A process running in a network of processes is said to have *deadlock* if it can reach a state from which cannot perform any visible action.

Proposition 6 *Suppose E and F are weakly failure equivalent. Then, P shows deadlock in $P \times E$ iff it does in $P \times F$.*

Weak Possible-Futures Equivalence

Given a process $P = (S, \Sigma, R, s_0)$, the set of weakly possible-futures of P is defined as $PF(P) := \{(\overline{\sigma}, L) \in \Sigma^* \times 2^{\Sigma^*} : \exists s \in S. s_0 \stackrel{\overline{\sigma}}{\Longrightarrow} s \ \ \& \ \ L_v(P_s) = L\}$.

We say two processes P_1, P_2 are *weakly possible-futures equivalent*, and denote it by $P_1 \equiv_{wpf} P_2$, iff $PF(P_1) = PF(P_2)$.

Proposition 7 *Let* Γ_{wpf} *be the union of* Γ_{wf} *and the set of all formulas of the form* $\langle\langle a_1\rangle\rangle \langle\langle a_2\rangle\rangle \cdots \langle\langle a_n\rangle\rangle \left(\langle\langle \alpha_1\rangle\rangle\langle\langle \alpha_2\rangle\rangle \cdots \langle\langle \alpha_{k_1}\rangle\rangle\mathbb{T} \wedge \cdots \wedge \langle\langle \theta_1\rangle\rangle\langle\langle \theta_2\rangle\rangle \cdots \langle\langle \theta_{k_m}\rangle\rangle\mathbb{T}\right)$ *where* $a_i, \alpha_i, \cdots, \theta_i \in \Sigma$. *Then, the equivalence relations* $\equiv_{wpf}$ *and* $\equiv_{\Gamma_{wpf}}$ *are identical.*

It can easily be shown that $\equiv_{wpf} \leq \equiv_{wf}$ [5].

The weak possible-futures equivalence specially useful to detect blocking. Sometimes a subset of states $F \subseteq S$ of a process P is marked as final states. A process running in a network of processes is said to have *blocking* if it can reach a state from which a marker state is not reachable. This notion of blocking is the same as *component blocking* defined in [1]. It can also be expressed in CTL$^*\backslash$X logic as follows: let p_f be the proposition of being a marker state, then a process P running in a network of processes does not show component blocking iff P satisfies $AGEFp_f$.

Proposition 8 *Let* $P = (S, \Sigma, R, s_0)$, $E = (S_E, \Sigma, R_E, s_{0e})$, $F = (S_F, \Sigma, R_F, s_{0f})$ *and* $E \equiv_{wpf} F$; *then* P *is blocking in* $P \times E$ *iff it is blocking in* $P \times F$.

Proof: Suppose P is blocking in $P \times E$; then, there exists a blocking state $(s, s_e) \in S \times S_E$ and $x \in \Sigma^*$ such that $(s_0, s_{0e}) \stackrel{x}{\Longrightarrow} (s, s_e)$ in $P \times E$ and for every reachable state (s', s'_e) from (s, s_e), we have s' is not a final state. Let L be the language of all visible strings from s_e in process E. Therefore, the pair $(x, L) \in PF(E) = PF(F)$. This implies that there exists $s_f \in S_F$ such that $(s_0, s_{0f}) \stackrel{x}{\Longrightarrow} (s, s_f)$ in $P \times F$ and the language of visible strings from s_f is L. The global state (s, s_f) is blocking in $P \times F$ because F allows for occurrence of the same strings from s_f as E does from s_e. □

In section 2.2, it will be explained how propositions 6,8 can be used to detect partial deadlocks and blocking of ring networks with an arbitrary number of isomorphic processes.

Weak Bisimulation

Given two processes $P_1 = (S_1, \Sigma, R_1, s_{01})$, $P_2 = (S_2, \Sigma, R, s_{02})$, and a relation $Rel \subseteq S_1 \times S_2$, we say Rel is a *weak bisimulation relation* iff the following conditions hold:

- If $(s_1, r_1) \in Rel$ and $s_1 \stackrel{\alpha}{\Longrightarrow} s_2$ for some $\alpha \in \Sigma \cup \{\epsilon\}$, then there exists some state r_2 such that $r_1 \stackrel{\alpha}{\Longrightarrow} r_2$ and $(s_2, r_2) \in Rel$;

- If $(s_1, r_1) \in Rel$ and $r_1 \stackrel{\alpha}{\Longrightarrow} r_2$ for some $\alpha \in \Sigma \cup \{\epsilon\}$, then there exists some state s_2 such that $s_1 \stackrel{\alpha}{\Longrightarrow} s_2$ and $(s_2, r_2) \in Rel$.

For $s_1 \in S_1$ and $s_2 \in S_2$, we write $(P_1, s_1) \equiv_{wb} (P_2, s_2)$ to mean that a weak bisimulation relation Rel exists and $(s_1, s_2) \in Rel$. Two processes P_1 and P_2 are *weakly bisimilar*, $P_1 \equiv_{wb} P_2$, iff there exists a bismulation relation Rel such that $(P_1, s_{01}) \equiv_{wb} (P_2, s_{02})$ [5].

Weak bisimulation is the strongest of all the previous equivalences we have discussed so far, and preserves all formulas of M^o.

Proposition 9 *Let Γ_{wb} be the set of all modal formulas M^o. Then, the equivalence relations $\equiv_{wb}$ and $\equiv_{\Gamma_{wb}}$ are identical.*

Proof: First we show that for every two processes P_1, P_2 and every two states s_1, s_2 belonging to their sets of states respectively, if $(P_1, s_1) \equiv_{wb} (P_2, s_2)$, then they both satisfy the same formulas of M^o. This will be done by using induction on the structure of modal formulas Ψ. The base case when Ψ is $\mathbb{T}$ or $\mathbb{F}$ clearly holds. Now let Ψ be of the form $\Psi_1 \wedge \Psi_2$, and assume that the proposition holds for Ψ_1 and Ψ_2. We know $(P_1, s_1) \models \Psi$ iff $(P_1, s_1) \models \Psi_1$ and $(P_1, s_1) \models \Psi_2$ iff $(P_2, s_2) \models \Psi_1$ and $(P_2, s_2) \models \Psi_2$ iff $(P_2, s_2) \models \Psi$. The case when Ψ is of the form $\Psi_1 \vee \Psi_2$ can be shown in a similar way. Now let's say Ψ is $[[K]]\Psi_1$ and (P_1, s_1) satisfies Ψ, which means that for any state s_1' and any action $\alpha \in K$ such that $s_1 \stackrel{\alpha}{\Longrightarrow} s_1'$, we have $(P_1, s_1') \models \Psi_1$. Now for every state s_2' and every action $\alpha \in K$ such that $s_2 \stackrel{\alpha}{\Longrightarrow} s_2'$, there exists a state s_1' of process P_1 such that $s_1 \stackrel{\alpha}{\Longrightarrow} s_1'$ and $(P_1, s_1') \equiv_{wb} (P_2, s_2')$. We also have $(P_1, s_1') \models \Psi_1$ which implies $(P_2, s_2') \models \Psi_1$ according to the premise of the induction. Consequently, $(P_2, s_2) \models \Psi$. A similar reasoning can be presented for the other operators of M^o. Thus, when $P_1 \equiv_{wb} P_2$ both processes satisfy the same formulas of M^o.

Now suppose that $P_1 \equiv_{\Gamma_{wb}} P_2$. Let's define the relation Rel as follows:

$$Rel := \{(s,r) : \forall \Psi \in M^o, \ \ (P_1, s) \models \Psi \text{ iff } (P_2, r) \models \Psi\}$$

We shall first show that the relation Rel is a weak bisimulation relation.

Suppose that Rel is not a weak bisimulation relation. That implies that there exist a pair $(s,r) \in Rel$ and $\alpha \in \Sigma \cup \{\epsilon\}$ for which $s \stackrel{\alpha}{\Longrightarrow} s'$ in P_1, but a state r' does not exist such that $r \stackrel{\alpha}{\Longrightarrow} r'$ in P_2 and $(s', r') \in Rel$. This could happen either because $r \stackrel{\alpha}{\not\Longrightarrow}$, or because for every state r_i, $i \in \{1, 2, \cdots n\}$, such that $r \stackrel{\alpha}{\Longrightarrow} r_i$, we have $(P_1, s') \models \Psi_i$ and $(P_2, r_i) \not\models \Psi_i$. If the first case holds, then (P_1, s) satisfies $\langle\langle\alpha\rangle\rangle\mathbb{T}$, but (P_2, r) does not; therefore, (s,r) cannot belong to Rel which is a contradiction. In the second case, (P_1, s') satisfies $\Psi = \Psi_1 \wedge \Psi_1 \wedge \cdots \Psi_n$, but (P_2, r_i) does not. Consequently, $(P_1, s) \models \langle\langle\alpha\rangle\rangle\Psi$ and $(P_1, r) \not\models \langle\langle\alpha\rangle\rangle\Psi$; therefore, (s,r) cannot belong to Rel which is again a contradiction.

Also note that Rel relates the initial states s_{01}, s_{02} of P_1, P_2 since (P_1, s_{01}) and (P_2, s_{02}) both satisfy the same M_o formulas. Therefore, $P_1 \equiv_{wb} P_2$. □

2.2 Computation Model

In this chapter, we particularly focus on ring networks of arbitrary size consisting of isomorphic processes – processes are all copies of the same template with appropriate relabelling of actions. More specifically, suppose that a template process $P = (S, \Sigma_n, R, s_0)$ is given, in which every action in Σ_n carries a subscript n (a left-hand action) or $n+1$ (a right-hand action), where n is a natural number. We denote the set of left-(right-) hand actions by $\Sigma_\ell(\Sigma_r)$. See figure 2.2 for an example. Given such a template, the i^{th} process in the ring P_i is obtained by evaluating n to i in all the action subscripts. More formally P_i is defined as (S, Σ_i, R_i, s_0) where Σ_i is a copy of Σ_n and R_i is a copy of R in which n is evaluated to i. A linear network consisting of processes $P_0, P_1, \cdots P_{N-1}$ can be created by taking their

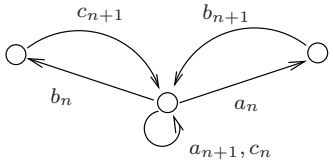


Figure 2.2: A Template Process Example

synchronous product:

$$\mathcal{L}_N := \Pi_{i=0}^{N-1} P_i = P_0 \times P_1 \times \cdots \times P_{N-1}$$

Define $P_i(mod\ N)$ exactly like P_i, but with the subscript of the actions evaluated using $mod\ N$ arithmetic. In this way, we can create a ring network of size N as follows:

$$\mathcal{R}_N := \Pi_{i=0}^{N-1} P_i(mod\ N) = \Pi_{i=0}^{N-2} P_i \ \times \ P_{N-1}(mod\ N)$$

In this structure, each process shares actions only with its immediate neighbors. Let Σ_{ri} denote those actions of P_i which are shared by its right-hand neighbor $P_{i+1}(mod\ N)$: $\Sigma_{ri} = (\Sigma_i \cap \Sigma_{i+1})(mod\ N)$. Similarly, let $\Sigma_{\ell i}$ denote the actions of P_i shared by its left-hand neighbor $P_{i-1}(mod\ N)$: $\Sigma_{\ell i} = (\Sigma_i \cap \Sigma_{i-1})(mod\ N)$. Any action of P_i which is not in Σ_{ri} or $\Sigma_{\ell i}$ is invisible.

In the above ring $\mathcal{R}_N$, one may want to model-check one specific process in the ring, say P_0, when interacting with the rest of the ring $\mathcal{S}_{N-1}$ – a ring segment of size $N-1$ which can be defined as follows:

$$\begin{aligned} \mathcal{S}_{N-1} :&= \big(\Pi_{i=1}^{N-1} P_i(mod\ N)\big) \backslash \overline{\Sigma_0} \\ &= \big(\Pi_{i=1}^{N-2} P_i \ \times \ P_{N-1}(mod\ N)\big) \backslash \overline{\Sigma_0} \end{aligned}$$

Note that hiding of actions outside Σ_0 does not have any effect on the properties of P_0 which are expressed on Σ_0. In some other network examples, the specific process P_0 may have a different structure than the other processes in the ring, or it may consist of a finite number of processes.

In the sequel, we show a recursive definition of a ring segment. We first need to define an operator $\odot$ for composition of processes over the template alphabet Σ_n. Given two template processes P' and P'' over Σ_n, define:

$$P' \odot P'' := \big((P' \times P''[n+1\backslash n, n+2\backslash n+1])\backslash(\Sigma_n \cap \Sigma_{n+1})\big)[n+1\backslash n+2]$$

Every subscript n (respectively $n+1$) in P'' is first renamed to $n+1$ ($n+2$), then the synchronous product of P' with the new P'' is taken; all the actions whose subscripts are $n+1$ are hidden (actions in $\Sigma_n \cap \Sigma_{n+1}$), and finally every subscript $n+2$ of the obtained process is renamed to $n+1$. Note that the result is a process over Σ_n, and $\odot$ is associative. Now a ring segment of size N can be defined recursively:

$$\mathcal{S}'_1 := P$$
$$\mathcal{S}'_N := P \odot \mathcal{S}'_{N-1} = \mathcal{S}'_{N-1} \odot P$$

It can be easily shown that $\mathcal{S}'_N$ and $\mathcal{S}_N$ are the same processes up to a simple renaming. More precisely:

$$\mathcal{S}'_N = \mathcal{S}_N[n\backslash 1, n+1\backslash 0]$$

Our next goal is to investigate the behavior of segments of different size $\mathcal{S}'_N$, and check whether they all fall into a finite number of equivalence classes for some equivalence relation $\equiv$. Define B to be the smallest bound (if such a bound exists) such that every segment of size larger than B is equivalent to a segment of size less than or equal to B. Suppose that $\mathcal{S}'_i \equiv \mathcal{S}'_{i+k}$ where i, k are positive natural numbers and $\equiv$ has the congruence property with respect to $\odot$. That implies $\mathcal{S}'_i \odot P \equiv \mathcal{S}'_{i+k} \odot P$. Therefore, $\mathcal{S}'_{i+1} \equiv \mathcal{S}'_{i+k+1}$ and $\mathcal{S}'_{i+2} \equiv \mathcal{S}'_{i+k+2}$ and so forth. It can be shown inductively that $\mathcal{S}'_{i+t} \equiv \mathcal{S}'_{i+r}$ where t modulo k is r. In other words, every segment of size larger than $i+k-1$ is equivalent to a ring segment of size less than or equal to $i+k-1$. On this basis, procedure $\mathcal{PROC}(P)$ compares each ring segment against the ring segments of smaller size until two ring segments $\mathcal{S}'_{k_1}, \mathcal{S}'_{k_2}$ are found

```
Procedure PROC(P)
begin
    flag:=True;
    n:=2;
    while flag do
        for i:=1 to n-1 do
            if S'_n ≡ S'_i then
                B:=n-1;
                flag:=False;
            endif
        endfor
        n:=n+1;
    endw
end
```

Figure 2.3: Procedure $\mathcal{PROC}$

such that $\mathcal{S}'_{k_1} \equiv \mathcal{S}'_{k_2}$ and $k_1 < k_2$; B is then set to $k_2 - 1$, and the procedure terminates – $\mathcal{PROC}(P)$ can be defined more formally as in Figure 2.3.

Ring segments $\mathcal{S}'_1, \mathcal{S}'_2, \cdots, \mathcal{S}'_B$ can be thought of as the representatives of ring segments equivalence classes. By $\mathcal{PROC}_{wtr}$, $\mathcal{PROC}_{ctr}$, $\mathcal{PROC}_{wf}$, $\mathcal{PROC}_{wpf}$, $\mathcal{PROC}_{wb}$, we denote $\mathcal{PROC}$ when the equivalence relation used in the procedure is $\equiv_{wtr}$, $\equiv_{ctr}$, $\equiv_{wf}$, $\equiv_{wpf}$, $\equiv_{wb}$ respectively. In the next section, it is shown that the problem of determining whether or not ring segments of arbitrary size are equivalent (for any equivalence relation stronger than weak trace equivalence and weaker than weak bisimulation) to those of bounded size is undecidable.

Suppose that for a ring network of arbitrary size $\mathcal{R}_N$, we are interested in verifying a property of the form $\bigwedge_{i=0}^{N-1} g(i)$ where $g(i)$ is an observable modal property expressed on Σ_i which is preserved under $\equiv$. By the symmetry of $\mathcal{R}_N$, the above formula holds iff $g(0)$ does.

Thus, we can restate the verification problem as that of checking whether $P_0 \times \mathcal{S}_{N-1} \models g(0)$ where N is an arbitrary natural number. If a bound B as defined above exists on ring segments, then the problem can be reduced to checking whether $P_0 \times \mathcal{S}_N \models g(0)$ for $\forall N \leq B$. The size of the largest ring network to be model-checked, also known as *cutoff size*, is therefore $B + 1$.

In particular, when $\mathcal{PROC}_{wf}$ (resp. $\mathcal{PROC}_{wpf}$) terminates we can decide whether a process can reach deadlock (resp. blocking) by only checking networks of up to size $B+1$.

A more general problem involves a property of the form $\bigwedge_{i=0}^{N-1} g(i, i+1, \cdots, i+M-1)$ where $g(i, i+1, \cdots, i+M-1)$ is a modal property expressed on $\bigcup_{j=i}^{i+M-1} \Sigma_i$ which is preserved under $\equiv$. By similar reasoning as in the previous case, the problem can be simplified to checking $g(0, 1, \cdots, M-1)$ for rings of up to a cutoff size $B+M$. Properties of the form $\bigwedge_{i=0}^{N-1} \bigwedge_{j=0}^{N-1} g(i, j)$, similarly, have a cutoff size $2B+2$ since a ring consists of processes P_i, P_j, and the two ring segments in between them which can be at most of size B. Similar results hold for LTL\X and CTL*\X properties according to the following theorem:

Theorem 1 *For any given processes E, F, P, if $E \equiv_{ctr} F$ (respectively $E \equiv_{wb} F$), then for every LTL\X (CTL*\X) formula ψ on the states of P, $P \times E \models \psi$ iff $P \times F \models \psi$.*

In fact, it can be shown by using natural induction that when $E \equiv_{ctr} F$, then $P \times E$ and $P \times F$ both have the same set of computations. Similarly, it can be shown that when $E \equiv_{wb} F$, then $P \times E$ and $P \times F$ perfectly mimic one another while preserving the atomic propositions of P. For more results on preserved temporal properties with respect to equivalence relations or partial order relations, refer to [41].

Note that for some fragments of CTL*\X, one may come up with a coarser equivalence relation than $\equiv_{wb}$ which preserves the truth of their formulae. For instance, the truth of formulae of the form $AG\ EF\ s$, where $s \in S$, is preserved under $\equiv_{wpf}$.

Therefore, if for a given ring network and an equivalence relation, our suggested procedure $\mathcal{PROC}$ terminates, then that network can be verified for preserved temporal and modal properties under that equivalence relation. However, $\mathcal{PROC}$ is not guaranteed to terminate.

In the sequel, an example of a ring network is shown in which the procedure of checking segments for weak trace equivalence never terminates.

Example: Consider the template of a ring network as shown in Figure 2.4: action con_{n+1} establishes connection with the immediate right-hand process, con_n establishes connection with the immediate left-hand process. Similarly, dc_{n+1} disconnects the process from its immediate right-hand process, and dc_n disconnects the process from its immediate left-hand process. Let the tuple $(s_1, s_2, \cdots s_N)$ denote the global state of a segment S_N

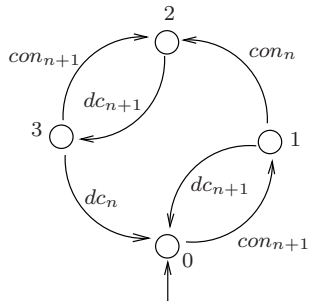


Figure 2.4: template process P

where $s_i \in \{0, 1, 2, 3\}$ represents the local state of process P_i. The visible actions of S_N have either a subscript 1 (left-hand actions of P_1), or a subscript $N+1$ (right-hand actions of P_N). The rest of the actions are invisible. By notation $i_1, i_2, \cdots i_m \mapsto j_1, j_2, \cdots j_m$, we denote that a global state which has a subsequent of states of the form $i_1, i_2, \cdots i_m$ can evolve to another global state whose corresponding subsequent of states is $j_1, j_2, \cdots j_m$ without affecting the rest of the segment. For instance, consider the local states of two neighbor processes P_k, P_{k+1} to be 0,1 respectively; then, they can share the action con_k to

evolve to the local states $1, 2$. Therefore, $0, 1 \mapsto 1, 2$. It can also be shown that

$$\text{I}: \quad 0, 1, 3 \mapsto 1, 3, 0$$

$$\text{II}: \quad 0, 1, 3 \mapsto 0, 0, 0$$

If the states of the two rightmost processes in a segment of size N are $0, 0$, then they can evolve to $1, 3$ by performing the sequence $con_{N+1}\ con_N\ dc_{N+1}\text{mod}(N+1)$ whose projection onto visible actions is $con_0\ dc_0$. Initially, all the processes in S_N are in their initial state 0; therefore, the two rightmost states can evolve to $1, 3$. By applying I repetitively, this pair can then be shifted to the left until the local states of the leftmost processes are $0, 1, 3$. Again, a new pair $1, 3$ can be generated by the two rightmost processes (by performing $con_0\ dc_0$) and shifted to the left until they reach the previous 1,3 pair. This can be repeated until a global state of the form $0, 1, 3, 1, 3, \cdots, 1, 3$ or $0, 1, 3, 1, 3, \cdots, 1, 3, 0$ is reached. Now by repetitive application of II, the pairs 1,3 can be removed starting from the leftmost one. Note that each time a pair 1,3 is removed a sequence of visible actions $con_1\ dc_1$ is performed. Consequently, the language of a segment of size $2N+1$ includes strings of form $(con_0\ dc_0)^i(con_1\ dc_1)^i$, $i \leq N$. Thus, such a ring segment can count the number of $con_0 dc_0$ substrings up to N which requires at least $N+1$ states. Consequently, the number of states of minimal S_N [3] increases as N grows; therefore, $\mathcal{PROC}_{wtr}(P)$ is not terminating. □

Remark: Rings with Unary Tokens. Consider a special case of ring networks in which processes communicate by passing a single unary token around the ring (as in [6]), and holding the token allows a process to execute a set of special (token-dependent) actions. Owing to the ring topology, additional tokens can never enter the network, but opening the loop into a ring segment allows multiple tokens to enter (the exact upper limit depending on the number of processes in the segment). For this reason, $\mathcal{PROC}$ does not

[3] Minimal S_N is the process with the minimum number of states which is weakly trace equivalent with S_N.

terminate for such systems. However, we can redefine ring segments as $\mathcal{S}'_N = (\mathcal{S}'_{N-1} \odot P) \times I$ where $I = (\{0,1\}, \{rcv_n, snd_{n+1}\}, \{(0, rcv_n, 1), (1, snd_{n+1}, 0)\}, 0)$; action rcv_n (respectively, snd_{n+1}) denotes the reception (respectively, sending) of a token. This results in termination of $\mathcal{PROC}_{wb}$. In fact, $B = 1$ for such rings, and the same cutoffs as in [17] can be achieved for different sets of temporal properties, explained earlier in chapter 1.

2.3 Undecidability Results

In this section, three important undecidability results of ring networks will be presented, which are mainly extensions to the results of [20, 21]. In fact, it will be shown that for any given equivalence relation finer than weak trace equivalence and coarser than weak bisimulation, and any given ring template, it is not decidable whether the number of ring equivalence classes are finite, and if so, give a bound on the size of the smallest representatives. Next, the problem of detecting component and network blocking is shown to be undecidable. Finally, the undecidability of checking whether the number of ring segment equivalence classes are finite will be proved.

2.3.1 Ring Networks Equivalence Classes

The problem of determining whether or not the rings of arbitrary size fall into a finite number of equivalence classes, and if so giving a bound on the size of the smallest representatives of these classes is undecidable. In [1], this was shown only for weak bisimulation. In the following theorem, we will show that it holds for any equivalence which refines weak trace equivalence and is coarser than weak bisimulation [21].

Theorem 2 *For any given equivalence relation finer than weak trace equivalence, and coarser than weak bisimulation and any given template, there is no algorithm which determines whether or not the number of equivalent classes of rings of arbitrary size is finite,*

and if so, computes a bound on the size of the smallest representatives of these classes.

Proof: The proof is by reduction from the halting problem. We take a similar approach to [1].

Given a Turing machine M, we construct a ring template P with two registers: "tape register" which can store a tape symbol of M, and a "state register" to store the state of M's control unit. We also define four different modes for P: $idle, active, passive, dead$. The template is initially in its *idle* mode with its tape register filled with the empty symbol, and its state register set to the initial state of M's control unit. The template, when in its *idle* mode, can become *active* by generating a token, or become *passive* by receiving a token from its left-hand neighbor. It can also turn into *dead* mode, and do nothing after that; it won't generate any token, and destroys any token received from its neighbors. The mode of a process will remain unchanged after switching to *active*, *passive*, or *dead*. Now consider a ring network consisting of N copies of the template P – initially all in their *idle* mode. The intuition is to simulate the Turing machine M with a sequence of consecutive processes where the tape register of each process accounts for a single tape cell of M. A process which becomes *active* possesses the leftmost tape cell, and his token denotes where the tape-head initially points to. A process holding a token can simulate M with respect to his current state and tape registers, and update their values. The token is then moved to his right, left, or stays stationary depending on where M's tape head moves. As a generated token moves further to the right, it may encounter *idle* processes, and change their mode to *passive*; or may encounter an *active* process, and destroy; i.e., the *active* process consumes the token, and won't pass it to another neighbor process. In fact, an *active* process, and all the consecutive *passive* processes to its right can faithfully simulate the computation of the Turing machine M until the token of that ring segment hits another *active* process; this destroys the token, and the simulation of the Turing machine stops incomplete. Note that if the state register of a process sets to the "halting state" of M's control unit, then we know that M halts on the empty string, and its computation has been

perfectly simulated by the corresponding ring segment. At this point, the token holding the "halting state" is passed to the left until it reaches the *active* process, and destroys.

Now suppose that the Turing machine halts by going through N tape cells. We shall show that a ring of size $2N-1$ is weakly bisimilar to any ring of larger size $2N+K-1$, $K \geq 0$; i.e, $\mathcal{R}_{2N-1}\backslash\overline{\Sigma_0} \equiv_{wb} \mathcal{R}_{2N+K-1}\backslash\overline{\Sigma_0}$, $K \geq 0$, Σ_0 is the set of visible actions of P_0. Figure 2.5 provides intuition. The ring depicted on the left, consists of $2N-1$ processes, and the one on the right consists of $2N+M-1$ processes. Our goal is to prove that the two rings are weakly bisimilar. We have slightly changed the indices of processes in the two rings in order to make the proof more clear. The processes have also been categorized in 3 segments. The first $N-1$ processes on the left-hand side of P_0 (resp. P'_0) comprise segment A, and the first $N-1$ processes on the right-hand side of P_0 (resp. P'_0) comprise segment B. Note that every process P_i in these two segments of $\mathcal{R}_{2N-1}$ has a corresponding process P'_i with the same index in $\mathcal{R}_{2N+K-1}$. Furthermore, the processes with indices $N, N+1, \cdots, N+K-1$ comprise segment C of $\mathcal{R}_{2N+K-1}$, and they don't have any corresponding process in $\mathcal{R}_{2N-1}$. For notation simplicity, sometimes we denote by A, B, C the set of indices in each of those segments. For instance, $B = \{1, 2, \cdots N-1\}$.

The rough intuition behind this proof is to correspond processes with the same indices in A (resp. B) segments of the two rings in order to simulate one another, and show that the actions of processes in C segment will not affect the behavior of P'_0. Let's denote by x, y the global states of the rings $\mathcal{R}_{2N-1}$ and $\mathcal{R}_{2N+K-1}$ respectively. Also suppose that x_i and y_j denote the states of processes P_i, P'_j in the two rings respectively. By $mode(x_i)$ ($mode(y_j)$), we denote the mode of process P_i (resp. P'_j) is state x_i (resp. y_j). Also let $org(x, P_i)$ be the index of the unique *active* process whose token has reached P_i in global state x. In other words, $org(x, P_i)$ returns the origin of the token that has reached P_i, and changed his mode to *passive*. If P_i is in an *active* mode itself, then the index of P_i, i, is returned, and if P_i is *idle* or *dead*, no index is returned. The function $org(y, P'_j)$ is defined similarly.

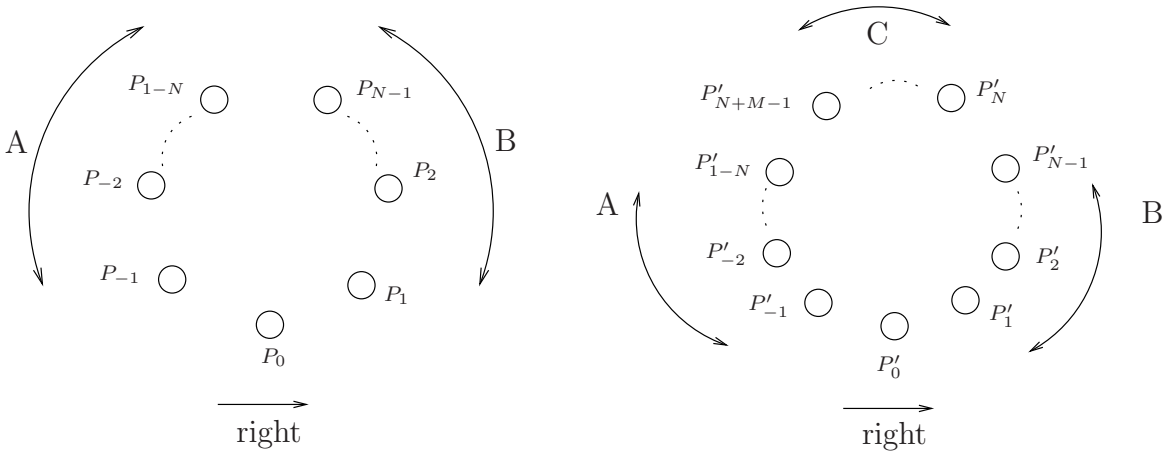


Figure 2.5: Weak Bisimilarity of Rings

We define the relation *Rel* to relate two global states x, y of $\mathcal{R}_{2N-1}\backslash\overline{\Sigma_0}$ and $\mathcal{R}_{2N+K-1}\backslash\overline{\Sigma_0}$ when the following conditions hold:

- $x_0 = y_0$;
- for every $i \in A$, $[x_i = y_i]$ or $[mode(x_j) = mode(y_j) = active$ for some $i < j \leq -1]$ or $[mode(x_i), mode(y_i) \in \{passive, dead\}$ and $org(x, P_i), org(y, P_i) \notin A]$;
- for every $i \in B$, $[x_i = y_i]$ or $[mode(x_j) = mode(y_j) = active$ for some $1 \leq j < i]$.

The first condition implies that processes P_0, P'_0 should both be in the same states. According to the second condition, the states of corresponding processes in segment A also need to be the same unless either 1) active processes P_j, P'_j exist in that segment which are closer to P_0, P'_0 than P_i, P'_i respectively. If that's the case then the transitions of processes before P_j, P'_j (including P_i, P'_i) won't affect the behavior of P_0, P'_0 respectively. This is because every token passed to an active process from its left is destroyed by that process, and cannot be carried further to the right. In other words, x_i, y_i could be different for $-N + 1 \leq i < j$; or 2) the modes of P_i, P'_i are *passive* or *dead*. Furthermore, if P_i (P'_i)

is *passive*, his unique *active* process is not in segment A. In fact, if that's the case the transitions of P_i, P_i' could not affect the behavior of P_0, P_0' (the generated token by the active process cannot reach P_0, P_0'), and therefore, x_i, y_i don't have to be the same. The third condition, enforces corresponding processes in segment B to have the same states unless *active* processes P_j, P_j' exist somewhere in that segment. If so, P_j, P_j', and those processes which are placed on their right, don't have to be matched in their states. In fact, when a token is passed to P_j, P_j' from their left, it is destroyed regardless.

Next, we need to show that the defined relation *Rel* is, in fact, a bisimulation relation. The idea is that whatever action either of $\mathcal{R}_{2N-1}\backslash\overline{\Sigma_0}$ or $\mathcal{R}_{2N+K-1}\backslash\overline{\Sigma_0}$ performs, can be mimicked by the other one in order to keep them in bisimilar states. The first $N-1$ processes on left- and right-hand side of P_0, P_0' are the only ones that can affect their behavior. Therefore, *Rel* is defined such that these neighboring processes are kept in identical states – as long as they can affect the behavior of P_0, P_0'. A subtle case is when a process in segment C of $\mathcal{R}_{2N+K-1}\backslash\overline{\Sigma_0}$ becomes active, and passes his token further down to his right into segment A. This may turn some *idle* processes in segment A into their *passive* mode, and therefore, they won't be able to affect the behavior of P_0' anymore. In such case, the corresponding processes in segment A of $\mathcal{R}_{2N-1}\backslash\overline{\Sigma_0}$ change their mode to *dead*. This will have the same effect from P_0, P_0''s perspective. The other scenarios can be explained in a similar way. On the other hand, the initial states of the two rings are also related according to *Rel*; therefore, $\mathcal{R}_{2N-1}\backslash\overline{\Sigma_0}, \mathcal{R}_{2N+K-1}\backslash\overline{\Sigma_0}$ are weakly bisimilar. Our proof holds for any natural number K. Consequently, if M halts on the empty string, then rings of different size are weakly bisimilar to the rings of size smaller than, or equal to a computable bound.

Now assume that all the rings of arbitrary size fall into a finite number of trace equivalence classes on which we can give a corresponding bound. If the Turing machine M halts on the empty string, then a possible executable string in an arbitrary large ring is when P_0 is passed a token carrying the halting-state symbol. The projection of this string onto the

visible actions includes the particular action of passing the token carrying the halting-state symbol to P_0. On the other hand, if such a string belongs to the visible language of a ring, then M halts on the empty string. This can be easily proved according to the construction used. Consequently, we only need to test the equivalence classes to see if any of them includes that string, and then it is known whether M halts on the empty string, or not.

Suppose that there exists an algorithm as described in the theorem. We apply it to the case when the template of the ring is constructed according to the Turing machine M. If the number of equivalence classes for a specific equivalence relation $\equiv$ is infinite, then so is the number of weak bisimulation classes (since $\equiv_{wb} \leq \equiv$), and therefore, the Turing machine M does not halt on the empty string. If the number of equivalence classes is finite, and we have a bound on their smallest representatives, then the same result holds for weak trace equivalence (since $\equiv \leq \equiv_{wtr}$), and therefore, halting of M can be checked. This implies that the halting problem is decidable which is a contradiction. □

2.3.2 Component and Network Blocking

By taking the same approach as in the previous problem, it can be proved that the blocking problem for ring networks is also undecidable. A ring network of arbitrary size $\mathcal{R}_N$ is said to have *component blocking* iff a process in that ring can reach a state from which no marker state is reachable. In other words, when a component in a particular instance of the ring is blocking. On the other hand, $\mathcal{R}_N$ is said to have *network blocking* iff it can reach a global state g which cannot evolve to a state with all processes in their marker states. [1].

Theorem 3 *The problems of deciding whether a ring network has component blocking (CBP), or network blocking (NBP) are undecidable.*

Again, we can show that the halting problem can be reduced to CBP and NBP. We only need to make a slight change in the construction of the ring template P from a given

Turing machine M, such that every time the state register holds the halting state, P enters a non-marker state, and remains there forever. Therefore, the Turing machine halts on the empty string iff the network with the constructed template P has component or network blocking. Note that for such construction, component and network blocking are equivalent; i.e., a network has component blocking iff it has network blocking. □

2.3.3 Ring-Segments Equivalence Classes

The last undecidability result relates to ring segments of arbitrary size rather than rings themselves. We will show that for a given template, there is no algorithm which decides on the equivalence of arbitrary size ring segments to ring segments of bounded size. The proof is done by reduction from the mortality problem [21].

Let M be a Turing machine which has a two-way infinite tape. Let Σ be its finite tape alphabet, and Q be the set of control states. A configuration of M can be represented as $lqr \in \Sigma^\omega Q \Sigma^\omega$ where $l, r \in \Sigma^\omega$ are infinite strings of tape symbols, and $q \in Q$ is a state of the control-unit. The read/write head of the tape is assumed to be on the leftmost letter of r. The Turing machine M is called *mortal* if and only if it always halts regardless of its initial configuration. The problem of determining whether a given Turing machine is mortal is called the *mortality problem*.

Theorem 4 *The mortality problem is undecidable [22].*

It is important to note that in the mortality problem the Turing machine is assumed to have a two-way tape, and the initial configuration is arbitrary, i.e., the tape maybe nonempty and the control state of the initial configuration could be any state from the set of states Q, which makes this problem different from the halting problem.

The Turing machine M is called *uniformly mortal* iff it halts starting from any initial configuration in a uniformly bounded number of steps. It is known that uniform mortality is equivalent to mortality:

Theorem 5 *A Turing machine is mortal if and only if it is uniformly mortal [23].*

According to the previous two theorems, the problem of uniform mortality is undecidable, and we shall show that it can be reduced to the ring-segments equivalence problem:

Theorem 6 *Given a template process P, the problem of determining whether ring segments, S_N, of arbitrary size are all equivalent to ring segments of bounded size, for any given equivalence relation which refines weak trace equivalence and is coarser than weak bisimulation equivalence, is undecidable.*

Proof: For a given Turing machine M, we will construct a template which can input a possible configuration of M, and output its successor configuration. A configuration of the form $\ldots l_3 l_2 l_1 q r_1 r_2 r_3 \ldots$ will be fed into the template process in the form of the string $q r_1 l_1 r_2 l_2 r_3 l_3 \ldots$. The template is equipped with the look-up tables of M in order to model its transitions. As soon as the first two letters of the input string are received by the template, it can compute the next state, the new tape symbol of the current tape-head position, and also the new position of the tape-head according to the look-up tables. We use the following notation to describe the input-output relation of the template:

q	r_1	l_1	r_2	l_2	r_3	l_3	$\cdots$	
	q'	r'_1	l'_1	r'_2	l'_2	r'_3	l'_3	$\cdots$

The string above the line shows the order of the inputs to the template, and the one below the line represents the corresponding output sequence. This notation represents the temporal interleaving of the input and output strings; an output event occurs right after the input event which is directly above it. So the visible string in the above example is of the form: $q\ r_1\ q'\ l_1\ r'_1\ r_2\ l'_1\ l_2\ r'_2\ r_3\ l'_2\ l_3\ r'_3\ l'_3\ \ldots$.

As stated earlier, the template is to simulate the Turing machine, and output the successor configuration of any valid configuration which is fed into it. In order to do that,

we construct the template according to the following rules – in all of these rules, we assume that q r_1 are the first inputs to the template, which implies that the control unit of the Turing machine is in state q, and the tape-head reads r_1. Let the Turing machine write r_1' to the tape, and go to a new state q'. Different cases arise depending on whether q is a halting state, and what direction the tape-head moves:

- If q is a non-halting state and the tape-head stays stationary, then

q	r_1	l_1	r_2	l_2	r_3	l_3	$\cdots$
		q'	r_1'	l_1	r_2	l_2	$\cdots$

- If q is a non-halting state and the tape-head moves to the right, then

q	r_1	l_1	r_2	l_2	r_3	l_3	$\cdots$
		q'	r_2	r_1'	r_3	l_1	$\cdots$

- If q is a non-halting state and the tape-head moves to the left, then

q	r_1	l_1	r_2	l_2	r_3	l_3	r_4	l_4	$\cdots$
		q'	l_1	l_2	r_1'	l_3	r_2	l_4	$\cdots$

- If q is a halting state, then

q	$\cdots$
q	

Now consider a linear segment consisting of the instances of the constructed template. The first instance receives as its input the initial configuration, and then it outputs the successor configuration, which in turn becomes the input to the second instance; the second instance outputs the third configuration for the next instance, and so forth. This continues until a halting configuration (a configuration with a halting state) is generated. The first instance which receives such a configuration only passes the state symbol along, and ignores

the rest of inputs. Note that the first input of each instance has to be a state symbol, otherwise it won't be accepted, and for this to be possible the tape symbols have to be different from the state symbols.

Assume that for the above linear segment whose template is simulating a Turing machine M, the segments of arbitrary size are all weakly trace equivalent to the segments of size B or smaller. Note that the existence of such a bound implies that it can be computed. In fact, ring segments of different size can be compared until two ring segments are found equivalent. This will be explained in further details in the next section. Therefore, the output configuration of any instance K_1 such that $K_1 > B$ should be the same as an output configuration of an instance K_2 such that $K_2 \leq B$. Therefore, the Turing machine can go through at most $B+1$ distinct configurations regardless of its initial configuration. Consequently, it is a finite-state machine of size smaller than a computable bound, and therefore, can be checked for uniform mortality.

On the other hand, if M is uniformly mortal, then sufficiently large segments can be shown to be weakly bisimilar. Assume that M halts in at most c computation steps. For any two segments of size greater than c, we can provide a relation on their set of states which holds when each of the first c instances of one segment has the same state as its respective counterpart in the other one, and their last instances are also in the same states. This is a weak bisimulation relation because for identical sequence of inputs, the first c instances of both segments can make the same transitions, and their last instances cannot perform any action until the "halting state" is outputted by one of the first c instances, and passed along to them. Then, they can output the "halting state" symbol, and stay in the same states. Furthermore, the defined relation includes the pair of the initial states of the both segments. Therefore, any two segments of size greater than c are weakly bisimilar.

Now let's say for a given equivalence relation $\equiv$, it is decidable whether all segments of different size fall into a finite number of classes. Apply it to the above construction. If the number of equivalence classes is finite, then so is the number of weak trace equivalence

classes since weak trace equivalence is coarser than $\equiv$ according to the theorem. It can then be determined whether the Turing machine M is uniformly mortal.

If there are infinite number of equivalence classes, then the number of weak bisimulation classes is infinite as well – weak bisimulation is stronger than $\equiv$ according to the theorem. Consequently, M is not uniformly mortal. In other words, the mortality problem is decidable which is a contradiction. □

2.4 Termination of $\mathcal{PROC}$

In this section, we propose some conditions on the structure of the ring template that are sufficient for termination of our proposed procedure.

2.4.1 Piecewise Recognizable Processes

Assume that the template process $P = (S, \Sigma_n, R, s_0)$ of a ring network is PR. Given any ring segment $\mathcal{S}_N$, we shall construct a tree process T_N which is weakly trace equivalent to $\mathcal{S}_N$. Then we show that T_N's, for arbitrary values of N, are all subtrees of a tree process which can be constructed according to P. The number of such subtrees is finite, therefore, the number of weak trace equivalence classes of ring segments of arbitrary size is finite. This results in $\mathcal{PROC}_{wtr}(P)$ to be terminating. An upper bound on the number of these classes will then be introduced which is double exponential in the number of non-self-loop transitions of the ring template P.

Process $\mathcal{S}_N = (M, \Sigma_0, \Delta, i)$, as defined earlier, is composed of N processes $P_1 \cdots P_N$ where P_1 (respectively P_N) is the leftmost (rightmost) process. Every global state $s_g \in M$ is of the form $(s_1, s_2, \cdots, s_n)$ where $s_j \in S$, the j^{th} element of s_g, represents the local state of process P_j, and is denoted by $s_g(j)$. The initial state $i = (s_0, s_0, \cdots, s_0)$. Every transition of $\mathcal{S}_N$ corresponds to either a transition of an individual process, or a shared transition between a pair of neighbor processes. Define the transitions of $\mathcal{S}_N$ which have a

corresponding non-self-loop transition of P_1 or P_N as type I transitions – the transitions which change the 1^{st} or N^{th} coordinates of a global state in $\mathcal{S}_N$; those with a corresponding self-loop transition of P_1 or P_N are defined as type II, and the rest as type III. Note that only type I and II transitions may be visible, according to the definition of $\mathcal{S}_N$. Assume that the ring template P has m non-self-loop transitions, then so do P_1 and P_N. Let's name every such transition in P_1, P_N by ℓ_i, r_i respectively, where $1 \leq i \leq m$. Then the set of transitions of P_1 and P_N can be denoted by $\Lambda_\ell = \{\ell_1, \ell_2, \cdots \ell_m\}$ and $\Lambda_r = \{r_1, r_2, \cdots r_m\}$ respectively. The projection function $Proj_I : \Delta \rightarrow \Lambda_\ell \cup \Lambda_r$ is defined such that given a type I transition of $\mathcal{S}_N$, it returns the corresponding non-self-loop transition of P_1 or P_N, and returns $\flat$ otherwise where $\flat$ denotes the empty sequence of transitions. This function can be extended to $Proj'_I : \Delta^* \rightarrow (\Lambda_\ell \cup \Lambda_r)^*$ inductively as follows:

$$Proj'_I(\flat) = \flat$$
$$Proj'_I(t\overline{t}) = Proj_I(t)Proj'_I(\overline{t})$$

where $t \in \Delta$ and $\overline{t} \in \Delta^*$. For simplicity, we use $Proj_I$ instead of $Proj'_I$ throughout the rest of the paper. We also define the projection of the sets of computation paths as follows:

$$Proj_I(\Gamma) = \{Proj_I(c) : c \in \Gamma\}$$

Define process T_N as the tuple $(M', \Sigma_0, \Delta', \flat)$ where the set of states $M' = Proj_I(\mathcal{C}(\mathcal{S}_N))$ which is finite. The transition relation Δ' is defined as: $\{(Proj_I(c), Act(c_1), Proj_I(cc_1)) : cc_1 \in \mathcal{C}(\mathcal{S}_N)\ \&\ c_1 \in \Delta\}$.

We will show that every computation path $c \in \mathcal{C}(\mathcal{S}_N)$ has a corresponding computation path $c' \in \mathcal{C}(T_N)$ with the same sequence of visible actions, and vice versa.

Theorem 7 *$\mathcal{S}_N$ and T_N are weakly trace equivalent.*

First we need to prove the following lemma.

Lemma 1 *If for* $c_1, c_2 \in \mathcal{C}(\mathcal{S}_N)$, *we have* $Proj_I(c_1) = Proj_I(c_2) = c_p$, $i \overset{c_1}{\rightsquigarrow} s_{g_1}$ *and* $i \overset{c_2}{\rightsquigarrow} s_{g_2}$, *then:*

- $\forall \alpha \in \Sigma_0, s_{g_1} \xrightarrow{\alpha} s_{g_1}$ iff $s_{g_2} \xrightarrow{\alpha} s_{g_2}$;
- $\exists c_1' \in \mathcal{C}(\mathcal{S}_N).Proj_I(c_1') = c_p, Act_v(c_1') = Act_v(c_2)$ and $i \overset{c_1'}{\rightsquigarrow} s_{g_1}$;
- $\exists c_2' \in \mathcal{C}(\mathcal{S}_N).Proj_I(c_2') = c_p, Act_v(c_2') = Act_v(c_1)$ and $i \overset{c_2'}{\rightsquigarrow} s_{g_2}$.

Visible self-loop transitions (type II) in any global state of $\mathcal{S}_N$ correspond to visible self-loop transitions of P_1 and P_N. On the other hand, if a local state s_l of P_1 (P_N) allows for a self-loop action α (it is obviously not shared with any other process of $\mathcal{S}_N$), then any global state s_g of $\mathcal{S}_N$ whose first (last) coordinate is s_l has an α self-loop transition. Therefore, any two global states of $\mathcal{S}_N$ whose first and last coordinates are the same allow for the same self-loop visible transitions. Taking computation paths in $\mathcal{S}_N$ with the same projection under $Proj_I$ results in such global states, and therefore, the first assertion of the lemma holds.

The computation paths of $\mathcal{S}_N$, whose projections under $Proj_I$ are the same, may only differ (in terms of sequence of visible actions) because of visible type II transitions occurring in between type I transitions since type III transitions are invisible. As stated earlier in the proof, visible type II transitions allowed in global states of $\mathcal{S}_N$ are the same as long as they have the same first and last coordinates. On this basis, the second and third assertions can be easily proved by using induction on the length of c_1 and c_2 respectively. □

Now let $c \in \mathcal{C}(\mathcal{S}_N)$. By using induction on the length of c, we show that there exists $d \in \mathcal{C}(T_N)$ such that $Act_v(c) = Act_v(d)$ and $\flat \overset{d}{\rightsquigarrow} Proj_I(c)$. The basic case when $c = \flat$ is obvious. Suppose that the proposition holds for computation paths of length m; we prove it for those of length $m+1$. A computation path of length $m+1$ can be written as cc_1 where $|c| = m$ and $|c_1| = 1$. Let $d_1 := (Proj_I(c), Act(c_1), Proj_I(cc_1))$, then $\flat \overset{dd_1}{\rightsquigarrow} Proj_I(cc_1)$

and

$$\begin{aligned} Act_v(dd_1) &= Act_v(d)Act_v(d_1) \\ &= Act_v(c)Act(c_1) \\ &= Act_v(cc_1). \end{aligned}$$

Therefore, dd_1 is the corresponding computation path in T_N.

Now we show that corresponding to every $d \in \mathcal{C}(T_N)$, there exists $c \in \mathcal{C}(\mathcal{S}_N)$ such that $Act_v(c) = Act_v(d)$ and $\flat \overset{d}{\rightsquigarrow} Proj_I(c)$. For the basic case when the computation path is empty, this is obvious. Suppose that the proposition holds for the computation paths of length m; we prove it for computation paths of length $m+1$. Consider a computation path of length $m+1$, dd_1, where $|d| = m$ and $|d_1| = 1$; d_1 is a transition of T_N, therefore, according to the transition relation definition of T_N, there should exist $c', c'c_1 \in \mathcal{C}(\mathcal{S}_N)$ such that $d_1 = (Proj_I(c'), Act(c_1), Proj_I(c'c_1))$. The source state of d_1 has to be the same as the destination state of d: $Proj_I(c) = Proj_I(c') = c_p$. Now according to lemma 1, there should exist $c'' \in \mathcal{C}(\mathcal{S}_N)$ such that c' and c'' reach the same global states, $Act_v(c) = Act_v(c')$ and $Proj_I(c'') = c_p$. We have $Proj_I(c'c_1) = Proj_I(c''c_1)$; therefore $\flat \overset{dd_1}{\rightsquigarrow} Proj_I(c''c_1)$ and

$$\begin{aligned} Act_v(c''c_1) &= Act_v(c'')Act_v(c_1) \\ &= Act_v(c)Act(c_1) \\ &= Act_v(d)Act(d_1) \\ &= Act_v(dd_1). \end{aligned}$$

Consequently, $c''c_1$ is the corresponding computation path in $\mathcal{S}_N$, and the proof is complete. □

A computation path is called *non-stuttering* iff none of its transitions are self-loop. Let's denote the set of non-stuttering computation paths of P_1 and P_N by $\mathcal{NS}_\ell$ ($\subseteq \Lambda_\ell^*$) and $\mathcal{NS}_r$ ($\subseteq \Lambda_r^*$) respectively; also denote by $inter(\mathcal{NS}_\ell, \mathcal{NS}_r)$ the set of all possible interleavings of

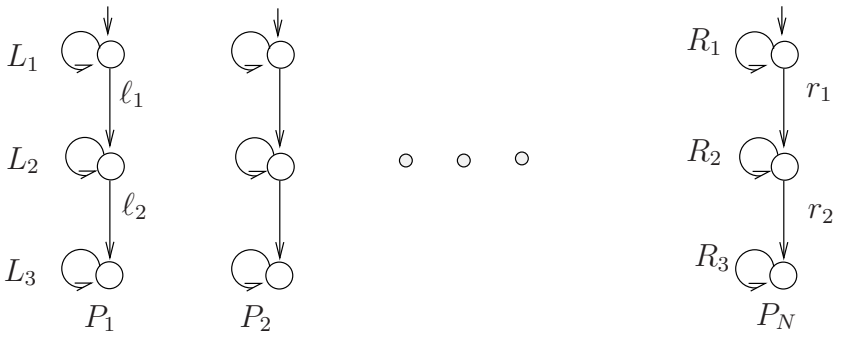


Figure 2.6: segment of size N $(\mathcal{S}_N)$

the computations in $\mathcal{NS}_\ell$ and $\mathcal{NS}_r$. It is not hard to see that $M' \subseteq inter(\mathcal{NS}_\ell, \mathcal{NS}_r)$ for any tree T_N. We call the special tree, whose set of states is $inter(\mathcal{NS}_\ell, \mathcal{NS}_r)$, the *maximal tree*, and denote it by T_{max}. In fact, every tree T_N is a subtree of T_{max}. Figure 2.6 shows a special case of a ring segment whose template process is thin piecewise and has only three states. The transitions of P_1 and P_N are annotated with labels: ℓ_i, r_i denote the non-self-loop transitions and L_i, R_i the sets of self-loop transitions.

The maximal tree of this ring is depicted in Figure 2.7. For the sake of simplicity, self-loop transitions are not shown. Note that every state s of a tree T_N represents those global states of $\mathcal{S}_N$ whose first as well as last coordinates are the same. These coordinates represent the local states of P_1 and P_N. Self-loop transitions at s are defined according to self-loop transitions at these local states. For instance, consider the state $s = r_1\ell_1 r_2$; this state corresponds to those global states of $\mathcal{S}_N$ in which P_1 is in state 2 and P_N in state 3 (the states of the ring template are named 1 to 3 from top to bottom). Therefore, actions of self-loop transitions at state s are the same as the actions of transitions in $L_2 \cup R_3$.

In a more general case when the template process of a ring is thin piecewise with $m+1$ states, a computation path of any possible subtree, starting from its initial state and ending at a leaf, would have at most m right edges and m left edges. We denote the total number of such subtrees by $M(m, m)$ where $M(m, n)$ can be recursively computed by the following

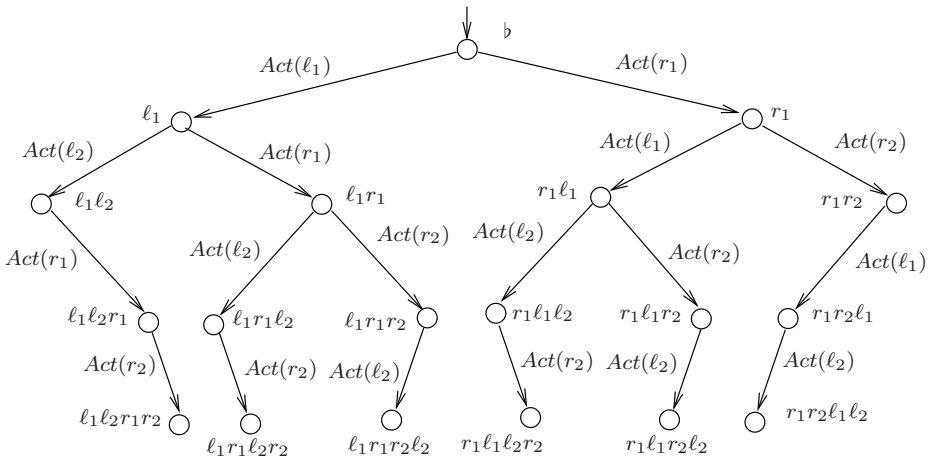


Figure 2.7: Process T_{max}

recursion formula:

$$M(m,n) = (1 + M(m-1,n))(1 + M(m,n-1))$$

$M(m,n) = 0$ for $m < 0$ or $n < 0$. It can also be shown that for any PR template (not necessarily thin piecewise), whose number of non-self-loop transitions is m, $|sub(T_{max})| \leq M(m,m)$. In the following theorem, $M(m,m)$ is shown to be double exponential in m.

Theorem 8 $M(m,m) \in 2^{2^{\Theta(m)}}$.

We first show that $2^{2^m} \leq M(m,m)$ for every $m \geq 1$ by using induction on m. For the base case when $m = 1$, $M(1,1) = (1 + M(1,0))^2 = (1+2)^2 = 9 \geq 2^2$. Now assume that the proposition holds for some natural number m: $2^{2^m} \leq M(m,m)$. Therefore, $M(m+1,m+1) = (1+M(m,m+1))^2 \geq M(m,m)^2 \geq (2^{2^m})^2 = 2^{2^{m+1}}$. So, the proposition holds for $m+1$ as well.

Next, we will show that $M(m,m) \leq 2^{2^{2m}}$ for every $m \geq 1$. For the base case when $m = 1$, $M(1,1) = 9 \leq 2^{2^2} = 16$. Similarly, assume that the proposition holds for some

natural number m: $M(m,m) \leq 2^{2^{2m}}$; we prove it for $m+1$. We have

$$\begin{aligned} & M(m+1,m+1) \\ & \quad = (1+M(m,m+1))^2 \\ & \quad = (1+(1+M(m,m))(1+M(m-1,m+1))^2 \\ & \quad \leq (M(m,m)M(m,m))^2 = (M(m,m))^4 \\ & \quad \leq (2^{2^{2m}})^4 = 2^{2^{2(m+1)}}, \end{aligned}$$

and the proof is complete. □

By taking a similar approach, we show that $\mathcal{PROC}_{wf}$ is terminating. In this case, we assign a pair $(T_N, \mathcal{A}_N)$ to every ring segment $\mathcal{S}_N$ where T_N represents the visible language of $\mathcal{S}_N$ ($L_v(\mathcal{S}_N) = L_v(T_N)$), and $\mathcal{A}_N$ is a function which assigns a subset of 2^{Σ_0} to every state of T_N. Let's say state s of T_N is reachable by a visible string $\overline{\sigma}$ and $A \in \mathcal{A}_N(s)$, then it means that $(\overline{\sigma}, A) \in Failures(\mathcal{S}_N)$. Let $Proj_I^{-1} : (\Lambda_\ell \cup \Lambda_r)^* \rightarrow \Delta^*$ be the function which inputs a state c_p of T_N and outputs the set of all computation paths c of $\mathcal{S}_N$ whose projections under $Proj_I$ are c_p:

$$Proj_I^{-1}(c_p) = \{c \in \mathcal{C}(\mathcal{S}_N) : Ptoj_I(c) = c_p\};$$

the function $\mathcal{A}$ is then more formally defined as follows:

$$\mathcal{A}_N(c_p) = \bigcup_{s_g \in reach(Proj_I^{-1}(c_p))} 2^{\mathcal{R}jct(s_g)}$$

Clearly, if $(T_N, \mathcal{A}_N) = (T_M, \mathcal{A}_M)$ for two distinct natural numbers N, M, then $\mathcal{S}_N$ and $\mathcal{S}_M$ are failure equivalent. On the other hand, the number of possible pairs $(T_N, \mathcal{A}_N)$ is finite because both T_N's and $\mathcal{A}_N$'s are finite for a given ring template. This results in a finite number of failure equivalence classes of ring segments, and therefore, $\mathcal{PROC}_{wf}$ terminates. Complete trace equivalence is coarser than weak failure equivalence; therefore, the number of complete trace equivalence classes of ring segments is finite as well, and this guarantees the termination of $\mathcal{PROC}_{ctr}$.

Similarly, it can be shown that $\mathcal{PROC}_{wpf}$ is terminating. Here, the pair assigned to a ring segment $\mathcal{S}_N$ is of the form $(T_N, \mathcal{B}_N)$ where T_N represents the visible language of $\mathcal{S}_N$ and $\mathcal{B}_N$ is a function which assigns a subset of $\{L_v(E) : E \in sub(T_{N_s})\}$ to every state s of T_N (T_{N_s} is T_N with s as its initial state). If s is reachable by a visible string $\overline{\sigma}$ and $B \in \mathcal{B}_N(s)$, then it means that $(\overline{\sigma}, B) \in PF(\mathcal{S}_N)$. The function $\mathcal{B}$ can be defined formally as follows:

$$\mathcal{B}_N(c_p) = \bigcup_{s_g \in reach(Proj_I^{-1}(c_p))} \{L_v(\mathcal{S}_{N_{(s_g)}})\}$$

where $\mathcal{S}_{N_{(s_g)}}$ is $\mathcal{S}_N$ with s_g as its initial state. If $(T_N, \mathcal{B}_N) = (T_M, \mathcal{B}_M)$ for two distinct natural numbers N, M, then $\mathcal{S}_N$ and $\mathcal{S}_M$ are possible-futures equivalent. The finite number of such pairs $(T_N, \mathcal{B}_N)$ on the other hand results in a finite number of possible-futures equivalence classes; the suggested procedure is then guaranteed to terminate.

Examples exist where the number of $\equiv_{wb}$ classes of ring segments is infinite, and consequently $\mathcal{PROC}_{wb}$ never terminates. Consider the template shown in the following figure. We will show that $S_N \not\equiv_{wb} S_M$ where N, M are any two distinct natural numbers.

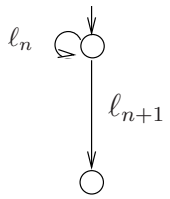


Figure 2.8: template T

Let's show the global state of a segment S_N with a sequence $s_1 s_2 \cdots s_N$ where $s_i \in \{0, 1\}$ represents the state of the i^{th} process in S_N. Obviously, the initial state of S_N is a sequence of zeros of length N: $00 \cdots 0$. According to the template, one process can move from state 0 to state 1 if its right-hand neighbor is in state 0, and the rightmost process can always do so. Therefore, S_N can go to the state $00 \cdots 01$ from its initial state by performing a

visible action ℓ_1. Process N will stay in state 1 after that, and thus process $N-1$ will be stuck in state 0 forever. The reached process is weakly bisimilar to a segment of size $N-2$ where the visible action of the rightmost process is hidden: $S_{N-2}\backslash\{\ell_1\}$. Another possibility is that process $N-1$ moves to state 1 first, and then process N moves to 1 by performing ℓ_1. This time the reached process is weakly bisimilar to $S_{N-3}\backslash\{\ell_1\}$, and so on.

Let $A_i = S_i\backslash\{\ell_1\}$. In particular, define A_0 to be a null process – a process with no transition. Process A_1 can perform a visible action ℓ_0, and evolve to A_0. It can also be shown that $A_i \stackrel{\epsilon}{\Longrightarrow} A_{i-2}$ for $i \geq 2$.

The processes $A_{N-2}, A_{N-3}, \cdots A_0$ are all reachable from S_N. We will show that A_i's are not weakly bisimilar, and therefore, two segments of different size are not weakly bisimlar because as the size of the segment grows the number of weak bisimulation equivalence classes of reachable processes from S_N grows as well; but weak bisimilarity of two segments of distinct size implies the existence of a finite bound on the number of such equivalence classes which is a contradiction.

In order to show that $A_i \not\equiv_{wb} A_j$ where $i \neq j$, we first need to mention some results from [5] regarding games on processes.

Let E_0, F_0 be two given processes. An interactive game $G(E_0, F_0)$ is defined between two players R (the refuter) and V (the verifier). A play of this game is a sequence of pairs of processes $(E_0, F_0)(E_1, F_1)\cdots(E_i, F_i)\cdots$ which is constructed as follows: starting from the pair (E_0, F_0), R starts the game by picking one of the processes in the pair, and performing a visible or an ϵ transition. It could for example choose E_0 and perform a transition t to reach a new process E_1. Now V picks the other process from the pair F_0 and performs a transition t' with the same label as t to reach F_1. This results in a new pair (E_1, F_1) which will be added to the sequence of the game. Now the players repeat the same game with the new pair (E_1, F_1). They continue this game until the verifier cannot match a transition which results in R to win the game. If the game is infinite, meaning that the verifier can match all the transitions of the refuter, V is called the winner. Note

that every process can always perform an ϵ transition, therefore the refuter always has a transition to perform.

A player may have a set of rules which she obeys in the game. This set of rules is known as the strategy of that player. The strategy of a refuter may only depend on the last pair of the processes achieved in the game which is known as a history-free strategy, or it may depend on the sequence of the pairs in the game. Similarly, the strategy of a verifier is called history-free if it only depends on the last pair of the processes in the game, and the last transition that the refuter made.

Theorem 9 *For every two processes E, F, either the refuter or the verifier has a history-free winning strategy for the game $G(E, F)$.*

If the verifier has a strategy to win the game $G(E, F)$, then E and F are called *game equivalent*. It can easily be shown that game equivalence is an equivalence relation. The following theorem shows that the game equivalence and weak bisimulation equivalence are identical.

Theorem 10 *Two given processes are weakly bisimilar iff they are game equivalent.*

Now we return to our problem of showing that $A_i \not\equiv_{wb} A_j$ where $i > j \geq 0$. We show that A_i, A_j, $i > j \geq 0$, are not game equivalent by using induction on i. The basic case when $i = 1, 2$ can be proved easily. The possible cases are: $G(A_2, A_1)$, $G(A_2, A_0)$ and $G(A_1, A_0)$ which we discuss them separately.

- in the game $G(A_1, A_0)$ ($G(A_2, A_0)$), let the refuter choose $A_1 (A_2) \stackrel{\ell_0}{\Longrightarrow} A_0$. The verifier cannot match this action, and therefore, the refuter wins the game.

- in the game $G(A_2, A_1)$, let the refuter choose $A_2 \stackrel{\epsilon}{\Longrightarrow} A_0$, then the verifier can only choose $A_1 \stackrel{\epsilon}{\Longrightarrow} A_1$; so the new pair is (A_0, A_1), and the rest of the strategy will be similar to first case.

Now assume that for a natural number $N \geq 2$, we have a strategy for the refuter to win the game $G(A_N, A_M)$ where $0 \leq M < N$, then we propose a strategy for the refuter to win the game $G(A_{N+1}, A_M)$ where $0 \leq M < N+1$. Three different cases can happen:

- If $M = N$, the refuter chooses the transition $A_{N+1} \stackrel{\epsilon}{\Longrightarrow} A_{N-1}$ and whatever ϵ transition that the verifier chooses for the process A_M, it will reach some A_i where $0 \leq i < n-1$ or $i = N$; so the new pair is (A_{N-1}, A_i), and from there the refuter follows the strategy of the game $G(A_{N-1}, A_i)$ which is known according to our assumption.

- If $M = N-1$, the refuter chooses the transition $A_{N+1} \stackrel{\epsilon}{\Longrightarrow} A_{N-2}$, and whatever ϵ transition that the verifier chooses for the process A_M, it will reach some A_i where $0 \leq i < N-2$ or $i = N-1$; so the new pair is (A_{N-2}, A_i), and the winning strategy of the refuter for the game $G(A_{N-2}, A_i)$ is known.

- If $M \leq N-2$, the refuter chooses the transition $A_{N+1} \stackrel{\epsilon}{\Longrightarrow} A_{N-1}$ and whatever ϵ transition that the verifier chooses for the process A_M, it will reach some A_i where $0 \leq i \leq N-2$; so the new pair is (A_{N-1}, A_i), and the rest of the strategy is known.

Consequently, no two segments of different sizes are game equivalent, and therefore, no two segments of different sizes are weakly bisimilar.

2.4.2 Shuffled processes

A *shuffled* process over the template alphabet Σ_n is one which is weakly bisimilar to the shuffle product of processes P_ℓ and P_r whose respective sets of visible actions are disjoint sets Σ_ℓ and Σ_r. In this subsection, we will show that if a ring segment (S'_N) is a shuffled process, then the procedure for checking the ring segments for weak bisimulation is guaranteed to terminate. Given a process P over Σ_n, we can easily check whether there exist processes P_ℓ, P_r such that $P \equiv_{wb} P_\ell \circledcirc P_r$, as follows: if such processes P_ℓ, P_r

exist, then $P\backslash\Sigma_\ell \equiv_{wb} P_\ell\backslash\Sigma_\ell \circledcirc P_r \equiv_{wb} P_r$ and $P\backslash\Sigma_r \equiv_{wb} P_\ell \circledcirc P_r\backslash\Sigma_r \equiv_{wb} P_\ell$. Consequently, $P \equiv_{wb} P\backslash\Sigma_r \circledcirc P\backslash\Sigma_\ell$. So we only need to check whether $P \equiv_{wb} P\backslash\Sigma_r \circledcirc P\backslash\Sigma_\ell$: if that holds, then $P\backslash\Sigma_r$ and $P\backslash\Sigma_\ell$ are our desired processes (up to weak bisimilarity); otherwise such processes do not exist. Before stating the main theorem, we need to prove the following lemma:

Lemma 2 *If $P_{\ell 1} \circledcirc P_{r1} \equiv_{wb} P_{\ell 2} \circledcirc P_{r2}$ where $P_{\ell 1}, P_{\ell 2}$ (and respectively P_{r1}, P_{r2}) have the same set of visible actions, then $P_{\ell 1} \equiv_{wb} P_{\ell 2}$ and $P_{r1} \equiv_{wb} P_{r2}$.*

Proof: This can be shown by similar reasoning as in the previous paragraph. □

Theorem 11 *If $S'_N = P \odot P \odot ... \odot P$ is weakly bisimilar to $P_\ell \circledcirc P_r$, then $S'_N \equiv_{wb} S'_{N+1}$.*

Proof: We know that $S'_N \odot P \equiv_{wb} P \odot S'_N$; therefore, $(P_\ell \circledcirc P_r) \odot P \equiv_{wb} P \odot (P_\ell \circledcirc P_r)$; consequently, $P_\ell \circledcirc (P_r \odot P) \equiv_{wb} (P \odot P_\ell) \circledcirc P_r$. According to the previous theorem, $P_\ell \equiv_{wb} P \odot P_\ell$ and $P_r \odot P \equiv_{wb} P_r$ which implies $S'_{N+1} = S'_N \odot P \equiv_{wb} P_\ell \circledcirc P_r \equiv_{wb} S'_N$. □

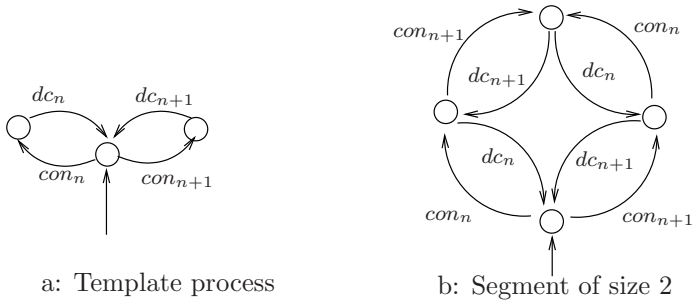
In the sequel, we will provide two examples of potential applications of this theorem.

Con-discon machine

Consider a ring of processes where each process can connect only to one of its immediate neighbors at a time, and then it needs to disconnect in order to establish a new connection. The template process of such a ring is shown in Figure 2.9.a. The ring segment of size 2 (S'_2) is weakly bisimilar to the one depicted in Figure 2.9.b.

It is easy to show that $P \odot P$ is weakly bisimilar to $P_\ell \circledcirc P_r$ where P_ℓ and P_r are shown in Figure 2.10.

Consequently, a ring segment of size two is weakly bisimilar to any ring segment of greater size.



a: Template process b: Segment of size 2

Figure 2.9: Con-discon example

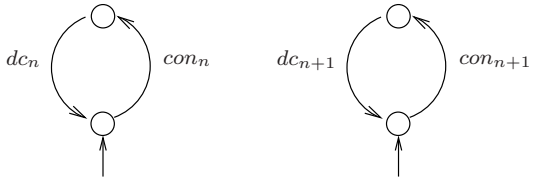


Figure 2.10: Processes P_ℓ and P_r

Token passing template

Define the template T of a ring network as follows:

- the set of states: $Q \times \{0, 1, ..., m\}$;
- the set of visible actions: Σ_n includes actions of the form $(a, num)_n$ or $(a, num)_{n+1}$ where $num \in \{2, .., m\}$;
- the initial state: $(q_0, 0)$;
- in the transition relation R, every transition of the form $((q_1, x), \tau, (q_2, y))$ satisfies: $y = x \neq 0$ or $(q_2, y) = (q_0, 1)$;
  every transition of the form $((q_1, x), (a, num)_{n+1}, (q_2, y))$ (send a token to the right-hand neighbor) satisfies: $x \neq m$, $num = x + 1$ and $y = x$;

and every transition of the form $((q_1, x), (a, num)_n, (q_2, y))$ (receive a token from left) satisfies:

- new token : if $num \neq x$, then $x \neq 1$ and $(q_2, y) = (q_0, num)$;
- old token : if $num = x$, then $x \neq 1$ and $y = num$.

Every state of a process in the ring is of the form (q, x) where $0 \leq x \leq m$ shows the *mode* of that state. An *active* state is one whose mode is 1. The mode of a process is the mode of its current state. Initially, all the processes are in mode 0, and cannot communicate with their neighbors. But, at any state, any process in the ring can set its mode to 1 (become active) by performing an internal action, and then communicate with its right-hand neighbors. It is useful to think of this communication as a token being passed. Every time two processes synchronize on an action, the mode of the right process is set to the mode of the left increased by 1. A token's value num is not allowed to exceed m. So the active process sets the mode of its immediate right-hand neighbor to 2 in their first communication, and that neighbor sets the mode of its own right-hand neighbor to 3 and so on until the the mode of the m^{th} process is set. In this way, every communication within the ring is limited to m processes, including an active process and its $m - 1$ right-hand neighbors.

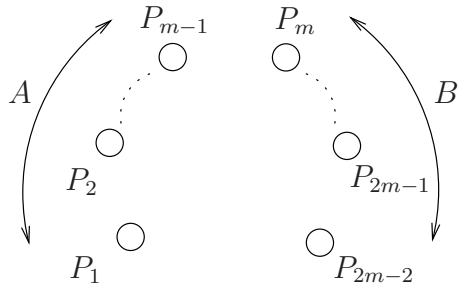


Figure 2.11: A ring segment of size $2m - 2$

By the above argument, it is not hard to see that a ring segment of size $2m-2$ is a shuffled process. In Fact, it can be shown that $P_\ell = S'_{m-1}\backslash\Sigma_r$ and $P_r = S'_{m-1}\backslash\Sigma_\ell$. Figure 2.11 shows a segment of size $2m-2$. The first $m-1$ processes are marked as group A processes, and the second $m-1$ processes are marked as group B. We define a weak bisimulation relation R between the states of S_{2m-2} and $P_\ell \circledcirc P_r$ as follows: one state of S_N is related to a state of $P_\ell \circledcirc P_r$ iff the following two conditions hold:

- for any $1 \leq j < m$ the j^{th} coordinates of the both states are the same or the i^{th} coordinates are active for some $i \leq j$;
- for any $0 \leq j < m-1$, if the mode of $(m+j)^{th}$ coordinate of one of the states is between 1 and $j+1$, then the $(m+j)^{th}$ coordinates of both of the states are the same or their $(m+i)^{th}$ coordinates are active for some $j < i < m-1$.

One can easily show that R is a weak bisimulation relation since every action taken from a state of S_{2m-2} can be mimicked from a related state of $P_\ell \circledcirc P_r$, and vice versa.

In this chapter, we focused on ring networks consisting of an arbitrary number of processes. An algebra-theoretic approach was taken to compare rings as well as ring segments of different size, and check whether they fall into a finite number of equivalence classes. The equivalence relation is chosen based on the type of modal property or temporal property that the network is being verified against. A few of interesting problems on ring networks such as blocking detection are shown to be undecidable. On the other hand, a semi-decidable procedure is introduced as a solution; however, this procedure is not guaranteed to terminate. It is then shown that if the template process of a ring network satisfies some sufficient conditions, then the termination of the above procedure is guaranteed.

Chapter 3

Infinite State Modelings and Fully-Connected Networks

Choosing the right modelling tool is a major step in analysis of any real-world system. Petri nets are a graphical and mathematical tool for modelling systems which are characterized as finite- or infinite-state, concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic [24]. They have proved themselves as a powerful tool for modelling, control, and analysis of communication and manufacturing systems, and many of their interesting properties are decidable. For surveys of Petri nets, their properties, and the complexity of their problems, see [24, 31, 32, 33]. Model-checking of Petri nets is the main focus of this chapter. Current results on model-checking of branching time logics on Petri nets are not very positive. In fact Esparza shows in [33] that model checking of VBPP's (a very weak class of Petri nets) against an action-based modal μ calculus as well as a very weak branching time logic UB^{-} is undecidable. The results are more promising for linear time temporal logic.

In this chapter, we start by first stating some preliminaries of Petri nets. Some fundamental and interesting problems on Petri nets including reachability problem, boundedness

problem, coverability problem, non-termination problem, and fair non-termination problem are discussed. We will also introduce a method of solving the fair non-termination problem by means of the coverability tree. Next, we survey the existing literature on model-checking of Petri nets, and introduce our decidability results on model-checking of a large fragment of linear temporal logic with marking predicates. Finally we will show how these results can be applied in rigorous analysis of some examples of infinite-state systems and parameterized systems.

3.1 Petri Net Preliminaries

3.1.1 Petri Net Models

A labelled Petri net $\mathcal{P}$ is a tuple of the form $(P, T, W, \Sigma, \rho, M_0)$ where P is a set of places, T is a set of transitions, $W : ((P \times T) \cup (T \times P)) \to \mathbb{N}$ is a weight function, Σ is a set of action labels, $\rho : T \to \Sigma$ is a labelling function which assigns to every transition an action label. A marking $M : P \to \mathbb{N}$ is a function which assigns to every place a natural number – $M(p)$ represents the number of tokens in place p in marking M; M_0 is the initial marking of $\mathcal{P}$. A non-labelled Petri net is one with no set of action labels and labelling function, denoted by $(P, T, W, -, -, M_0)$. A transition t is called enabled at a marking M if $M(p) \geq W(p, t)$ for every place $p \in P$; then t can be fired at M and reach a new marking M' where $M'(p) = M(p) - W(p, t) + W(t, p)$. We denote this transition by $M \xrightarrow{t} M'$. Given a sequence of transitions $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} M_n$, sometimes we omit the intermediate markings, and write $M_1 \xrightarrow{\sigma} M_n$ where $\sigma = t_1 t_2 \cdots t_{n-1}$; marking M_n is then said to be reachable from M_1. Every marking is reachable from itself. By $\mathcal{R}(\mathcal{P})$, we denote the set of all markings which are reachable from the initial marking M_0 of $\mathcal{P}$. The *reachability problem* (RP) is to decide for a given Petri net $\mathcal{P}$, and a marking M whether $M \in \mathcal{R}(\mathcal{P})$.

A place p of $\mathcal{P}$ is said to be bounded if there exists a non-negative constant c such that

for every marking M in $\mathcal{R}(\mathcal{P})$, we have $M(p) \preceq c$. A Petri net $\mathcal{P}$ is said to be bounded if all of its places are bounded. The *boundedness problem* (BP) is to decide whether a given Petri net $\mathcal{P}$ (or a particular place p of $\mathcal{P}$) is bounded.

An infinite firing sequence of $\mathcal{P}$ is of the form $M_0, t_1, M_1, \cdots$ where M_0 is the initial marking of $\mathcal{P}$, and $M_{i-1} \xrightarrow{t_i} M_i$ for $1 \preceq i$. A firing sequence is sometimes referred to as a computation. We denote by $\mathcal{C}_\omega(\mathcal{P})$ the set of all infinite firing sequences of $\mathcal{P}$.

We extend the labelling function ρ to a sequence of transitions in a natural way: $\rho(t\sigma) = \rho(t)\rho(\sigma)$. For the sake of notation simplicity, we also use $\rho(c)$ to denote the actions sequence corresponding to the transitions of the infinite computation c. More formally, $\rho(c) = \rho(t_1)\rho(t_2)\cdots$ where $c = M_0, t_1, M_1, \cdots$. Also define $inf(c)$ as the set of transitions which occur in c infinitely often.

The language of $\mathcal{P}$ is the set of all action sequences corresponding to finite firing sequences of $\mathcal{P}$; more formally, $\mathcal{L}(\mathcal{P}) = \{\rho(\sigma) | M_0 \xrightarrow{\sigma} M\}$.

The ω language $\mathcal{L}_\omega(\mathcal{P})$ of $\mathcal{P}$ is defined as the set of all action sequences corresponding to infinite firing sequences of $\mathcal{P}$. More formally, $\mathcal{L}_\omega(\mathcal{P}) = \{\rho(c) | c \in \mathcal{C}_\omega(\mathcal{P})\}$.

A Büchi net $\mathcal{B}$ is a pair $(\mathcal{P}, \mathcal{T})$ where $\mathcal{P} = (P, T, W, \Sigma, \rho, M_0)$ is a Petri net, and $\mathcal{T}$ is a subset of T known as the set of *final* transitions. The language $\mathcal{L}_\omega(\mathcal{B})$ of $\mathcal{B}$ is defined as the set of all action sequences corresponding to infinite computations c of $\mathcal{P}$ for which $inf(c) \cap \mathcal{T} \neq \emptyset$. More formally, $\mathcal{L}_\omega(\mathcal{B}) = \{\rho(c) | c \in \mathcal{C}_\omega(\mathcal{P}) \ \& \ inf(c) \cap \mathcal{T} \neq \emptyset\}$. In the special case when $\mathcal{T} = T$, we have $\mathcal{L}_\omega(\mathcal{B}) = \mathcal{L}_\omega(\mathcal{P})$.

One important feature of Petri nets is their ability to model unlimited, and limited-size buffers, and that makes them suitable for modelling of manufacturing systems. Figure 3.1 [26], depicts an example of a non-labelled Petri net which models a Producer-Consumer system. Place p_1 represents the initial state of the producer (left block). Two transitions t_1, t_2 need to be fired in a sequence to produce a part. A token is then added to place p_5 to represent the new part stored in the buffer. This buffer is assumed to be of infinite size. The right block of the model represents a consumer which takes token from p_5 upon the

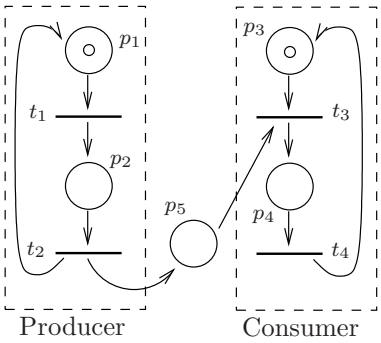


Figure 3.1: Producer and Consumer with an infinite size Buffer

execution of t_3. In a real model, one may need to consider the buffer to be of limited size (say 3). In order to model that, we can add another place p_6 to the model whose tokens represent the number of free places in the buffer. Therefore, a part can be added to the buffer if it has at least one free place; i.e. there is at least one token left in p_6. On the other hand, the consumer can remove a part from the buffer if it has least one part in it; i.e., there is at least one token in p_5.

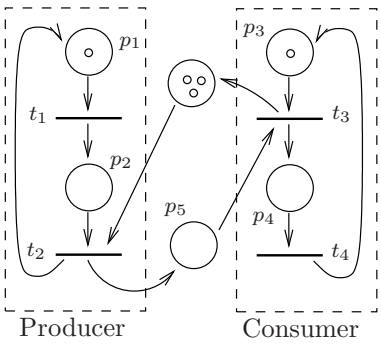


Figure 3.2: Producer and Consumer with a Buffer of size 3

Figure 3.3 [26], depicts a mutual exclusion example where two subsystems are sharing a common resource represented by place p_4. This place needs to be marked (resource avail-

able) in order for any of the subsystems 1,2 to fire t_2, t_5 respectively. After the subsystems finish their job with the resource, they return it by execution of t_3, t_6 respectively. This structure guarantees that the resource is only held by one subsystem at a time.

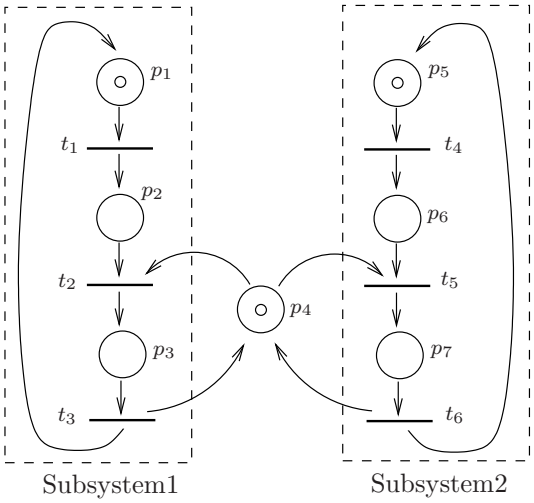


Figure 3.3: Mutual Exclusion example

3.1.2 Reachability Tree and Coverability Tree

The reachability tree of a labelled Petri net $\mathcal{P}$ is a labelled transition system $E = (\Sigma, Q, T, q_0)$ where Σ is the set of action labels of $\mathcal{P}$; Q is the set of reachable markings of $\mathcal{P}$, $T = \{(M, \alpha, M') | M \xrightarrow{t} M' \text{ and } \rho(t) = \alpha\}$, and q_0 is the initial marking of $\mathcal{P}$. If the reachability tree of a Petri net is finite, then the common algorithms of the finite state systems can be applied to answer many of the interesting questions on Petri nets such as reachability of a particular marking, or liveness of a particular transition; however the set of reachable markings may be infinite resulting in an infinite state reachability tree. The coverability tree of a Petri net, on the other hand, is a more abstract version of the reachability tree which enumerates the covers of reachable markings instead of the reachable markings

themselves. In other words, one can decide whether a cover of a particular marking of a Petri net is reachable by looking at its corresponding coverability tree. This tree, which is guaranteed to be finite, can be useful to answer many interesting questions including BP. In order to define the construction steps of the coverability tree, we need to define a new symbol ω with the following properties: $\omega > n$, $\omega + n = \omega - n = \omega$ and $\omega > \omega$ for any integer n – one can think of ω as infinity. For two given extended markings $M1, M_2$, we say M_1 is covered by M_2, $M_1 \leq M_2$, iff $M_1(p) \leq M_2(p)$ for every place $p \in P$. Furthermore, we say $M_1 < M_2$ iff $M_1 \leq M_2$ and $M_1 \neq M_2$. The construction steps of the coverability tree can then be defined as follows [25, 24]:

1) Create the root note M_0 and mark it "new".
2) While exists a node with a "new" tag do the following steps.
   2.1) Select a new node M.
   2.2) If M has an identical predecessor (defined below), then mark M "old" and go to 2.
   2.3) If no transition is enabled at M, mark M "dead" and go to 2.
   2.4) Do the following steps for every enabled transition t at M.
      2.4.1) Obtain the marking M' such that $M \xrightarrow{t} M'$.
      2.4.2) If M' has a predecessor $M'' < M'$, replace $M'(p)$ by ω
      for each place p such that $M''(p) < M'(p)$.
      2.4.3) Create a node M' and mark it "new"; connect M to M' with an arc labelled t.

A node marking M' is called a *predecessor* of another node marking M if M' is on the path from the root to M.

Theorem 12 *[25] The coverability tree of every given Petri net is finite.*

To prove this theorem, we first need to show that: every infinite sequence $\pi = a_1, a_2, a_3, \cdots$ of elements of $(\mathbb{N} \cup \{\omega\})^r$ has an infinite subsequence $\pi' = a_{i_1}, a_{i_2}, a_{i_3}, \cdots$ such that $a_{i_1} \leq a_{i_2} \leq a_{i_3} \leq \cdots$. The proof is very simple. One can first extract a subsequence

of π which is non-decreasing in its first coordinates; then extract a subsequence of the obtained sequence which is non-decreasing in its second coordinate, and so forth.

Now suppose that the coverability tree of a given Petri net has an infinite branch extending from its root: $M_0 M_1 M_2 \cdots$. According to what we showed earlier, there exists a non-decreasing subsequence of this marking sequence: $M_{i_1} M_{i_2} M_{i_3} \cdots$. Note that the markings of this subsequence cannot be identical because according to the coverability tree construction procedure, that would result in a finite path. Therefore, $M_{i_1} < M_{i_2} < M_{i_3} < \cdots$. Thus, each marking has to have at least one more ω coordinate than its previous one. However, this is impossible since the number of coordinates is finite. □

Corollary 1 *A Petri net $\mathcal{P}$ is unbounded iff it has a firing sequence $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1} M + L$ where $L > 0$.*

Let's first assume that such a token generator exists, then $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1^n} M + n * L$ is also an acceptable firing sequence of $\mathcal{P}$. Since L is a non-negative integer vector, the reachable marking $M + n * L$ can be made arbitrarily large in the positive coordinates of L; therefore, $\mathcal{P}$ is unbounded.

Now suppose that $\mathcal{P}$ is unbounded. Construct its corresponding coverability tree. If ω does not appear in the tree, then the set of reachable states of $\mathcal{P}$ is finite; therefore, $\mathcal{P}$ is bounded which contradicts our assumption. Existence of ω in the tree, on the other hand, implies the existence of a firing sequence of the form $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1} M'$ where $M' > M$. That completes the proof. □

Theorem 13 *[25] For a given marking M of a Petri net $\mathcal{P}$ the following statement holds: a marking M_r is reachable from the initial marking of $\mathcal{P}$ such that $M_r \geq M$ iff a node marking M_n of $\mathcal{P}$'s coverability tree exists such that $M_n \geq M$.*

According to the above theorem, one can decide whether a cover of a marking is reachable by means of the coverability tree. This problem is known as the *coverability problem*

(CP). Also note that if some coordinates of a node marking in a coverability tree are ω, then those coordinates can grow arbitrarily large by repeating the transition sequences resulting in them. Therefore, one can decide whether a Petri net (or a particular place of that Petri net) is unbounded by means of its coverability tree. In other words, BP can be solved using the coverability tree construction. There are several other problems that can be solved using the coverability tree. For instance,

Non-termination problem (NTP): Decide whether a given Petri net has an infinite firing sequence.

If ω appears anywhere in the coverability tree, that means that the Petri net has a token generator, and therefore, has an infinite path. Otherwise, the coverability tree is a finite-state reachability tree. Now we can look at the leaves of the tree, and check whether exists any leaf whose tag is "old". That implies that the Petri net has a firing sequence of the form $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma_1} M$, and consequently, has an infinite path. Otherwise, all the leaves are labelled "dead" and the Petri net does not have an infinite path. A more general case of NTP is as follows:

Fair non-termination problem (FNTP): For a given Petri net $\mathcal{P}$ and a finite subset of transitions $X \subseteq T$, decide whether $\mathcal{P}$ has an infinite computation c such that $inf(c) \cap X \neq \emptyset$.

We know from [37], that FNTP is PTIME equivalent to BP. However, we propose an algorithm for deciding this problem by means of the coverability tree construction. This problem is not as straight forward as NTP. Consider the two Petri nets in Figure 3.4. They both have the same coverability trees as depicted in the Figure; however, the right-hand one has an infinite path which fires t_3 infinitely often, but the left-hand one does not have such a path.

Let $\mathcal{M}_{max}$ be the set of all node markings $M \in (\mathbb{N} \cup \{\omega\})^r$ of the coverability tree that are not covered by any other one. In fact, $\mathcal{M}_{max}$ is the set of the maximal markings of the

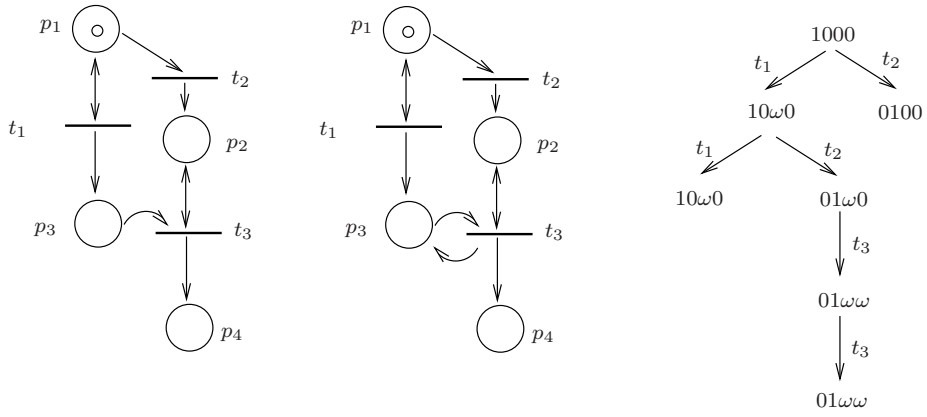


Figure 3.4: different Petri nets with the same coverability tree

coverability tree. Now, consider the following simple lemma from [37].

Lemma 3 *A given Petri net $\mathcal{P}$ has an infinite path c such that $inf(c) \cap X \neq \emptyset$, $X \subseteq T$, iff $\mathcal{P}$ has a firing sequence of the form $M_0 \xrightarrow{\sigma} M \xrightarrow{\sigma'} M'$ where $M' \geq M$ and $T_{\sigma'} \cap X \neq \emptyset$ where $T_{\sigma'}$ is the set of transitions in σ'.*

Therefore, if $\mathcal{P}$ has such an infinite path, there exists $M \in \mathcal{R}(\mathcal{P})$ such that $M \xrightarrow{\sigma'} M'$, $M' \geq M$, $T_{\sigma'} \cap X \neq \emptyset$. Obviously, every cover of M' (and therefore of M) in $\mathcal{M}_{max}$ has this property as well. On the other hand, it can be shown that if a marking in $\mathcal{M}_{max}$ has this property, then so does a reachable marking of $\mathcal{P}$. Therefore, it suffices to check whether there exists a marking $M \in \mathcal{M}_{max}$ such that $M \xrightarrow{\sigma} M$; if so, $\mathcal{P}$ has a fair infinite path. Furthermore, suppose that $M \xrightarrow{\sigma} M$ holds for some $M \in \mathcal{M}_{max}$; therefore, we have $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \cdots M_n$ where $M_1 = M_n = M$. It is easy to see that every marking M_i, $1 \leq i \leq n$, has the above property as well; i.e., there exists a sequence of transitions σ_i such that $M_i \xrightarrow{\sigma_i} M_i$ for every $1 \leq i \leq n$; Hence, $M_1^{max} \xrightarrow{t_1} M_2^{max} \xrightarrow{t_2} \cdots M_n^{max}$ also holds where M_i^{max} is the maximal node marking in $\mathcal{M}_{max}$ which covers M_i. Consequently, we can only search for self-covering loops occurring among the markings of $\mathcal{M}_{max}$.

Considering the above fact, a finite state machine (FSM) $\Pi = (\Sigma, Q, R)$ can be constructed where the set of transition labels $\Sigma = T$; the set of states $Q = \mathcal{M}_{max}$; and the transition relation $R \subseteq Q \times \Sigma \times Q$ is defined as $\{(M_1, t, M_2) | M_1 \xrightarrow{t} M_2\}$. A loop L of Π is defined as a sequence of markings (states of Π) $M_0, M_1, \cdots M_n$ where $(M_i, t_i, M_{i+1 \bmod n}) \in R$, $0 \leq i \leq n$. The weight $\nu(L)$ of L is defied as the sum of all the integer vectors corresponding to the transitions of L: $\Sigma_{i=0}^{n} t_i$. We also denote by $T(L)$ the set of transitions t_i of L. The loop L is called "simple" if $M_i = M_j$ implies $i = j$. In other words, in a simple loop, the markings are not repetitive. Two simple loops are called "interconnected" when they have at least one common marking. A subset A of simple loops in Π is called interconnected, if members of A can be ordered as $L_1, L_2, \cdots L_p$ such that L_i, L_{i+1} are interconnected for $1 \leq i \leq p-1$.

Lemma 4 *An FSM Π has a loop L where $T(L) \cap X \neq \emptyset$ and $\nu(L) \geq 0$ iff there exist a set of interconnected simple loops $\{L_1, L_2, \cdots L_p\}$, and positive numbers $n_1, n_2, \cdots n_p$ such that $(\bigcup_{i=1}^{p} T(L_i)) \cap X \neq \emptyset$ and $\Sigma_{i=1}^{p} n_i \times \nu(L_i) \geq 0$ holds.*

According to the above lemma, the NTP problem reduces to check whether $\Sigma_{i=1}^{p} n_i \times \nu(L_i) \geq 0$ has a positive solution (refer to [47] for a solution of such linear inequality systems) for a subset $\{L_1, L_2, \cdots L_p\}$ of interconnected simple loops of Π for which $(\bigcup_{i=1}^{p} T(L_i)) \cap X \neq \emptyset$. Consider the Petri net in Figure 3.5 [1]. Places p_1, p_2 act as a sequencer. As long as place p_1 contains a token, transition t_1 is enabled and it can be fired arbitrarily many times, resulting in arbitrarily many tokens in place p_3. At some point, p_1 may pass its token to p_2 by firing t_2. Non-emptiness of p_2 is a sufficient condition for firing transitions t_3 and t_4.

One may wish to know whether $\mathcal{P}$ has an infinite firing sequence which fires t_4 infinitely often. In order to answer that question, we construct the coverability tree, and its corresponding FSM as depicted in Figures 3.6 and 3.7. The set of maximal nodes

[1] Note that a two-headed arrow denotes two arrows with opposite directions.

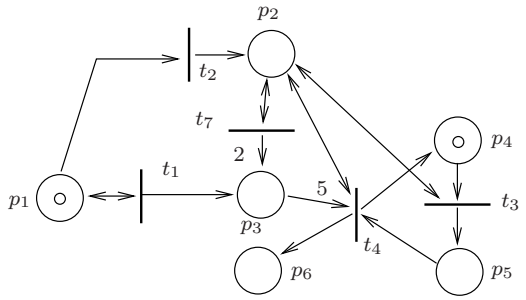


Figure 3.5: Petri Net $\mathcal{P}$

$\mathcal{M}_{max}$ is $\{01\omega10\omega, 01\omega01\omega, 10\omega100\}$ which implies that the FSM has only three states. The simple loops of the FSM are as follows: $L_1 = 01\omega10\omega, 01\omega01\omega$ $L_2 = 01\omega10\omega$ $L_3 = 01\omega01\omega$ $L_4 = 10\omega100$, and their corresponding weights are: $\nu(L_1) = t_3 + t_4 = [0\ 0\ 0\ -1\ 1\ 0]^T + [0\ 0\ -5\ 1\ -1\ 1]^T = [0\ 0\ -5\ 0\ 0\ 1]^T$, $\nu(L_2) = \nu(L_3) = t_7 = [0\ 0\ 2\ 0\ 0\ 0]^T$, $\nu(L_4) = t_1 = [001000]^T$. We have $2 \times \sigma(L_1) + 5 \times \sigma(L_2) = [0\ 0\ 0\ 0\ 0\ 2]$ is a non-negative vector, and $T(L_1) \cup T(L_2) = \{t_3, t_4, t_7\}$; therefore, there exists a firing sequence with infinitely many occurrences of t_4.

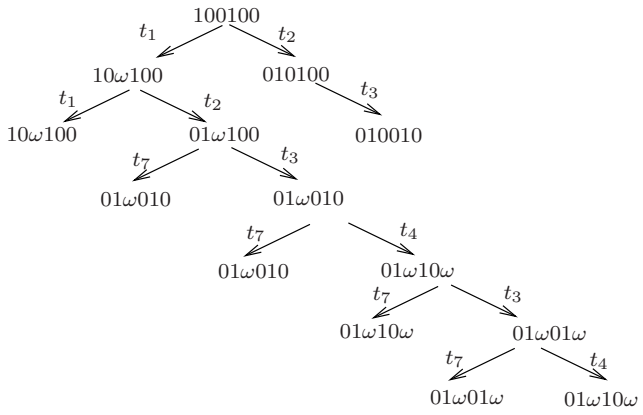


Figure 3.6: Coverability Tree of $\mathcal{P}$

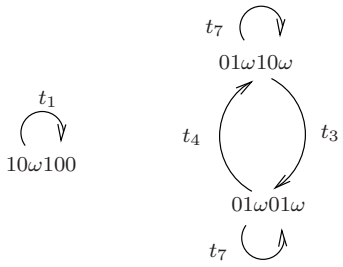


Figure 3.7: FSM corresponding to coverability tree of $\mathcal{P}$

3.1.3 Linear Temporal Logic

In this subsection, we will introduce a linear temporal logic with three different predicates, and the usual temporal operators $G, F, X, U, R, \wedge, \vee, \neg$. The predicate $fi(t)$, also known as a transition predicate, holds at a marking of a computation path when t is the next transition executed from that marking. The other two predicates, known as marking predicates, are employed to express the properties of markings. The predicate $ge(p, k)$ holds at a marking of a computation path when the number of tokens in place p is greater than or equal to the constant k. Another such predicate is $en(t)$ which holds when transition t is enabled. This predicate is also a marking predicate since it can be expressed as a conjunction of $ge(p, k)$ marking predicates. More formally, the syntax and the semantics of this logic are defined below.

Given a Petri net $\mathcal{P} = (P, T, W, \Sigma, \rho, M_0)$, we define the syntax of the linear temporal logic $\mathcal{L}$ inductively as follows:

(a) Every basic predicate $fi(t)$, $ge(p, k)$, $en(t)$ where $p \in P$, $t \in T$, $k \in \mathbb{N}$, is a formula of $\mathcal{L}$;

(b) If φ is a formula of $\mathcal{L}$, then $\neg\varphi$, $X\varphi$ are in $\mathcal{L}$;

(c) If φ_1 and φ_2 are formulas of $\mathcal{L}$, then $\varphi_1 \vee \varphi_2$, $\varphi_1 U \varphi_2$ are in $\mathcal{L}$.

The semantics of this logic is defined inductively for any infinite computation $\pi = M_0, t_1, M_1, \cdots$ and any natural number n as follows:

- $(\pi, n) \models fi(t)$ iff $t_{n+1} = t$
- $(\pi, n) \models ge(p, k)$ iff $M_n(p) \geq k$
- $(\pi, n) \models en(t)$ iff $M_n(p) \geq W(p, t)$ for every $p \in P$
- $(\pi, n) \models \neg\varphi$ iff $\neg((\pi, n) \models \varphi)$
- $(\pi, n) \models \varphi_1 \vee \varphi_2$ iff $(\pi, n) \models \varphi_1 \vee (\pi, n) \models \varphi_2$
- $(\pi, n) \models X\varphi$ iff $(\pi, n+1) \models \varphi$
- $(\pi, n) \models \varphi_1 U \varphi_2$ iff $\exists i \succeq n, \forall j\ n \preceq j \prec i, (\pi, j) \models \varphi_1 \wedge (\pi, i) \models \varphi_2$

Intuitively, a formula φ holds at (π, n) when marking M_n of π satisfies φ. The predicates of the form $ge(p, k)$, $en(t)$ are known as *marking predicate*s since they are interpreted on markings. Note that a predicate $en(t)$ can be written as $\bigwedge_{p \in P} ge(p, W(p, t))$. The predicates of the form $fi(t)$ are known as *transition predicate*s.

As usual, we use the abbreviations $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$; $\varphi_1 R \varphi_2 = \neg(\neg\varphi_1 U \neg\varphi_2)$; $F\varphi = TrueU\varphi$; $G\varphi = FalseR\varphi$.

We say a Petri net $\mathcal{P}$ *existentially* satisfies a formula φ of logic $\mathcal{L}$, $\mathcal{P} \models_\exists \varphi$, iff $\exists \pi \in \mathcal{C}_\omega(\mathcal{P}), (\pi, 0) \models \varphi$. Similarly, we say $\mathcal{P}$ *globally* satisfies φ, $\mathcal{P} \models_\forall \varphi$, iff $\forall \pi \in \mathcal{C}_\omega(\mathcal{P}), (\pi, 0) \models \varphi$.

3.1.4 Product of Petri Nets

Given two Petri nets $\mathcal{P}_1 = (P_1, T_1, W_1, \Sigma, \rho_1, M_{01})$ and $\mathcal{P}_2 = (P_2, T_2, W_2, \Sigma, \rho_2, M_{02})$, define their synchronous product $\mathcal{P}_1 \times \mathcal{P}_2$ as a Petri net $(P, T, W, \Sigma, \rho, M_0)$ where

$$P := P_1 \uplus P_2$$

$$T := \{(t_1, t_2) | t_1 \in T_1, t_2 \in T_2, \rho_1(t_1) = \rho_2(t_2)\}$$

$$W(p, (t_1, t_2)) := \begin{cases} W_1(p, t_1) & \text{if } p \in P_1 \\ W_2(p, t_2) & \text{if } p \in P_2 \end{cases}$$

$$W((t_1, t_2), p) := \begin{cases} W_1(t_1, p) & \text{if } p \in P_1 \\ W_2(t_2, p) & \text{if } p \in P_2 \end{cases}$$

$$\rho((t_1, t_2)) := \rho_1(t_1) = \rho_2(t_2)$$

$$M_0(p) := \begin{cases} M_{01}(p) & \text{if } p \in P_1 \\ M_{02}(p) & \text{if } p \in P_2 \end{cases}$$

Product of a Petri net $\mathcal{P}_1$ and a Büchi net $(\mathcal{P}_2, \mathcal{T}_2)$ is a Büchi net $(\mathcal{P}, \mathcal{T})$ where $\mathcal{P} = \mathcal{P}_1 \times \mathcal{P}_2$ and $\mathcal{T} = \{(t_1, t_2) | (t_1, t_2) \in T \ \& \ t_2 \in \mathcal{T}_2\}$.

Proposition 10 *Given two Petri nets $\mathcal{P}_1, \mathcal{P}_2$, and a Büchi net $\mathcal{B}$, we have $\mathcal{L}_\omega(\mathcal{P}_1 \times \mathcal{P}_2) = \mathcal{L}_\omega(\mathcal{P}_1) \cap \mathcal{L}_\omega(\mathcal{P}_2)$, $\mathcal{L}_\omega(\mathcal{P}_1 \times \mathcal{B}) = \mathcal{L}_\omega(\mathcal{P}_1) \cap \mathcal{L}_\omega(\mathcal{B})$.*

3.1.5 Variants of Ordinary Petri nets

In this section, we briefly talk about two variants of ordinary Petri nets, and compare their computational power with other mathematical modelling tools.

Petri nets with inhibitors: the ordinary Petri nets, as defined earlier, cannot test whether a place has no token in it. Petri nets with inhibitors add this capability to ordinary Petri nets. An inhibitor is an arc connecting a place p to a transition t, and has a small circle instead of an arrow at its terminating point where it connects to t [24]. The role of the inhibitor is disable the transition t if there exists any token in p; i.e., the emptiness of place p is a precondition for firing t. It has been shown that adding inhibitors to ordinary petri nets increases their computational power to the level of Turing machines [24]. In the mutual exclusion example discussed earlier (Figure 3.3), none of the two subsystems has

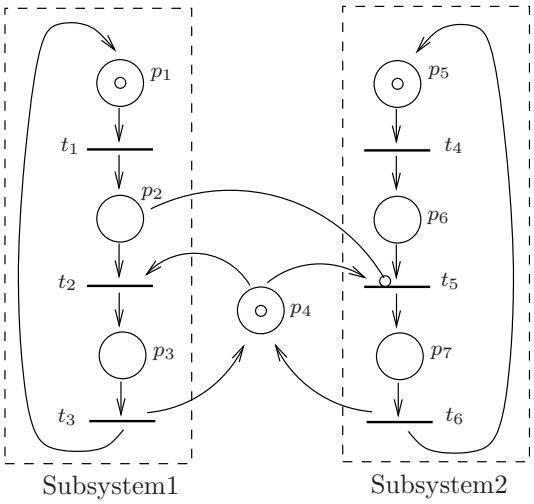


Figure 3.8: Mutual Exclusion example with assigned priority

priority over the other one in taking the resource when they both are requesting it – places p_2, p_6 are marked. One may introduce a new rule such that every time the two subsystems are requesting the available resource, then subsystem 1 takes it. This can be modeled using an inhibitor connecting p_2 to t_5 which disables t_5 every time p_2 is marked (subsystems 1 is requesting the resource). The new model is depicted in Figure 3.8 [26].

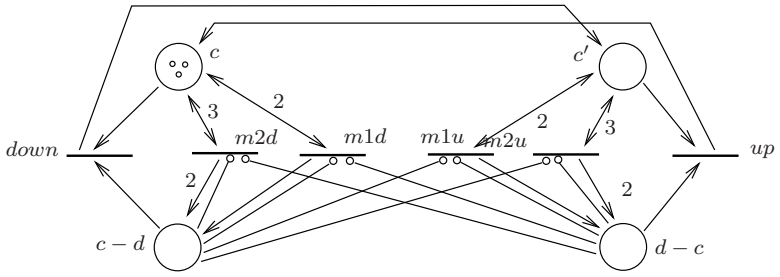


Figure 3.9: Elevator System

To illustrate the role of inhibitors, consider an elevator system with only one lift. The

system has two variables c which denotes the current level, and d which represents the destination level. The current level tracks the destination number as long as they are not equal. If $d > c$, then increase c by one, and if $d < c$, then decrease c by one. After c reaches the same value as d, then d sets to a new value non-deterministically. Figure 3.9 depicts the Petri net model of an elevator with 3 levels. Place c represents the current floor. Place c' is the complement of c; i.e., the total number of tokens in c, c' is 3. $d - c$ denotes the difference between the destination floor and the current floor when $d > c$; otherwise it's set to zero. Similarly, $c - d$ denotes the difference between the current floor and the destination floor when $d < c$; otherwise it's set to zero. After c reaches d, $c - d$ and $d - c$ both become zero, and one of the 4 transitions $m1u, m2u, m1d, m2d$ fires. That would set a new destination value, and disables the 4 transitions until $c-d, d-c$ become zero again. Meanwhile, the current floor c increases (decreases) one by one by firing the transition *up* (*down*) if $d - c$ $(c - d)$ is positive. The elevator is initially in the third floor, and so is the destination floor, $c = d = 3$.

Colored Petri nets: In the ordinary Petri nets, the tokens all have the same type or color, and we can not distinguish among them. In colored Petri nets, on the other hand, we assign types or colors to the tokens. This can be very useful in modelling of systems consisting of isomorphic, but not identical, subsystems. This can be best explained by means of an example. Consider the example of the dining philosophers [27]. The Petri net of one philosopher can be modelled as in Figure 3.10. A philosopher can be is initially in "thinking" state – the place *Think* is marked. He can then evolve to his "eating" state by taking the two chopsticks on his left- and right-hand sides (demonstrated by places C_1, C_2) if they are available; the place *Eat* becomes marked. Otherwise, he has to wait until his neighbor philosophers return the chopsticks. Figure 3.11, depicts a ring consisting of five philosophers, P_1, P_2, P_3, P_4, P_5. All the philosophers are initially in their thinking state. As can be seen, the size of the model grows linearly by the number of the philosophers in the ring. In fact, a ring with n philosophers has $3n$ places and $2n$ transitions ($5n$ nodes),

although all the philosophers have exactly the same structure. Also note that only the neighboring states of a philosopher affect his transitions. In other words, the philosophers can distinguish among themselves, and therefore, the identity of the philosophers needs to be kept in the modelling of such system. This can be achieved by adding colors to the tokens of ordinary Petri nets. A colored Petri net model of such system can be depicted as in Figure 3.12. As can be seen, the number of nodes of this model is only 5 (3 places and 2 transitions). This, therefore, is a more abstract and understandable model of the dining philosophers example. A *type* is defined as a set of values. In this example, $Phil$ is a type with P_1, P_2, P_3, P_4, P_5 values. Every place in the model is assigned a type, and that determines the type (color) of the tokens in that place. Places $Think, Eat$, for instance, are of type $Phil$. The place C is of type $Chop$ which is defined as the set $\{c_{12}, c_{23}, c_{34}, c_{45}, c_{51}\}$. c_{ij} denotes the chopstick which is places between philosophers P_i, P_j.

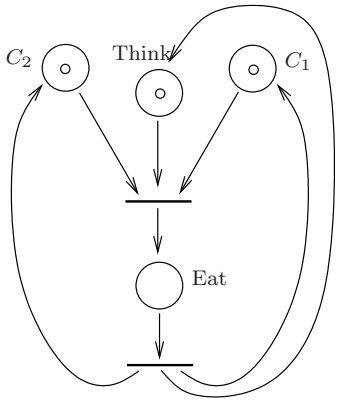


Figure 3.10: One Dining Philosopher

A multi-set m over a given set S is defined as a function from S to the set of natural numbers $\mathbb{N}$. Sometimes, we represent that as a sum $\sum_{s \in S}\ m(s)'s$ [28]. Marking of a place is then defined as a multi-set on the type assigned to that place. For instance, we could define $M(Think) = 1'P_1 + 2'P_2 + 4'P_5$ meaning that place $Think$ has one token of color

P_1, two tokens of color P_2, and 4 tokens of color P_5. The initial marking of a colored Petri net can then be defined by assigning a marking to each one of the places in the net. In our example of dining philosophers, we define $M(Think) = 1'P_1 + 1'P_2 + 1'P_3 + 1'P_4 + 1'P_5$, $M(C) = 1'c_{12} + 1'c_{23} + 1'c_{34} + 1'c_{45} + 1'c_{51}$, and $M(Eat) = \emptyset$ where $\emptyset$ is a multi-set with no elements. Transition t_1 takes a token P_i from $Think$, one token of each $c_{i(i+1)}, c_{(i-1)i}$ from C, and adds a token P_i to Eat where $i \in \{1, 2, \cdots, 5\}$. This is determined by the expressions assigned to the arcs connected to t_1. These arc expressions may have some free variables. For instance, in the arc expression P_i of the arc connecting the place $Think$ to the transition t_1, there is a free variable i which is a natural number between 1 to 5. The action of a transition depends on bindings of the variables in the arcs connected to that transition. For instance, firing of t_1 with the binding $i = 2$ removes one P_2 token from $Think$, and one token of each c_{12}, c_{23} from C, and adds a token P_2 to Eat. Similarly, transition t_2 removes a token P_i from Eat, and adds a token P_i to $Think$, and one token of each $c_{i(i+1)}, c_{(i-1)i}$ to C. Note that by increasing the number of philosophers, only the token types $Phil, Chop$ become larger, but the structure of the net remains unchanged.

Since colored Petri nets are not the main focus of this thesis, we will not cover any more details of these high-level nets. Please refer to [28], for a more formal definition of colored Petri nets. It is however important to emphasize that the computational power of colored Petri nets is the same as ordinary Petri nets, and they are only a better formalism for modelling systems with isomorphic subsystems.

Languages are a common way of measuring the computational power of mathematical modelling tools. Figure 3.13 [29], depicts a comparison of the class of Petri nets languages with other language categories in the Chomsky hierarchy. Finite automata (FA), push down automata (PDA), and linear bounded automata (LBA) are the mathematical models of regular, context free, and context sensitive languages respectively. As can be seen from the Figure, regular languages as well a subset of context sensitive and context free languages can be expressed using ordinary Petri nets. Petri nets with inhibitors (IPN) have the same

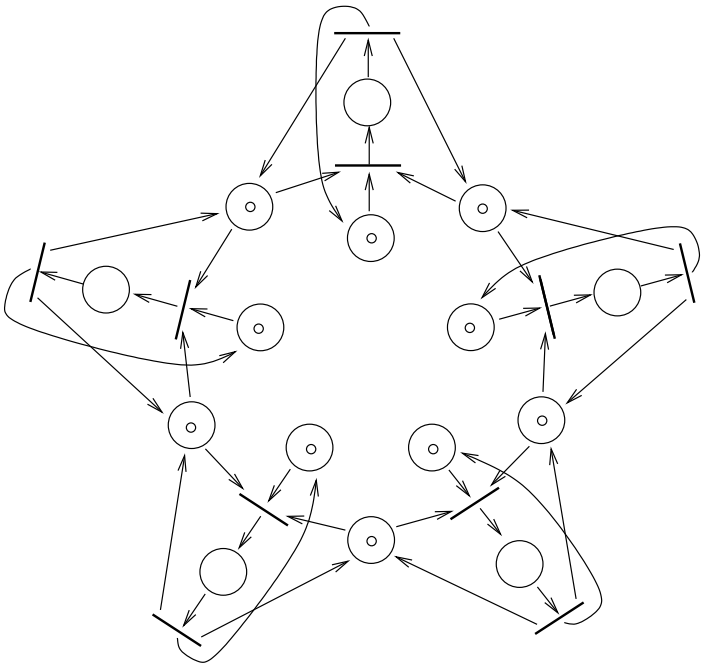


Figure 3.11: A Ring of 5 Dining Philosophers

power as the Turing machines (TM), and colored Petri nets (CPN) have the same power as ordinary Petri nets (PN).

3.2 Model Checking of Ordinary Petri Nets

In this section, we discuss the model-checking of Petri nets against linear temporal logic introduced earlier. There are a few very important results in the literature on this concept. It is known from [34] that the model checking of linear time μ-calculus with $fi(t)$ as the only predicate is decidable. However, as you extend the logic by adding the marking predicates $ge(p, c)$, $en(t)$, the model-checking problem becomes undecidable. Some fragments of the logic $\mathcal{L}$ however have been shown to be decidable.

In [37], a fragment of $\mathcal{L}$ is defined such that F is the only temporal operator allowed,

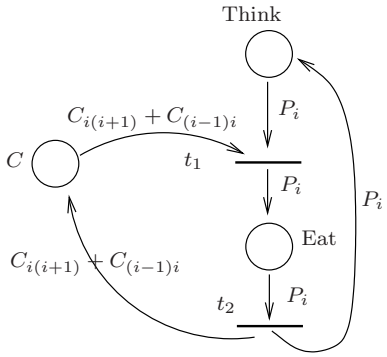


Figure 3.12: Colored Petri Net Model of the Dining Philosophers

and negation is only applied to predicates. It is shown that deciding whether a given Petri net existentially satisfies a formula of this logic is PTIME equivalent to the reachability problem. As an example, one can decide whether $\mathcal{P} \models_\exists F(fi(t) \wedge (F\neg ge(p,3)))$.

In another work [38], the decidability of a similar syntax as [37] with GF (infinitely often) as the only temporal operator allowed is proved. This is done by reducing the problem to an exponential number of instances of the reachability problem. For instance, $\mathcal{P} \models_\exists GF(fi(t) \wedge (GF\neg ge(p,3)))$.

In this section, two fragments $\mathcal{L}^{\mathcal{E}}$ ($\mathcal{L}^{\mathcal{O}}$) of linear temporal logic $\mathcal{L}$ are defined in which every predicate $ge(p,k), en(t)$ is in the scope of an even (resp. odd) number of negations. It is shown that the problem of deciding whether a given Petri net existentially (globally) satisfies a formula of $\mathcal{L}^{\mathcal{E}}$ (resp. $\mathcal{L}^{\mathcal{O}}$) can be reduced to the boundedness problem. The results then are extended to model checking of Petri nets under fairness constraints.

3.2.1 A Decidable Fragment of Linear Temporal Logic

By $\mathcal{L}^{\mathcal{E}}$ (resp. $\mathcal{L}^{\mathcal{O}}$), we denote the fragment of $\mathcal{L}$ formulae whose marking predicates are all in the scope of an even (resp. odd) number of negations. A formula with only transition predicates is considered to be in both $\mathcal{L}^{\mathcal{E}}$ and $\mathcal{L}^{\mathcal{O}}$. For instance, $\neg(F(fi(t) \vee G\neg ge(p,2)))$

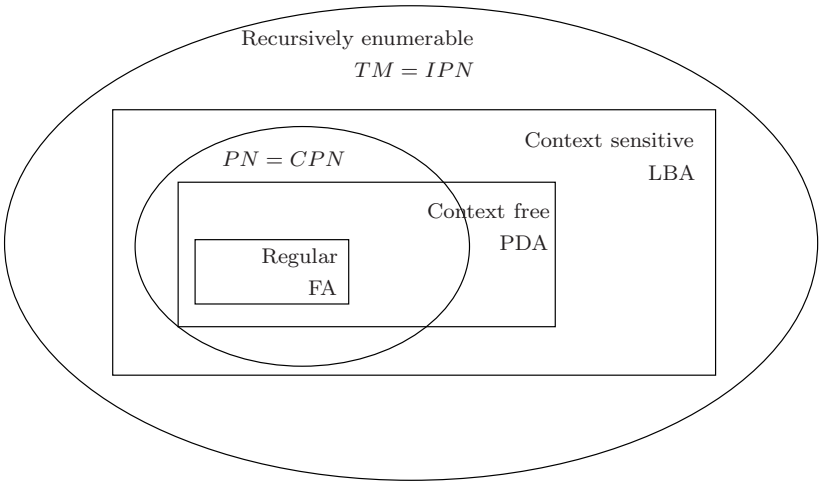


Figure 3.13: Petri nets in Chomsky Hierarchy

belongs to $\mathcal{L}^{\mathcal{E}}$ since $ge(p, 2)$ (its only marking predicate) lies in the scope of 2 negations. Note that negation of a formula of $\mathcal{L}^{\mathcal{O}}$ belongs to $\mathcal{L}^{\mathcal{E}}$, and vice versa. A formula of $\mathcal{L}$ can always be written in the *negation normal form* in which negations are only applied to the predicates. This can be done by pushing negations inward, using the equivalences $\neg X\varphi = X\neg\varphi$, $\neg F\varphi = G\neg\varphi$, $\neg(\varphi_1 U \varphi_2) = \neg\varphi_1 R \neg\varphi_2$ [41]. Therefore, if we put a formula of $\mathcal{L}^{\mathcal{E}}$ in this form, negations can only appear on the transition predicates. As for the above formula, we can write it as $G(\neg fi(t) \wedge Fge(p, 2))$. In the sequel, we will assume that all the formulas are in the negation normal form.

It will be shown that the problem of deciding whether a Petri net $\mathcal{P}$ existentially satisfies a formula of $\mathcal{L}^{\mathcal{E}}$, or globally satisfies a formula of $\mathcal{L}^{\mathcal{O}}$ is decidable by reduction to BP. We take an automata-theoretic approach as in [34]. Given a Petri net $\mathcal{P}$, and a formula φ, a Büchi net $\mathcal{B}_{\varphi}$ (resp. $\mathcal{B}_{\neg\varphi}$) is first constructed which represents all predicate behaviors accepted by φ (resp. $\neg\varphi$). Secondly, a Petri net $\mathcal{E}xt(\mathcal{P})$ is built which represents all predicate sequences of $\mathcal{P}$. Model checking of $\mathcal{P}$ against φ is then reduced to decide whether the language of the Büchi net $\mathcal{E}xt(\mathcal{P}) \times \mathcal{B}_{\varphi}$ (resp. $\mathcal{E}xt(\mathcal{P}) \times \mathcal{B}_{\neg\varphi}$) is empty. The emptiness of

this language can be tested by reduction to BP.

Construction of $\mathcal{E}xt(\mathcal{P})$: Let $\mathcal{P} = (P, T, W, -, -, M_0)$ and φ be a formula of $\mathcal{L}$ to be model-checked against $\mathcal{P}$. We shall explain the construction of $\mathcal{E}xt_\varphi(\mathcal{P}) = (P_e, T_e, W_e, \Sigma_e, \rho_e, M_{0e})$ by adding some new transitions to $\mathcal{P}$ with respect to marking predicates in φ. The set of places P_e remains unchanged, $P_e := P$, and so is the initial marking, $M_{0e} := M_0$. The set of transitions T_e is the union of T, and the set of some new transitions which correspond to marking predicates of φ. The weight function W_e assigns the same value as W to every old pair in $(P \times T) \cup (T \times P)$. The weight assignments to the new pairs will be defined as we add new transitions to T_e. For every marking predicate of the form $ge(p, c)$ in φ, add a new transition t such that $\rho_e(t) := ge(p, c)$, $W_e(p, t) = W_e(t, p) := c$, and $W_e(p', t) = W_e(t, p') := 0$ for every other place $p' \in P_e$. This transition can occur at a marking only when $ge(p, c)$ holds, and its occurrence does not affect the current marking. Similarly, for every predicate of the form $en(t)$, add a new transition t' such that $\rho_e(t') := en(t)$ and $W_e(p, t') = W_e(t', p) := W(p, t)$ for every place $p \in P_e$. Transition t' can occur whenever t is enabled without affecting the current marking. Note that the occurrence of a new transition t only implies the truth of $\rho_e(t)$ at the current marking. We also label every old transition t of T with $fi(t)$. The set of action labels $\Sigma_e := \Gamma$ where Γ is the union of all the marking predicates in φ and a set of transition predicates $\{fi(t) | t \in T\}$. Intuitively, $\mathcal{E}xt_\varphi(\mathcal{P})$ has the same computations as $\mathcal{P}$ with some new transitions interleaved into them.

Construction of $\mathcal{B}_\varphi$: Next, the construction of $\mathcal{B}_\varphi$ with respect to $\mathcal{P}$ and φ is discussed.

Let Γ be a set of predicates as defined earlier. Define a *predicate run* γ on Γ as a function from the set of natural numbers to the power set of Γ; i.e., $\gamma : \mathbb{N} \to 2^\Gamma$. A predicate run can also be thought of as an infinite sequence of truth assignments to the predicates in Γ, denoted by $\Gamma_0\Gamma_1, \cdots$ where at each time instant i the predicates in Γ_i are

evaluated to $True$, and the ones in $\Gamma \setminus \Gamma_i$ to $False$.

According to [40], a Büchi automaton $\mathcal{A}_\varphi = (Q, R, q_0, F)$ can be constructed over the alphabet 2^Γ that accept all the predicate runs that satisfy φ. As usual, Q is a set of states, $R \subseteq Q \times 2^\Gamma \times Q$ is a transition relation, $q_0 \in Q$ is the initial state, and F is the set of accepting states. Every transition $t \in R$ is annotated with a subset of predicates in Γ. We could instead assign a boolean expression e to each transition [41]. Every such expression e represents those truth assignments to predicates in Γ which evaluate it to $True$.

Note that a single boolean expression can represent all the transitions from a given state s to another state r. In fact, all the transitions of the form (s, e_i, r), $1 \preceq i \preceq m$, can be merged into a single transition $(s, \bigvee_{i \in \{1 \cdots m\}} e_i, r)$. Let e be a boolean expression in *disjunctive normal form* (DNF); call e *positive* if all of its predicates are non-negated. For instance, $(ge(p_1, 2) \wedge en(t_1)) \vee fi(t_2)$ is positive. A positive Büchi automaton is a Büchi automaton whose transition labels are positive.

Proposition 11 *Every formula φ of the logic $\mathcal{L}^{\mathcal{E}}$ can be translated into a positive Büchi automaton.*

Proof sketch: First, we use the algorithm of [41], to construct the Büchi automaton $\mathcal{A}_\varphi$ from φ, and then, show how it can be converted into a positive one. The algorithm starts by transforming the temporal operators G, F according to equivalences $F\varphi = TrueU\varphi$ and $G\varphi = FalseR\varphi$; these transformations do not change the number of negations covering a predicate in φ. In the next step, the obtained formula is put in negation normal form by pushing all the negations inward. As explained earlier, this results in a formula whose marking predicates are not negated.

At this point, the algorithm starts from a list of a single node, and recursively adds new nodes to the list where a node is the basic data structure of this algorithm representing a state of $\mathcal{A}_\varphi$. Each node r has a list of properties, denoted by $Old(r)$, assigned to it. Intuitively, this list is a subset of subformulas of φ which hold in the suffixes of computations

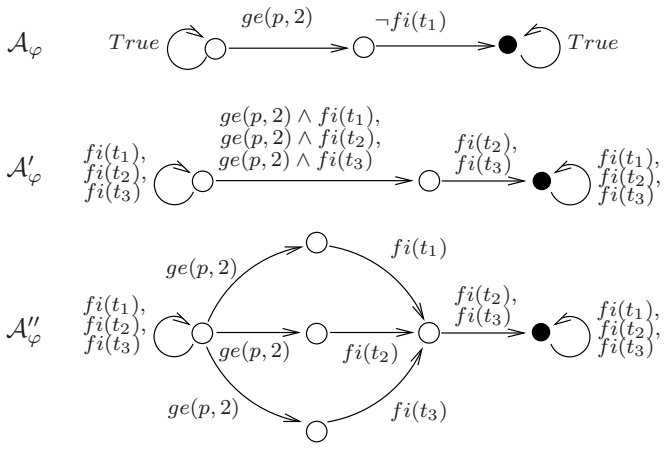


Figure 3.14: Büchi net construction steps

satisfying φ. The node list is complete when no new node is generated. Finally, the transition relation of $\mathcal{A}_\varphi$ is defined as the set of transitions of the form (r, α, r') where r' is a successor node of r, and α is the conjunction of negated and non-negated predicates in $Old(r')$; however, $Old(r')$ may only contain negated transition predicates since the marking predicates are not negated. On the other hand, every negated transition predicate $\neg fi(t)$ can be rewritten as $\bigvee_{t' \in T \& t' \neq t} fi(t')$ – if t is not the next transition to fire, then a different transition t' has to fire. □

Given a positive Büchi automaton $\mathcal{A}_\varphi$, we shall construct a Büchi net $\mathcal{B}_\varphi$ as follows. The construction is done in 3 steps. Figure 3.14 depicts the first two steps for the Büchi automaton of the formula $\varphi = F(ge(p,2) \wedge X \neg fi(t_1))$. It is assumed that the Petri net model $\mathcal{P}$ has three transitions t_1, t_2, t_3.

First, we convert $\mathcal{A}_\varphi$ into another Büchi automaton $\mathcal{A}'_\varphi$ which has the same set of (accepting) states, and whose transition labels are the conjunction of a finite number of predicates. Let (s, e, r) be a transition of $\mathcal{A}_\varphi$ connecting state s to r; Suppose e is a positive boolean expression of the form $d_1 \vee d_2 \cdots \vee d_m$. If any of the disjunctive terms d_i has two transition predicates $fi(t_i)$ and $fi(t_j)$ where t_i, t_j are distinguished transitions,

evaluate d_i to $False$ since two transitions t_i, t_j cannot be both the first transition to fire at a marking of a computation. Furthermore, if d_i does not include a transition predicate as one of its conjunctive terms, then we can rewrite it as $d_i \equiv d_i \wedge True \equiv d_i \wedge \bigvee_{t\in T} fi(t) \equiv \bigvee_{t\in T} d_i \wedge fi(t)$.

By applying the above rewriting rules, e will be transformed into a boolean expression whose disjunctive terms are the conjunction of a finite number of marking predicates and a solitary transition predicate. Let $e' = d'_1 \vee d'_2 \cdots \vee d'_{m'}$ be the obtained expression from e; then, replace (s, e', r) with m' transitions (s, d'_i, r), $1 \preceq i \preceq m'$. For instance, transition $(s, ge(p, 2), r)$ is first transformed into $(s, (ge(p, 2) \wedge fi(t_1)) \vee (ge(p, 2) \wedge fi(t_2)) \vee (ge(p, 2) \wedge fi(t_3)), r)$, and then replaced by three transitions $(s, ge(p, 2) \wedge fi(t_1), r)$, $(s, ge(p, 2) \wedge fi(t_2), r)$, and $(s, ge(p, 2) \wedge fi(t_3), r)$ as depicted in figure 3.14. Note that every transition label of $\mathcal{A}'_\varphi$ is the conjunction of a (possibly empty) sequence of marking predicates followed by a single transition predicate.

In the next step, we convert $\mathcal{A}'_\varphi$ into another Büchi automaton $\mathcal{A}''_\varphi$ whose transitions are labelled only by individual predicates as opposed to conjunction of predicates. This is mainly because transitions of $\mathcal{E}xt(\mathcal{P})$ are labelled with individual predicates, and our goal is to eventually take the product of the final Büchi net with $\mathcal{E}xt(\mathcal{P})$.

For every transition of the form $(s, b_1 \wedge b_2 \cdots \wedge b_m, r)$ where $b_m = fi(t)$ for some $t \in T$, we first add $m - 1$ intermediate states $s_1, s_2, \cdots, s_{m-1}$, and then replace the transition with m transitions (s_i, b_{i+1}, s_{i+1}) where $i \in \{0, 1, \cdots, m - 1\}$, $s_0 = s$, and $s_m = r$. The set of accepting states of $\mathcal{A}''_\varphi$ is the same as $\mathcal{A}'_\varphi$. See figure 3.14 to see an example of this construction.

Finally, a Büchi net $\mathcal{B}_\varphi = ((P_\varphi, T_\varphi, W_\varphi, \Sigma_\varphi, \rho_\varphi, M_{0\varphi}), \mathcal{T}_\varphi)$ is constructed from $\mathcal{A}''_\varphi = (\Gamma, Q, R, q_0, F)$ where:

$$P_\varphi := Q$$

$$T_\varphi := R$$

$$\Sigma_\varphi := \Gamma$$

$$W_\varphi(s,(q,\alpha,q')) := \begin{cases} 1 & \text{if } s = q \\ 0 & \text{otherwise} \end{cases}$$

$$W_\varphi((q,\alpha,q'),s) := \begin{cases} 1 & \text{if } s = q' \\ 0 & \text{otherwise} \end{cases}$$

$$\rho_\varphi((q,\alpha,q')) := \alpha$$

$$M_{0\varphi}(q) := \begin{cases} 1 & \text{if } q = q_0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{T}_\varphi := \{(q,\alpha,q') | q' \in F\}$$

Lemma 5 *Given a Petri net $\mathcal{P}$, and a formula φ, we have*

- *if φ is in $\mathcal{L}^{\mathcal{E}}$, then $\mathcal{P} \models_\exists \varphi$ iff $\mathcal{L}_\omega(\mathcal{E}xt_\varphi(P) \times \mathcal{B}_\varphi) \neq \emptyset$;*
- *if φ is in $\mathcal{L}^{\mathcal{O}}$, then $P \models_\forall \varphi$ iff $\mathcal{L}_\omega(\mathcal{E}xt_\varphi(P) \times \mathcal{B}_{\neg\varphi}) = \emptyset$.*

The main result of this section is stated below.

Theorem 14 *Given a Petri net $\mathcal{P}$, and a formula $\varphi \in \mathcal{L}^{\mathcal{E}}$ ($\mathcal{L}^{\mathcal{O}}$), the problem of deciding whether $\mathcal{P}$ existentially (globally) satisfies φ is decidable.*

Consider first the case when φ is an even formula ($\varphi \in \mathcal{L}^{\mathcal{E}}$). We know from [37], that given a Petri net $\mathcal{P}$ and a finite set of non-empty subsets of transitions $\mathcal{X}$, the problem of deciding whether $\mathcal{P}$ has an infinite computation c such that $inf(c) \cap X \neq \emptyset$ for some $X \in \mathcal{X}$ is PTIME equivalent to BP.

According to lemma 5, $\mathcal{P} \models_\exists \varphi$ iff $\mathcal{L}_\omega(\mathcal{E}xt_\varphi(P) \times \mathcal{B}_\varphi) \neq \emptyset$ iff the Büchi net $\mathcal{E}xt_\varphi(P) \times \mathcal{B}_\varphi$ has an infinite path c such that $inf(c) \cap \mathcal{T} \neq \emptyset$ where $\mathcal{T}$ is the set of final transitions of the Büchi net $\mathcal{E}xt_\varphi(P) \times \mathcal{B}_\varphi$. This reduces the model-checking problem to BP.

For the case when φ is an odd formula ($\varphi \in \mathcal{L}^{\mathcal{O}}$), we have $P \models_{\forall} \varphi$ iff $P \not\models_{\exists} \neg\varphi$ where $\neg\varphi$ is an even formula. $\square$

Remark: Extension of $\mathcal{L}$

The logic $\mathcal{L}$ can be extended by adding new predicates to the logic. For instance, the predicate $ge(p, k)$ can be extended to involve more than one place. Therefore, $ge(p_1, p_2, \cdots, p_r, k)$ holds at a marking of a firing sequence if the total number of tokens in places $p_1, p_2, \cdots, p_r$ is more than, or equal to constant k. More formally for an infinite computation $\pi = M_0, e_1, M_1, \cdots$ and any natural number n:

$$(\pi, n) \models ge(p_1, p_2, \cdots, p_r, k) \text{ iff } \Sigma_{i=1}^{r} M_n(p_i) \geq k$$

One can also extend the definition of ge to transitions. The predicate $ge(t_1, t_2, \cdots, t_r, k)$ holds at a marking M of a firing sequence if the total number of transitions $t_1, t_2, \cdots, t_r$ firings from the initial marking M_0 to M is greater than or equal to k. More formally for an infinite computation $\pi = M_0, e_1, M_1, \cdots$ and any natural number n:

$$(\pi, n) \models ge(t_1, t_2, \cdots, t_r, k) \text{ iff } f(n) \geq k$$

where $f(n)$ is a recursive function defined as:

$$f(0) = 0$$

$$f(n) = \begin{cases} f(n) & : \quad e_n \notin \{t_1, t_2, \cdots, t_r\} \\ f(n) + 1 & : \quad e_n \in \{t_1, t_2, \cdots, t_r\} \end{cases}$$

It can easily be shown that the same result as in theorem 14 holds for the extended $\mathcal{L}$ with $ge(pt_1, pt_2, \cdots, pt_r, k)$ predicates where $pt_1, pt_2, \cdots, pt_r$ are either places or transitions.

In the sequel, we will use the predicate $le(pt_1, pt_2, \cdots, pt_r, k)$ as an abbreviation for $\neg ge(pt_1, pt_2, \cdots, pt_r, k{+}1)$, and $e(pt_1, pt_2, \cdots, pt_r, k)$ as an abbreviation for $ge(pt_1, pt_2, \cdots, pt_r, k)$ $\wedge$ $le(pt_1, pt_2, \cdots, pt_r, k)$.

Remark: Bounded Places

In the remainder of this section, it is explained how to extend the decidable fragment of our defined logic by allowing the marking predicates $ge(p, c)$ of any bounded place p in a formula to be model-checked; i.e., the number of negations covering such a predicate is not important.

We need to make some minor changes in the construction of $\mathcal{E}xt(\mathcal{P})$ by adding a complementary place p_c corresponding to every bounded place p when $ge(p, c)$ is a predicate of the formula to be model-checked. First, we update the weight function such that $W_e(p_c, t) := W_e(t, p)$ and $W_e(t, p_c) := W_e(p, t)$ for every $t \in T_e$. The initial number of tokens in p_c, $M_{0e}(p_c)$, is set to $k - M_{0e}(p)$ where k is a bound on the number of tokens in p. This guarantees that $M(p) + M(p_c) = k$ at any reachable marking M of $\mathcal{E}xt(\mathcal{P})$. Then, add a new transition t with action $\neg ge(p_c, c)$ such that $W_e(p_c, t) = W_e(t, p_c) := k - c + 1$, and $W_e(p', t) = W_e(t, p') := 0$ for every other place $p' \in P_e$. It is not hard to see that the new transition t can occur when the predicate $ge(p, c)$ does not hold. We will see later how this result can be beneficial in model-checking of manufacturing systems models.

3.2.2 Model-Checking Under Fairness

Sometimes, we need to define some constraint on the computations of a Petri net known as a fairness condition. Any computation satisfying the fairness condition is called a fair path. A fair net is a pair $(\mathcal{P}, f)$ where $\mathcal{P}$ is a Petri net, and f is fairness condition expressed as a formula of $\mathcal{L}$. Define $\mathcal{L}_\omega(\mathcal{P}, f)$ as the set of all the action sequences corresponding to the fair paths of $\mathcal{P}$. There are different notions of fairness for Petri nets in the literature. In fact, in [37] 24 versions of fairness are defined. We only state two of them. *Weak fairness* on a Petri net $\mathcal{P}$ requires every transition of $\mathcal{P}$ to fire infinitely often, or be disabled infinitely often; more formally, it is expressed as $\bigwedge_{t \in T} GFfi(t) \vee GF\neg en(t)$ where T is the set of transitions of $\mathcal{P}$. *Strong fairness*, on the other hand, requires every transition to fire infinitely often if it is enabled infinitely often; more formally, it is expressed as $\bigwedge_{t \in T} GFen(t) \Rightarrow GFfi(t)$.

Proposition 12 *For given fair Petri nets* $(\mathcal{P}_1, f_1)$ *and* $(\mathcal{P}_2, f_2)$*, a fairness condition* f *exists such that* $\mathcal{L}_\omega(\mathcal{P}_1 \times \mathcal{P}_2, f) = \mathcal{L}_\omega(\mathcal{P}_1, f_1) \cap \mathcal{L}_\omega(\mathcal{P}_2, f_2)$.

Let $\mathcal{P}_i = (P_i, T_i, W_i, \Sigma, \rho_i, M_{0i})$ for $i \in \{1, 2\}$. First, we transform f_1 and f_2 into f_1' and f_2' respectively by applying the following two modifications:

- every predicate of the form $en(t)$ in f_i, $i \in \{1, 2\}$, is replaced by $\bigwedge_{p \in P_i} ge(p, c)$ where $W_i(p, t) = c$. Obviously if $c = 0$, then $ge(p, c)$ is equivalent to *True*.
- every predicate of the form $fi(t)$ in f_i is replaced by $\bigvee_{s \in \mathcal{H}_t} fi(s)$ where

$$\mathcal{H}_t := \{(t_1, t_2) \in T | t_i = t\}$$

Now define f as $f_1' \wedge f_2'$. Every computation of $\mathcal{P}_1 \times \mathcal{P}_2$ is of the form $\sigma = (M_{01}, M_{02})$ (t_{11}, t_{12}) $(M_{11}, M_{12}) \cdots$ which corresponds to computations $\sigma_1 = M_{01}\ t_{11}\ M_{11} \cdots$ and $\sigma_2 = M_{02} t_{12} M_{12} \cdots$ of $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively. For a given place $p \in P_1$ (P_2), we have $(M_{i1}, M_{i2})(p) = M_{i1}(p)$ $(M_{i2}(p))$. Therefore, a marking predicate $ge(p, c)$ holds at a marking (M_{i1}, M_{i2}) of σ iff it holds at M_{i1} (M_{i2}). As for the transition predicates, if $fi(t)$ holds at M_{i1} (M_{i2}) of σ_1 (σ_2) for some $t \in T_1$ (T_2), then $t = t_{i+1,1}$ $(t = t_{i+1,2})$; therefore, $fi((t_{i+1,1}, t_{i+1,2}))$ and consequently $\bigvee_{s \in \mathcal{H}_t} fi(s)$ holds at (M_{i1}, M_{i2}). On the other hand, if $\bigvee_{s \in \mathcal{H}_t} fi(s)$ holds at (M_{i1}, M_{i2}), then (t, t') (or (t', t)) holds at (M_{i1}, M_{i2}) for some $t' \in T_2$ (T_1); therefore, $fi(t)$ holds at M_{i1} (M_{i2}).

Consequently, σ_1 and σ_2 satisfy fair conditions f_1 and f_2 respectively iff σ satisfies f_1' and f_2' iff σ satisfies $f = f_1' \wedge f_2'$. □

Proposition 13 *Given a Büchi net* $(\mathcal{P}, \mathcal{T})$*, we have* $\mathcal{L}_\omega((\mathcal{P}, \mathcal{T})) = \mathcal{L}_\omega((\mathcal{P}, f))$ *where* $f = GF \bigvee_{t \in \mathcal{T}} fi(t)$.

We say a fair net $(\mathcal{P}, f)$ existentially satisfies a formula φ of $\mathcal{L}$, $(\mathcal{P}, f) \models_\exists \varphi$, iff a fair path c of $\mathcal{P}$ exists such that $c \models \varphi$. Similarly, $(\mathcal{P}, f)$ is said to globally satisfy φ, $(\mathcal{P}, f) \models_\forall \varphi$, iff for every fair path c we have $c \models \varphi$.

Theorem 15 *Given a fair net $(\mathcal{P}, f)$ where f is a weak fairness condition, and a formula $\varphi \in \mathcal{L}^{\mathcal{E}}$ ($\mathcal{L}^{\mathcal{O}}$), the problem of deciding whether $(\mathcal{P}, f)$ existentially (globally) satisfies φ is decidable.*

Let φ be an even formula. Construct Petri net $\mathcal{E}xt(\mathcal{P})$ and Büchi net $\mathcal{B}_\varphi$ on Γ according to $\mathcal{P}$ and φ as explained earlier. The weak fairness condition f can simply be translated into another weak condition f_1 on $\mathcal{E}xt(\mathcal{P})$, resulting in a fair net $(\mathcal{E}xt(\mathcal{P}), f_1)$. Büchi net $\mathcal{B}_\varphi$ can also be translated into a fair net $(\mathcal{P}_2, f_2)$ according to proposition 13. Now the question is to decide whether the product of the two fair nets $(\mathcal{E}xt(\mathcal{P}) \times \mathcal{P}_2, f_1' \wedge f_2')$ has a fair path where $f_1' \wedge f_2'$ has only GF operators, and its negations are applied to predicates. According to [38], this can be reduced to an exponential number of RP instances. The other case of the theorem where φ is an odd formula can be decided similarly. □

When the fairness condition is strong the model-checking problem becomes undecidable even when the temporal property is $True$. For a given Petri net $\mathcal{P}$ if we could decide whether $(\mathcal{P}, f) \models_\exists True$ where f is a strong fairness, then it was decidable whether $\mathcal{P}$ has a strongly fair path which is known to be undecidable from [36].

3.2.3 Factory Example

In this section, we shall investigate the application of our results on a real-world example. Consider a factory which produces some mechanical parts, and pack them. This is done in two stages; first the parts are produced, and then in the second stage they are checked against some quality standards. If a part passes the test, it will be packed, and if it fails, it will be thrown away. Note that production of the parts and their test and packing process cannot be performed simultaneously since running the machinery in parallel consumes more electrical power than allowed by the safety regulations. The Petri net model $\mathcal{P}$ of such factory is depicted in Figure 3.15.

The number of tokens in p_{prt} demonstrates the total number of parts generated in the

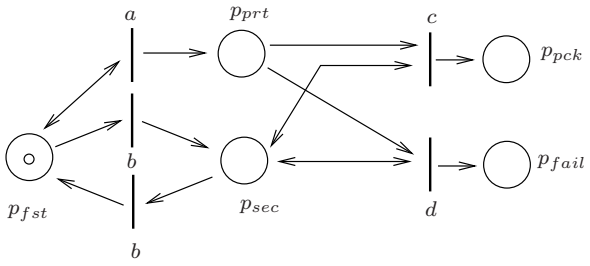


Figure 3.15: Petri net model of a factory

first stage, and ready to be tested and packed. Tokens in p_{pck} demonstrate the number of parts which passed the test, and have been packed. Tokens in p_{fail}, on the other hand, are the ones which failed the test. p_{fst} (resp. p_{sec}) when marked implies that the factory is producing parts (resp. testing and packing). Each transition is labelled by an action from $\Sigma = \{a, b, c, d\}$. The language $\mathcal{L}(\mathcal{P}) := \{\overline{(a^*b(c+d)^*b)^*} : \#(a) \succeq \#(c) + \#(d)\}$. Note that $\mathcal{P}$ is the model of an infinite state system since it is unbounded, and $\mathcal{L}(\mathcal{P})$ is not a regular language.

As a safety property, we may wish to make sure that the factory cannot produce and pack the parts simultaneously, i.e., $\mathcal{P} \models_\forall G\neg(ge(p_{fst}, p_{sec}, 2))$.

We know that the two places p_{fst} and p_{sec} are bounded – this is usually the case for the controller part of any manufacturing system. Therefore, we can decide whether $\mathcal{P}$ globally satisfies the property $G\ \ (eq(p_{fst}, 1) \Leftrightarrow eq(p_{sec}, 0)) \wedge (eq(p_{fst}, 0) \Leftrightarrow eq(p_{sec}, 1))$.

We can also decide $\mathcal{P} \models_\forall G\ eq(p_{sec}, 1) \Rightarrow \neg fst(t_a)$. This means that if the second controller p_{sec} is active, it does not allow for production of any mechanical part – only the test and packing segment of the factory can be running.

Remark. The model of a manufacturing system usually consists of some places representing the sources, and buffers of that system (for instance, inventory of packed parts), and also some places representing its logic which controls the flow through the system. The logic part usually consists of bounded places. Therefore, all properties expressed on

such places can be model-checked.

3.3 Networks of Identical Processes

3.3.1 Computation Model

A process is a tuple (Σ, S, R, F, s_0) where Σ is a set of actions which is the disjoint union of 3 sets: local actions (Σ_l), rendezvous actions (Σ_r), and broadcast actions (Σ_b); S is a finite set of states; R is a set of transitions. A transition is defined as a triple of the form (s_1, ℓ, s_2) where $s_1, s_2 \in S$ and $\ell \in \Sigma_l \cup (\Sigma_r \times \{!, ?\}) \cup (\Sigma_b \times \{!!, ??\})$ – sometimes this transition is written as $s_1 \xrightarrow{\ell} s_2$; $F \subseteq S$ is the set of marker states; finally, s_0 is the initial state. For the sake of notation simplicity, we write $a!$ (resp. $a??$) instead of $(a, !)$ (resp. $(a, ??)$).

We assume that for every broadcast action $a \in \Sigma_b$ and every state $s \in S$, there exists a state $s' \in S$ such that $s \xrightarrow{a??} s'$. We will see later that in a network of processes, every time a process broadcasts a message $a!!$, all the other processes should be able to receive it by performing $a??$; the assumption is to fulfill that goal.

A network $\mathcal{N}$ consists of an arbitrary number of identical processes $\mathcal{T} = (\Sigma, S, R, F, s_0)$. Process $\mathcal{T}$ is also referred to as the *template process* of network $\mathcal{N}$. An instance of the network with a fixed number n copies of $\mathcal{T}$ is denoted by $\mathcal{N}_n$. These processes are numbered 1 to n. The set of global states of $\mathcal{N}_n$ is $G_n = S^n$. A global state $g \in G_n$ is a tuple of n elements $(s_1, s_2, \cdots, s_n)$ where $s_i \in S$, $1 \leq i \leq n$, denotes the state of the i^{th} process. We also use $g(i)$ to denote the i^{th} element of g. The initial state g_{0n} of $\mathcal{N}_n$ is defined so that $g_{0n}(i) = s_0$ for $1 \leq i \leq n$. We define a transition relation $R_n \subseteq G_n \times \Sigma \times G_n$ as follows: 1) local actions of individual subsystems: let g be a global state such that $g(i) = s$ and $s \xrightarrow{\alpha_l} s'$ for some $1 \leq i \leq n$ and $\alpha_l \in \Sigma_l$, then $(g, \alpha_l, g') \in R_n$ where $g'(i) = s'$ and $g'(j) = g(j)$ when $i \neq j$; 2) rendezvous of a pair of processes: let g be a global state such

that $g(i) = s_i$, $g(j) = s_j$, $s_i \stackrel{\alpha_r!}{\rightarrow} s'_i$ and $s_j \stackrel{\alpha_r?}{\rightarrow} s'_j$ for some distinct numbers $1 \leq i, j \leq n$, then $(g, \alpha_r, g') \in R_n$ where $g'(i) = s'_i$, $g'(j) = s'_j$ and $g'(k) = g(k)$ for any $1 \leq k \leq n$ other than i, j; and 3) a broadcast action: let g be a global state such that $g(i) = s_i$, $s_i \stackrel{\alpha_b!!}{\rightarrow} s'_i$, then $(g, \alpha_b, g') \in R_n$ where $g'(i) = s'_i$ and $g(j) \stackrel{\alpha_b??}{\rightarrow} g'(j)$ for $j \neq i$.

We say a global state $t \in G_n$ is reachable from a global state $r \in G_n$, and write $r \rightarrow t$ if there exists a sequence of global states $g_1, g_2, \cdots, g_k$ such that $g_1 = r$, $g_k = t$ and $(g_i, \alpha_i, g_{i+1}) \in R_n$ for $1 \leq i \leq k-1$ and $\alpha_i \in \Sigma$. Every state is assumed to be reachable from itself. The set of all reachable states from a state $r \in G_n$ is denoted by $\mathcal{R}_n(r)$.

Template processes, in their general form, are sometimes called *broadcast templates* since they allow for broadcast actions. A special class of broadcast templates are the ones with empty sets of broadcast actions, $\Sigma_b = \emptyset$. Such templates are known as *rendezvous templates* since they only allow for rendezvous and local actions.

3.3.2 Petri Nets Modelling of Networks with Rendezvous Templates

First, we construct a Petri net $\mathcal{P}_n = (P, T, W, -, -, M_0)$ to simulate an instance $\mathcal{N}_n$ of the network $\mathcal{N}$ with n processes, and then, we will show how our construction can be extended to a more general model $\mathcal{P}_e$ whose computations are the union of the computations of every instance model $\mathcal{P}_n$. Let $\mathcal{T} = (\Sigma, S, R, F, s_0)$ be the rendezvous template of $\mathcal{N}$; $\mathcal{P}_n$ is defined according to $\mathcal{T}$ as follows: let $P = \{p_s : s \in S\}$ – one place corresponding to each state of S. The number of tokens at each place denotes the number of processes in the corresponding state. Therefore, we set the initial marking M_0 such that $M_0(p_{s_0}) = n$, and $M_0(p) = 0$ for every place $p \in P \setminus \{p_{s_0}\}$. Every transition of $\mathcal{N}_n$ is either a local action of one process, or a rendezvous action shared between two processes. Corresponding to every transition $(r_1, \alpha_l, r_2) \in R$, add a transition t to T such that $W(p_{r_1}, t) = W(t, p_{r_2}) = 1$ –

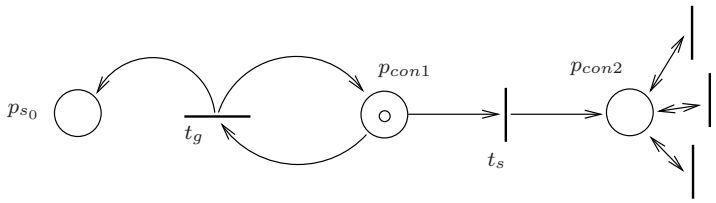


Figure 3.16: Token Generator in Petri Net $\mathcal{P}_e$

firing of t removes a token from p_{r_1}, and adds a token to p_{r_2}. Furthermore, for every two transitions $(r_1, \alpha_r!, r_2), (s_1, \alpha_r?, s_2) \in R$ add a new transition t to T such that $W(p_{r_1}, t) = W(p_{s_1}, t) = W(t, p_{r_2}) = W(t, p_{s_2}) = 1$ – firing of t removes a token from each one of the two places p_{r_1}, p_{s_1}, and adds a token to p_{r_2}, p_{s_2}. This completes the construction if $\mathcal{P}_n$. Note that the only difference between instance models $\mathcal{P}_n$ is in their initial markings. We wish to change $\mathcal{P}_n$ slightly so that the initial number of tokens in p_{s_0} can be set randomly. A new Petri net model $\mathcal{P}_e$ is then constructed by adding two new places p_{con1}, p_{con2}, and two new transitions t_g, t_s to $\mathcal{P}_n$. The new places and transitions, and the way they connect to old places and transitions in $\mathcal{P}_n$ are depicted in the following Figure.

The place p_{con1} is initially marked with a single token. This enables transition t_g, and it can therefore fire to add an arbitrary number of tokens to p_{s_0}. Firing of t_s, on the other hand, removes the token of p_{con1}, and adds a token to p_{con2}. This stops the generation of tokens in p_{s_0}. Also note that p_{con2} being marked is a necessary condition for firing any transition in $T \setminus \{t_s, t_g\}$. In other words, none of the transitions corresponding to $\mathcal{N}$ transitions can fire until p_{con2} is marked. The construction of $\mathcal{P}_e$ is such that it initially generates an arbitrary number n of tokens in p_{s_0}; then t_s is fired, and an instance of the network $\mathcal{N}$ with n processes is simulated. The initial marking of $\mathcal{P}_e$ is defined so that p_{con1} is marked with a single token, and every other place is empty.

Now let $\Psi \in \mathcal{L}$ be a formula expressed on $\mathcal{P}_n$. One may wish to know whether $\mathcal{P}_n$ globally satisfies Ψ for every natural number n. This, however, cannot be done in a finite

time since it has an infinite number of instances. Instead, we could check whether $\mathcal{P}_e$ globally satisfies $(F(fi(t_s) \wedge X\Psi)) \bigvee \neg(Ffi(t_s))$. In other words,

$$\sum_{n\in\mathbb{N}} \mathcal{P}_n \models_\forall \Psi \quad \text{iff} \quad \mathcal{P}_e \models_\forall (F(fi(t_s) \wedge X\Psi)) \bigvee \neg(Ffi(t_s))$$

In fact, a possible computation for $\mathcal{P}_e$ is when t_g is fired infinitely to generate an infinite number of tokens in p_{s_0}. Every other computation of $\mathcal{P}_e$ is a sequence of n firings of t_g followed by a computation of $\mathcal{P}_n$. Therefore, $\sum_{n\in\mathbb{N}} \mathcal{P}_n \models_\forall \Psi$ holds iff every computation of $\mathcal{P}_e$ after firing t_s (if t_s fires at all) reaches a marking which satisfies Ψ.

A similar approach can be taken to model a network consisting of an arbitrary number of identical processes $\mathcal{T}$, and a distinguished process $\mathcal{T}_c$, which is sometimes referred to as the *control process*. In order to model such a network, we need to add extra places to Petri net $\mathcal{P}_e$ corresponding to states of $\mathcal{T}_d$. Note that $\mathcal{T}_d$ may only differ from $\mathcal{T}$ in his initial state. If so, we could slightly change the initial marking of $\mathcal{P}_e$ such that the place corresponding to the initial state of $\mathcal{T}_d$ has a single token.

Consider for instance a network with a template process which has three states: nt (non-trying), t (trying), and c (critical) as depicted in Figure 3.17.

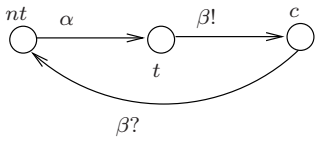


Figure 3.17: Template Process of Network $\mathcal{N}$

Every process in this network can perform a local action α and reach t from nt. In order to reach its critical state c, a process needs to perform $\beta!$, and synchronize with another process performing $\beta?$. The initial state of the network is such that only one process

(control process) is in state c, and the rest of processes are in nt. The Petri net model of such a network is depicted in Figure 3.18. Transition t_1 in this model corresponds to the local action α, and t_2 corresponds to rendezvous action β.

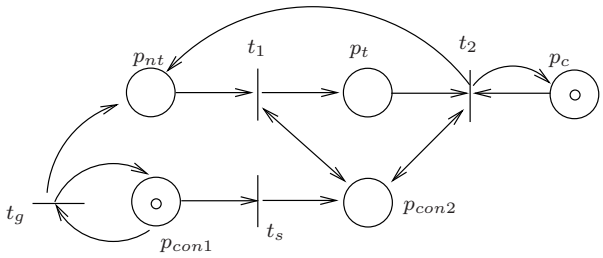


Figure 3.18: Petri Net Model of Network $\mathcal{N}$

As a safety property, we may wish to require the network to satisfy the mutual exclusion property; i.e., it should not be possible for two processes to be in their critical state simultaneously. We can therefore define Ψ as $G\neg ge(p_c, 2)$, and check whether $\mathcal{P}_e \models_\forall (F(fi(t_s) \wedge X\Psi)) \bigvee \neg(Ffi(t_s))$. The formula to be model-checked is an odd formula, and therefore, according to theorem 14, it is decidable whether $\mathcal{P}_e$ globally satisfies it.

We may also wish to model-check a property of a single process $\mathcal{T}^*$ running in the network. To do so, one token needs to be distinguished from the rest. This could be done by assigning new places to the distinguished token. The new model of the network will be as shown in Figure 3.19 where p_{nt}, p_t, p_c represent the states of $\mathcal{T}^*$, and p'_{nt}, p'_t, p'_c represent the states of the rest of processes. There should be a two-headed arc connecting p_{con2} to every transition t_i, $1 \leq i \leq 4$, in the model. These transitions are not depicted in the Figure to avoid confusion.

As a local property of $\mathcal{T}^*$, it is desirable that every time it enters its trying state, it eventually enters its critical state. Let's define Ψ as $G(fi(t_1) \Rightarrow Ffi(t_4))$, and check whether $\mathcal{P}_e \models_\forall (F(fi(t_s) \wedge X\Psi)) \bigvee \neg(Ffi(t_s))$. This property does not hold because there

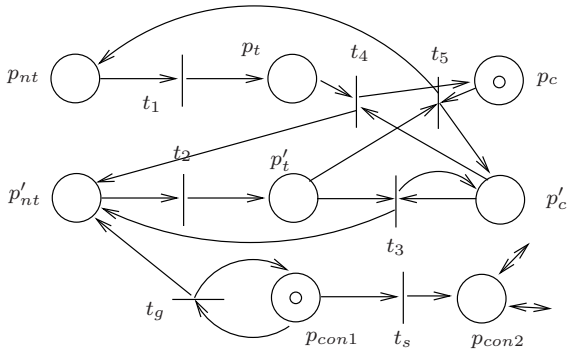


Figure 3.19: Petri Net Model of Network $\mathcal{N}$ with a Distinguished Process

exist non-fair paths in which $\mathcal{T}^*$ is kept in its trying state forever. This problem can be solved by adding a fairness condition to the model. So define $f = \bigwedge_{t \in T} GF(fi(t) \vee \neg en(t))$. The resulted fair net $(\mathcal{P}_e, f)$ satisfies the liveness property of our interest.

3.3.3 Component and Network Blocking

An instance $\mathcal{N}_n$ of a network $\mathcal{N}$ is said to have component blocking if there exists a global state $r \in \mathcal{R}_n(g_{0n})$ and $1 \leq i \leq n$ such that for any $t \in \mathcal{R}_n(r)$, $t(i) \notin F$. This means a global state is reachable (from the initial state) from which subsystem (component) i cannot reach a marker state – component i is blocking. Note that $\mathcal{N}_n$, as we have defined it, is symmetrical, and therefore, if one subsystem in $\mathcal{N}_n$ is blocking, then so is every other subsystem. A network $\mathcal{N}$ has component blocking if an instance of $\mathcal{N}$ does.

An instance $\mathcal{N}_n$ of network $\mathcal{N}$ is said to have network blocking if there exists a global state $r \in \mathcal{R}_n(g_{0n})$ such that for every $t \in \mathcal{R}_n(r)$, $t(i) \notin F$ for some $1 \leq i \leq n$. In other words, reachable state r cannot extend to a state t in which all the subsystems are in their marker states. A network $\mathcal{N}$ has network blocking if an instance of $\mathcal{N}$ does.

Consider the template process $\mathcal{T}$ shown in figure 3.20. This template does not have any broadcast action; therefore, it is a rendezvous template. The initial state of $\mathcal{T}$ is the

state with an arrow pointing into it, and the marker states are distinguished by exiting arrows. The only non-marker state is therefore the rightmost state. Let $\mathcal{N}$ be a network consisting of an arbitrary number of such processes. It can easily be seen that the instances $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ of $\mathcal{N}$ are not blocking, and instances of size larger than 3 (and therefore $\mathcal{N}$) are both component and network blocking because a subsystem can reach its non-marker state, and stay there forever.

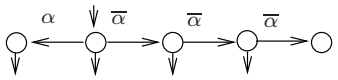


Figure 3.20: Template Process $\mathcal{T}$

In the following theorem, we shall show that problems of deciding the existence of component blocking (CBP) and network blocking (NBP) are undecidable for a network with a broadcast template.

Theorem 16 *Given a network $\mathcal{N}$ with a broadcast template $\mathcal{T}$, it is undecidable whether $\mathcal{N}$ has component or network blocking.*

We shall show that a two-counter machine can be simulated by a network with a broadcast template. The halting problem of counter machines is then reduced to CBP and NBP.

A two-counter machine $\mathcal{C}$ consists of two counters c_1, c_2, and a program with a finite number r of instructions $I_0, I_1, \cdots, I_{r-1}$. An instruction I_i is of one of the following forms.

- $inc(c_i)$: c_i increments by one and the program evolves to I_{i+1}.
- $dec(c_2)$: c_i decrements by one and the program evolves to I_{i+1}.
- $jump(c_i, j, k)$: if $c_i = 0$ the program jumps to I_j; otherwise to I_k.

- *halt*: the program halts.

Given a two-counter machine $\mathcal{C}$ whose counters are initially zero, we construct a broadcast template $\mathcal{T}$ to simulate $\mathcal{C}$. Define the set of states of $\mathcal{T}$ to be $\{I, R, E,\ H, c_1,\ c_2, s_0,\ s_1, \cdots,\ s_{r-1}\}$. Initially, all the processes are in state I. A broadcast action can happen which results in one of the processes to evolve to state s_0 (become a control process), and the rest to state R (become user processes).

Intuitively, the control process simulates the program of $\mathcal{C}$, and the numbers of user processes in state c_1, c_2 represent the value of counters c_1, c_2 respectively. Now we add transitions to $\mathcal{T}$ according to each instruction I_i of $\mathcal{C}$. There are four possibilities.

(a) $inc(c_1)$: add $s_i \xrightarrow{inc(c_1)!} s_{i+1}$, $R \xrightarrow{inc(c_1)?} c_1$; this increments the number of processes in state c_1, and moves the control process to state s_{i+1} to simulate the next instruction. Note that if there is no user process in state R, a deadlock happens. However, the number of user processes is arbitrary, and it can always be chosen large enough so that this instruction is simulated properly. The case of $inc(c_2)$ can be done in a similar way.

(b) $dec(c_i)$: add $s_i \xrightarrow{dec(c_1)!} s_{i+1}$, $c_1 \xrightarrow{dec(c_1)?} R$; this decrements the number of processes in state c_1, and moves the control process to state s_{i+1} to simulate the next instruction. Note that if there is no user process in state c_1, a deadlock happens as $\mathcal{C}$ fails.

(c) $jump(c_1, j, k)$: add $s_i \xrightarrow{zero(c_1)!!} s_j$,$s_i \xrightarrow{nzero(c_1)!} s_k$, $R \xrightarrow{zero(c_1)??} R$, $c_1 \xrightarrow{zero(c_1)??} E$, $c_2 \xrightarrow{zero(c_1)??} c_2$, $c_1 \xrightarrow{nzero(c_1)?} c_1$; therefore, the control process has two choices. It can either (i) broadcast $zero(c_1)!!$ and evolve to s_j. In this case, there should not be any user process in state c_1. All the user processes which are in state R, c_2 remain in the same state, and those which are in state c_1 (if exists any) evolve to an error state E. This determines a cheating on the simulation of $\mathcal{C}$; or (ii) perform $nzero(c_1)!$. This action can only happen when there exists at least one user process in state c_1. In this case, cheating is not possible.

(d) *halt*: add $s_i \xrightarrow{halt} H$. The control state enters the H state which is the only non-marker state.

We also add the following transitions: $E \xrightarrow{test!} E$, $H \xrightarrow{test?} H'$. If the control process

enters the non-marker state H while one user process is in its error state E, then they can synchronize on the action *test* which takes the control process to a marker state H'. So the only case when the control process remains in H is when there is no process in the error state E – the network has not cheated. In other words, the two-counter machine halts iff the broadcast network blocks. On the other hand, halting problem of a two-counter machine is undecidable, and therefore, CBP and NBP are undecidable. Note that in this construction network blocking and component blocking are equivalent. □

In the sequel, we shall show that if we restrict ourselves to rendezvous templates, then CBP and NBP become decidable. Before stating the theorems, we need to define a home space for a Petri net.

A subset L of $\mathbb{N}^n$ is *linear* if there exist vectors $u, v_1, v_2, \cdots, v_r \in \mathbb{N}^n$ such that

$$L = \{u + \sum_{i=1}^{r} n_i v_i : n_i \in \mathbb{N}\}$$

The vectors v_i, $1 \leq i \leq r$ are known as the *period*s of the linear set L, and u as its *base*. The union of a finite number of linear sets is a *semilinear* set [39].

A set of markings $\mathcal{H}$ of a given Petri net $\mathcal{P}$ is called a *home space* iff every reachable marking M of $\mathcal{P}$ can reach some marking M' in $\mathcal{H}$. It is known from [39] that the problem of determining whether a given set of markings $\mathcal{H}$ is a home space for a given Petri net $\mathcal{P}$ is decidable when $\mathcal{H}$ is a linear set, or the union of a finite number of linear sets with the same periods.

Theorem 17 *Given a network $\mathcal{N}$ with a rendezvous template $\mathcal{T}$, it is decidable whether $\mathcal{N}$ has network blocking.*

Proof: Let $\mathcal{T} = (\Sigma, S, R, F, s_0)$ be the template process of a network $\mathcal{N}$. The goal is to decide whether a global state of this network is reachable which cannot evolve to another state with all processes in their marker states. Let $\mathcal{P}_e$ be the Petri net model of $\mathcal{N}$.

Let $\mathcal{H}$ be the set of markings in which all the tokens are in places corresponding to marker states of $\mathcal{T}$. More formally, $\mathcal{H} = \{M \in \mathbb{N}^{|P|} : \forall s \notin F, M(p_s) = 0\}$ where P is the set of places of $\mathcal{P}_e$. This set is clearly linear.

The problem of network blocking of $\mathcal{N}$ is then reduced to the problem of checking whether $\mathcal{H}$ is a home space for $\mathcal{P}$ which is known to be decidable. Therefore, NBP is decidable. □

Theorem 18 *Given a network $\mathcal{N}$ with a rendezvous template $\mathcal{T}$, it is decidable whether $\mathcal{N}$ has component blocking.*

Proof: This time we construct P_e by assigning distinguished places to the states of one process in the network. In other words, every state $s \in S$ is represented by a place p_s in $\mathcal{P}_e$. Let $\mathcal{H}$ be the set of markings which have at least one token in p_q for some $q \in F$. More formally, $\mathcal{H} = \{M \in \mathbb{N}^{|P|} : \exists q \in F, M(p_q) \geq 1\}$ where P is the set of places of $\mathcal{P}_e$. This set is clearly the union of a finite number of linear sets with the same set of periods: $\mathcal{H} = \bigcup_{q \in F} \{u_q + \sum_{i=1}^{|P|} n_i b_i : n_i \in \mathbb{N}\}$ where the b_i's are unity vectors and

$$u_q(p) := \begin{cases} 1 & \text{if } p = p_q \\ 0 & \text{otherwise} \end{cases}$$

Note that there is always only one place p_s, $s \in S$, which holds a token. Therefore, if for a given reachable marking $M(p_q) \geq 1$, $q \in F$, then $M(p_q) = 1$ and $M(p_s) = 0$ for any $s \in S$ other than q. In other words, the distinguished process in the network is in a marker state q.

The problem of component blocking of $\mathcal{N}$ can then be reduced to the problem of checking whether $\mathcal{H}$ is a home space for $\mathcal{P}_e$ which is known to be decidable. □

3.4 Networks of Isomorphic Processes

In networks of identical processes, the subsystems cannot distinguish one another. In networks of isomorphic processes, on the other hand, every subsystem has a distinguished index. The subsystems are numbered 1 to n (in a network with n subsystems), and can pass messages through rendezvous or broadcast actions. It can be determined what subsystem the information is sent to by including that subsystem's index in the passed message. This type of communication is needed to model complex networks such as telephone networks.

The template process of a rendezvous network of isomorphic processes is best defined in a *precondition-effect* style [46]. The states of a process is described in terms of its variables, and the transitions are expressed by 1) their preconditions; more precisely, in what states they are enabled; and 2) their effect on the states. Note that the number of processes in the network is a parameter of the template, and therefore, the number of actions as well as states of a process may be dependent on this parameter. A template process is of the form $P(i, n)$ where i denotes the unique process identity, and n denotes the number of processes in the network. The processes in the network can be generated by appropriate valuation of these parameters. For instance to generate a process whose ID is 4 in a network with 10 processes, we replace i by 4, and n by 10 in the template process.

It will be shown that CBP and NBP are undecidable for such networks by reduction from the halting problem.

The intuition is that for any given Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, h)$, we define a network N which can simulate M, and its blocking is equivalent to the halting of M on the empty tape – a similar approach is taken in [20] to show that blocking is undecidable in ring networks of arbitrary size. As usual, Q is the set of control states of M, $\Sigma \subseteq \Gamma$ is the input alphabet, Γ is the tape alphabet, q_0 is the initial state and h is the halting state. The transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{R, L, S\}$ maps a pair (q, a) to a new state denoted by $suc_s(q, a)$, a new tape letter denoted by $suc_l(q, a)$, and also returns the

direction to which the tape head moves, denoted by $dir(q, a)$.

Processes in N communicate by first setting two other processes as their immediate right- and left-hand neighbors. Every process P_i can initiate a communication by sending out a connection message to another process P_j by performing $con_{ij}!$; P_j then receives this message by performing $con_{ij}?$. This results in P_i's becoming an *initiator* and P_j's becoming its right-hand neighbor – an initiator only has a right-hand neighbor. Process P_j, on the other hand, has P_i as its left-hand neighbor; however, it still needs to establish its right-hand neighbor. It does this similarly by performing $con_{jk}!$. Note that any process which is not an initiator has to have both right- and left-hand neighbors. In this way, a *chain* of processes $c = S_1, S_2, \cdots S_n$ can be formed where S_1 is the initiator of c and S_i is a right-hand neighbor of S_{i-1} and a left-hand neighbor of S_{i+1}, $1 < i < n$. Define the *size* of a chain to be the number of processes in that chain. Therefore, we have $size(c) = n$. Note that the right-hand neighbor of S_n is not established, and that denotes the last process of c. At any time S_n may set up its right-hand neighbor, and extend the size of c to $n + 1$. Also note that all the processes in c are distinguished, i.e., they all have different indices.

Every process P_i has two variables n_l and n_r to store the indices of its left- and right-hand neighbors respectively. These two variables are initially blank since P_i's neighbors are not established yet. Process P_i is also equipped with a tape register π, a boolean flag b, and a state variable q. The state register π is to store a tape symbol of Γ. This register in S_i (the i^{th} process in chain c) corresponds to the i^{th} tape cell of M. The registers of the processes in c, therefore, represent the first n tape cells of M where $size(c) = n$. This register is initially set to Δ, the tape blank symbol. The boolean flag b when set to true indicates where the tape head is pointing. Therefore, only one process in a chain of processes has its flag set to true at any time. This flag is initially set to $False$. The state variable q is used to store a control state of the Turing machine M, and is initially set to q_0, the initial state of M. More formally, the model of a process P_i is defined as follows:

Process P_i

States:

n_l, n_r, indices, initially blank

q, a state in Q, initially q_0

π, a tape letter in Γ, initially Δ

b, boolean, initially $False$

Transitions:

con_{ij}!

Effect

case

$var = free$: $n_l := 0$, $n_r := j$, $b := True$, $var := set$

$var = half\_set$: $n_r := j$, $var := set$

con_{ji}?

Precondition

$var = free$

Effect

$n_l := j$, $var := half\_set$

int!

Precondition

$var = set$, $b = True$, $dir(q, \pi) = S$

Effect

$q := suc_s(q, \pi)$, $\pi := suc_l(q, \pi)$

$out_{i\ n_r}(suc_s(q, \pi))$!

Precondition

$var = set$, $b = True$, $dir(q, \pi) = R$

Effect

$\pi := suc_l(q, \pi)$, $b := False$

$out_{i\ n_l}(suc_s(q, \pi))$!

Precondition

$var = set$, $b = True$, $dir(q, \pi) = L$

Effect

$\pi := suc_l(q, \pi)$, $b := False$

$out_{ji}(q_x)$?

Precondition

$var = set$, $b = False$, $j \in \{n_l, n_r\}$

Effect

$q := q_x$, $b := True$

As mentioned earlier, every process needs to set its neighbors first before any further action. Process P_i has a variable var which denotes if the neighbors of P_i have been set. This variable is initially $free$ meaning that none of P_i's variables are set yet. At this point, P_i may send a connection request con_{ij}! to another process P_j, and become an initiator. This sets the right-hand neighbor of P_i to P_j ($n_r := j$), and its left-hand neighbor to nil ($n_l := 0$) since P_i is an initiator, and does not have any left-hand neighbor [2]. This transition also sets the boolean flag b to $True$; the tape head is pointing to the first cell in its tape. In a second scenario, P_i is called by another process P_j by performing con_{ji}?. This sets P_j as P_i's left-hand neighbor, and sets its var to $half\_set$ since its right-hand neighbor is not set yet. Similarly, it can send a connection request to set its right-hand neighbor, and become set. This procedure can be followed to establish a chain of n processes where n is less than or equal to the number of processes in the network.

Process P_i whose neighbors are established and its flag is set to $True$ simulates the Turing machine on its current control state and tape letter (q, π). It then stores the new tape letter $suc_l(q, \pi)$, and sends out the new control state $suc_s(q, \pi)$ to its proper neighbor, or saves it in its own state variable if $dir(q, \pi) = S$. If $dir(q, \pi) \neq S$, the boolean flag b

[2] We assume that no process in the network has a zero index.

becomes $False$ – the tape head moves to another tape cell. If flag is set to $False$, on the other hand, P_i can receive a message from its left- or right-hand neighbor by performing $out_{ji}(q_x)?$. This transition updates its control state to q_x.

Define those states of P_i whose state variable is the halting state of M, $q = h$, as its non-marker states. A process that has reached a non-marker state will stay there forever since there is no transition out of the halting state of a Turing machine. This therefore causes component and network blocking.

If M halts on the empty tape by going through n of its tape cells, then obviously an instance of network N with at least n processes "can" truthfully simulate M. This results in a process to evolve to a non-marker state. Consequently, the network N will have component and network blocking. On the other hand, when N is blocking, it means that a process of an instance of N can evolve to a non-marker state where $q = h$, and this cannot happen unless M halts on the empty tape. Therefore, if we could decide whether N blocks, then the halting problem would be decidable, which is not the case.

3.4.1 A Generic Template

Consider again networks consisting of an arbitrary number of isomorphic processes. Model-checking of such networks against temporal properties is an instance of *parameterized model checking problem* (PMCP), and in known to be undecidable. We also proved earlier that checking blocking as a more specific temporal property is also undecidable since such networks can easily simulate a Turing machine. Our goal is to propose a general template to make model checking and blocking detection in such networks decidable, and at the same time expressive enough to model real-life processes. The intuition is to always restrict the total number of processes communicating by some fixed number d_{max}. In other words, a set of d_{max} processes which have established connections through an algorithm (which will be explained in more details) can pass messages among each other, but the rest of processes in the network are not allowed to communicate with these processes.

Processes can be thought of as vertices of a dynamic graph G. Each process P_i is distinguished from other processes by its unique index i. The index number i is sometimes referred to as its ID. The corresponding vertex of P_i in G is affixed with i. The connection of two processes is represented by an edge connecting their corresponding vertices in G. Processes in the network need first to connect, and become neighbors; then, they can take their communication further by passing messages. They also can disconnect, and stop their communication. Initially, the vertices of graph G are all disconnected – the set of edges is empty. However, by establishing and removing connections, connected subgraphs may create and disappear. Our proposed algorithm $\mathcal{ALG}$ ensures that the total number of vertices in a connected subgraph is less than or equal to d_{max}.

Every subgraph is initiated by some process P_i when it sends a connection request to another process P_j. This results in P_i's becoming the *master* process of the created 2-node subgraph G_{ij} with two vertices i, j and one edge (i, j). The master process is in fact the first process which initiates a subgraph. Another process, say P_k, may now connect to G_{ij} through P_i and create a 3-node graph $G_{ijk} = (\{i, j, k\}, \{(i, j), (i, k)\})$. Every such connection is first checked with the *master* process of the existing subgraph. The *degree* of a subgraph, $deg(G)$, is defined as the total number of vertices in that subgraph.

The *master* process stores the information of its corresponding subgraph, and therefore, can decide whether a requested connection is allowed. There is always only one master process in every subgraph. The rest of processes are slave. The new connections to the processes outside a subgraph is always done through the master process. In fact, the master process has the most updated information of the whole subgraph, and can decide whether establishing or removing a connection will keep the subgraph degree less than or equal to d_{max}, and then update the graph information accordingly. The master process can pass its status as well as the subgraph information to another slave process, and thereafter, P_j can establish or remove a connection.

Process P_i

States:

nbs, a set of indices, initially empty

G, a subgraph, initially empty

$stat$, with values in $\{neutral, master, slave\}$, initially $neutral$

The variable nbs, also known as P_i's set of neighbors, is a set of indices of processes which are connected to P_i. This set is initially empty since P_i is not connected to any other processes. The variable G is used to store a subgraph information; it is a pair of two sets V and E where V is a set of vertices and E is a set of edges. The next variable $stat$ denotes the status of P_i which is initially set to $neutral$, and can become $slave$, or $master$ as P_i becomes part of a subgraph. Before defining the transitions of P_i, we need to define two functions $merge(G, i, j)$ and $split(G, i, j)$. The first function takes as input a connected subgraph $G = (V, E)$, and two indices i, j, and returns $G' = (V \cup \{i, j\}, E \cup \{(i, j)\})$. In fact, it updates the subgraph G by adding the new edge (i, j) to it. The second function $split(G, i, j)$ returns a connected subgraph which results by removing the edge (i, j) from G. Note that by removing the edge (i, j) from G, it may split into two connected subgraphs. However, $split(G, i, j)$ returns the part which includes i.

Transitions:

$con_{ij}!$

Precondition

$stat = master$, $j \notin V$, $deg(G) < d_{max}$

Effect

add j to nbs, $G := merge(G, i, j)$

$con_{ji}?$

Precondition

$stat = neutral$

Effect

add j to nbs, $stat := slave$

$intcon_{ij}$!

Precondition

$stat = master$, $j \in V \setminus \{i\}$

Effect

add j to nbs, $G := merge(G, i, j)$

$intcon_{ji}$?

Precondition

$stat = slave$

Effect

add j to nbs

$rmv_{ij}(G)$!

Precondition

$stat = master$, $j \in V$

Effect

remove j from nbs, $G := split(G, i, j)$

$rmv_{ji}(G_x)$?

Effect

Case

$deg(G) = 2$ or $G - (i, j)$ is *connected*: remove j from nbs

otherwise: $stat = master$, $G = split(G_x, i, j)$

$switch_{ij}(G)$!

Precondition

$stat = master$, $j \in V \setminus \{i\}$

Effect

$stat = slave$

$switch_{ji}(G_x)$?

Precondition

$stat = slave$

Effect

$stat = master$, $G = G_x$

reset

Precondition

$nbs = \emptyset$

Effect

reset all the variables to their initial values

The *master* process P_i in a subgraph can perform $con_{ij}!$ to connect to another process P_j outside P_i's subgraph. In order to do so, the degree of the obtained subgraph should not exceed d_{max}, and also P_j has to be a *neutral* process. This is clear from the preconditions of $con_{ij}!$ and $con_{ij}?$. As a result, j is added to the list of P_i's neighbors *nbs*, and its subgraph G is updated accordingly. Process P_j, on the other hand, adds i to its list of neighbors, and becomes a *slave*. According to this definition, only *neutral* processes can join an already existing subgraph. We will see later that a *neutral* process cannot carry any information of its previous actions within other subgraphs.

Transition $intcon_{ij}$ is very similar to con_{ij}. However, it is used to connect processes within a subgraph. The *master* process P_i checks whether another process P_j is part of its subgraph; then, they connect, and their list of neighbors as well as P_i's subgraph are updated.

Transition $rmv_{ij}(G)$ is to remove an already established connection between two processes P_i and P_j where P_i is a *master* process, and P_j is a *slave* within the same subgraph. If removal of the edge (i, j) from P_i's subgraph G splits G into two connected subgraphs, then P_i will become the *master* of $split(G, i, j)$ and P_j the *master* of $split(G, j, i)$.

Transition $switch_{ij}(G)$ is to switch status between a *master* and a *slave* process in a given subgraph. The graph information of the new *master* process is then updated

according to what it receives from the old *master* process.

When a process disconnects from all its neighbors, it can reset its variables to their initial values by performing the internal transition *reset*. Then, P_i returns to its *neutral* status, and can become part of another subgraph.

One can think of a process P_i as a composition of a higher level process P_{il} and a lower level process P_{ih}. Variables and transitions, discussed so far, comprise the lower level structure. This part of P_i is responsible for establishing and removing connections with other processes. The distributed algorithm $\mathcal{ALG}$ run by lower level parts of processes in the network guarantees that the degree of created subgraphs do not exceed d_{max}. It also guarantees that a subgraph of degree d_{max} cannot communicate with any process outside its subgraph. The higher level transitions, on the other hand, are utilized for communication among neighbors. In other words, after two processes P_i and P_j become neighbors according to their lower level parts P_{il} and P_{jl}, then their higher level parts P_{ih} and P_{jh} can talk through higher level actions. Any variable or action other than the ones that we have defined so far is considered higher level. A higher level action of process P_i is of the form $ac_{ij}!$ or $ac_{ji}?$ with a precondition $j \in nbs$. Note that transitions of P_{ih} cannot affect the state variables of P_{il}. They can only read the variable *nbs* in their preconditions. Also note that the lower level transition *reset*, resets both lower and higher level variables to their initial values.

Theorem 19 *A Network N as defined above has component (network) blocking iff a network instance of size up to d_{max} does.*

We shall show that a particular process P_i running in a network of size $m = d_{max}$ is blocking iff a process P_j, running in a network of size $n > d_{max}$, is blocking. Let denote by G_i and G_j the subgraphs of P_i and P_j respectively. Note that according to our template definition, a process can only communicate with processes in its own subgraph, and the

state of other processes, outside its subgraph, does not affect its behavior. On the other hand, a new process can only join its subgraph if it's been reset to its initial state; therefore, it cannot have any memory of its previous computations. Furthermore, from P_i's (resp. P_j's) perspective the identities of processes in G_i (resp. G_j) are not important, and only their mutual status determines their enabled transitions.

A subgraph as discussed earlier is developed by adding and removing vertices one at a time. We assign a number to processes in a network based on the order in which they are added to their subgraphs. We refer to this number as the *ordering number* of a process, and denote it by $O(r)$ for a process P_r. Initially, all processes have a 0 ordering number. As stated earlier, a subgraph is created when two *neutral* processes P_i and P_j synchronize on actions $con_{ij}!$ and $con_{ij}?$. As a result, we will have $O(i) = 1$ and $O(j) = 2$. The ordering number of the third process joining their subgraph will be 3, and so forth. More formally, if k is the maximum ordering number of processes in a subgraph, then a new process will be labeled with $k + 1$ upon connection to that subgraph. Note that the ordering number of processes in a subgraph may exceed d_{max} although the total number of processes in that subgraph is limited by d_{max}.

Let f be a function which maps processes in G_i to processes G_j. We call f an isomorphism if it satisfies the following properties:

- f is bijective;
- $f(i) = j$;
- if $O(r) \prec O(r')$, then $O(f(r)) \prec O(f(r'))$ – f preserves the ordering;
- if (r, r') is an edge in G_i, then $(f(r), f(r'))$ is an edge in G_j – f preserves subgraph structure.

Obviously, G_i and G_j should have the same size if such an isomorphism exists; furthermore, every process in G_i will have a unique corresponding process in G_j, and vice

versa. Also note that processes in G_i are from a fixed limited set of processes as opposed to processes in G_j which are from a broader range.

The equivalence relation R relates a state of N_m to a state of N_n if there exists an isomorphism f mapping G_i to G_j, and every process P_r in G_i and its corresponding process $P_{f(r)}$ in G_j are in isomorphic states. In other words, $P_{f(r)}$'s state can be obtained from P_r's state by relabeling of indices according to f.

In fact, when a global state of N_m and a global state of N_n are related according to R, then the two networks can perfectly mimic one another by evolving through equivalent states in which G_i and G_j have the same structure, and corresponding processes in the two subgraphs are in isomorphic states. Therefore, if P_i is in a marker state, then so is P_j, and vice versa. This completes the proof.

□

Chapter 4

Conclusion and Future Work

Due to today's rapidly advancing technology, software and hardware systems tend to become larger and more complicated, and therefore, they become more prone to errors introduced at the design level. Verification of these systems, therefore, is becoming a major step in their design. This step is specially important in the design of safety critical systems.

Model checking is a very strong verification method which has proved itself very useful for finite state systems. In model checking of a finite-state system, the system model (a finite-state automaton) is checked against a temporal property which may be given as a formula, or another automaton. This is done by exhaustive traversal of the system model to see whether the given property holds.

Many important engineering systems, however, are in essence infinite-state, or parameterized. For instance, a system to be model-checked may consist of an infinite size buffer; the number of floors, or the number of elevators may be parameters in the model of an elevator system; the number of subscribers in a telephone network may be varying, or in other words a parameter. Model checking of such systems is undecidable in general. In fact, one cannot model-check an infinite-state system, or a parameterized system by exhaustive traversal of its model since it requires an infinite amount of time.

Model checking of such systems is an active topic of research. In this thesis, we mainly focused on networks consisting of an arbitrary number of processes. We investigated different network topologies including ring networks and fully connected networks. Model checking of such parameterized networks against some given property ϕ is to determine whether every size instance of that network (with a fixed number of processes) satisfies ϕ. Since the number of instances is infinite, we cannot check every instance separately. A huge amount of research has been done to develop restricted frameworks which make the problem decidable. Some of them, including our approach, introduce a bound on the parameters of the defined network to be model checked. This reduces the problem to the model checking of a large finite number of instances. As a specific temporal property, we investigated component and network blocking for the mentioned types of network.

In chapter 2, we discussed ring networks. The respective results of this chapter provide sufficient conditions for the effective model-checking of M^o, LTL\X and CTL*\X properties of the behavior of component subsystems in ring networks. The approach taken is in the spirit of [12, 13] in that it employs induction on the basis of process congruences. But for the case of piecewise recognizable processes, it provides fully automatic model-checking of a subset of M^o and LTL\X properties by showing that all ring networks fall into a finite number of equivalence classes. It also considers networks in which actions of a given process affect only a bounded number of others by defining shuffled processes. If a ring segment is a shuffled process, then all ring networks fall into a finite number of weak bisimilarity classes; consequently, M^o and CTL*\X properties may be effectively checked. A general template is then introduced which guarantees a ring segment to be a shuffled process. This therefore reduces the model-checking problem of ring networks with such templates to checking network instances of up to some fixed known size.

Although finding a bound on the parameters of a system eases the problem a lot, it still leaves us with model checking of a large number of network instances. One way of solving this is by verifying all the instances one by one which is not a very elegant way

considering the fact that these instances may have only minor changes. A better way of approaching this problem has been proposed by Emerson, Trefler and Wahl in [16]. They change the problem to the model checking of an aggregate system which takes only a little more time than model checking of the largest instance. In their method, all the parameters for which the property being checked does not hold will be returned. This method can definitely be applied in our semi-decidable procedure of model checking ring networks. This is something that we will consider in our future work in more details.

In chapter 3, we investigated Petri nets as a modeling tool for infinite-state systems, and showed that the model-checking problem of deciding whether a given Petri net existentially (globally) satisfies a formula of $\mathcal{L}^{\mathcal{E}}$ ($\mathcal{L}^{\mathcal{O}}$) is decidable. The space complexity of our algorithm is exponential in the size of the Petri net, and double exponential in the size of the formula. It is also shown that the problem remains decidable for fair nets with a weak fairness condition, but becomes undecidable with a strong fairness condition.

Model-checking of LTL Properties of Petri nets under weak and strong fairness is discussed in [35] as well; however, the authors restrict themselves to Petri nets with a finite set of reachable markings. In this thesis, however, we are interested in verification of infinite state systems which are modeled with Petri nets.

We have shown how fully connected networks consisting of an arbitrary number of identical processes can be modeled with Petri nets. Such networks have been defined in [45]. It has also been shown how such parameterized networks can be model-checked against correctness specifications in *propositional linear temporal logic*. But the focus of [45] is mainly on verification of local properties of individual processes, although a few specific global properties such as deadlock and mutual exclusion are also discussed. The way we modeled such networks as well as the power of the logic $\mathcal{L}$, makes it more convenient to express global properties. Note that a similar modeling can be used for networks that allow more than 2 processes (a finite number of up to a fixed number c) to synchronize at a time.

In chapter 3, we also discussed fully connected networks of isomorphic processes which communicate by means of rendezvous and broadcast actions. We introduced a general template which always limits the total number of processes connected at a time to some fixed number d_{max}. This therefore reduces the model-checking problem to model-checking instances of size up to d_{max}.

Bibliography

[1] S. Nazari.*Model Reduction in Distributed Supervision.* M.A.Sc. thesis, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, 2004.

[2] O. Gordon. *Supervisory Control Models of Telephone Networks: Symmetry and Scalability.* M.A.Sc. thesis, Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, January 2003.

[3] C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems.* Kluwer Academic Publishers, Boston, 1999.

[4] D. A. Peled. *Software Reliability Methods.* Texts in Computer Science. Springer, 2001.

[5] C. Stirling. *Modal and Temporal properties of processes.* Texts in Computer Science. Springer-Verlag, 2001.

[6] J. Shallit. *Advanced Topics in Formal Langugaes and Automata Theory.* Course Notes of CS462/662, Faculty of Mathematics, University of Waterloo, Waterloo, Canada, 2005.

[7] N. Klarlund and R. Trefler. Regularity Results for FIFO Channels. *Electronic Notes in Theoretical Computer Scince,* 128(6): 21-36, 2005.

[8] R.J. Van Glabbeek. The Linear Time - Branching Time Spectrum. in *Proceedings of CONCUR '90,* LNCS 458, pages 278-297, Springer-Verlag, 1990.

[9] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters,* 22:307-309, 1986.

[10] E. M. Clarke, O. Grümberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. Department of Computer Science CMU-CS-86-155, Carnegie-Mellon University, October 1986.

[11] E. M. Clarke, O. Grümberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 294-303, 1987

[12] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *proceedings of the international workshop on automatic verification methods for finite state systems,* pages 68-80. Springer-Verlag New York,Inc.,1990.

[13] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. *Information and Computation*, 117:1-11, 1995.

[14] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the Association for Computing Machinery,* 39(3):675-735, July 1992.

[15] E. Allen Emerson and Vineet Kahlon. Reducing model checking of the many to the few. In *Conference on Automated Deduction,* pages 236-254, 2000.

[16] E.A. Emerson, R.J. Trefler, T. Wahl. Reducing model checking of the few to the one. to be submitted.

[17] E. A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages,* pages 85-94, 1995.

[18] A. E. Emerson and V. Kahlon. Model Checking Large-scale and Parameterized Resource Allocation Systems. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 251-265, 2002.

[19] E. Clarke, M. Talupur, T. Touili, H. Veith. Verification by network decomposition. In *Fifteenth International Conference on Concurrency Theory,* LNCS 3170, pages 276-291, 2004.

[20] J.G. Thistle and S. Nazari. Model Reduction in Distributed Supervision. In *16th International Symposium on Mathematical Theory of Networks and Systems*, Leuven, Belgium, July 2004.

[21] J.G. Thistle and S. Nazari. Analysis of Arbitrarily Large Networks of Discrete-Event Systems. In *44th IEEE Conference on Decision and Control and European Control Conference*, 2005.

[22] P.K. Hooper. The undecidability of the Turing machine immortality problem. *Journal of Symbolic Logic*, vol. 31, no. 2, pp. 219-234, June 1966.

[23] G.G. Hillebrand, P.C. Kanellakis, H.G. Mairson, and M.Y. Vardi. Undecidable boundedness problems for datalog programs. *Journal of Logic Programming*, vol. 25, no. 2, pp. 163-190, 1995.

[24] T. Murata. Petri Nets: Properties, Analysis and Applications. In *Proceedings of the IEEE,* 77(4):541-580, 1989.

[25] R.M. Karp and R.E. Miller. Parallel Program Schemata. In {Journal of Computer and System Sciences,} 3:147-195, 1969.

[26] A. Bobbio. System Modelling with Petri Nets. In *Systems Reliability Assessment,* pages 103-143, 1990.

[27] D. Garlan. Lecture Notes on Models of Software Systems: Petri Nets, Carnegie Mellon University, 2001.

[28] K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In *A Decade of Concurrency,* LNCS 803, pages 230-272, Springer-Verlag, 1993.

[29] D. Stotts and R. Furuta. Language-Theoretic Classification of Hypermedia Paths. In *Proceedings of the fifteenth ACM conference on Hypertext and hypermedia,* pages 40-41, 2004.

[30] R. Lipton. The Reachability Problem Requires Exponential Space. Thechnical Report 62, Yale University, Department of Computer Science, January 1976.

[31] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets – a Survey. In *Journal of Information Processing and Cybernetics,* 30(3):143-160, 1995.

[32] J. Esparza. Decidability and Complexity of Petri Net Problems – an introduction. In *Lectures on Petri Nets I: Basic Models,* LNCS 1491, pages 374-428, Springer-Verlag, 1998.

[33] J. Esparza. Decidability of Model Checking for Infinite-state Concurrent Systems. In *Acta Informatica,* 34(2):85-107, 1997.

[34] J. Esparza. On the Decidability of Model Checking for Several μ-calculi and Petri Nets. In *Colloquium on Trees in Algebra and Programming,* LNCS 787, pages 115-129, Springer-Verlag, 1994.

[35] T. Latvala and K. Heljanko. Coping With Strong Fairness. Fundamenta Informaticae, 43(1-4), pages 175-193, 2000.

[36] H. Carstensen. Decidability Questions for Fairness in Petri Nets. In *Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science,* LNCS 247, pages 396-407, 1987.

[37] R.R. Howell, L.E. Rosier, H. Yen. A taxonomy of fairness and temporal logic problems for Petri nets. Theoretical Computer Science 82, pages 341-372, 1991.

[38] P. Jančar. Decidability of a Temporal Logic Problem for Petri Nets. Theoretical Computer Science 74, pages 71-93, 1990.

[39] D. Frutos-Escrig and C. Johnen. Decidability of Home Space Property. *Technical Report LRI 503,* 1989.

[40] R. Gerth, D. Peled, M.Y. Vardi, P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification,* pages 3-18. Chapman & Hall, 1995.

[41] E.M. Clarke, O. Grumberg and D.A. Peled. Model Checking. MIT Press, ISBN 0-262-03270-8, 1999.

[42] Krzysztof R. Apt and Dexter C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters,* 22:307-309, 1986.

[43] A. E. Emerson and V. Kahlon. Reducing model checking of the many to the few. In *Conference on Automated Deduction,* pages 236-254, 2000.

[44] A. E. Emerson and K. S. Namjoshi. Automatic Verification of Parameterized Synchronous Systems (Extended Abstract). In *8th International Conference on Computer Aided Verification,* Pages 87-98, 1996.

[45] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the Association for Computing Machinery,* 39(3):675-735, 1992.

[46] N. A. Lynch. Distributed Algorithms.

[47] J. Von zur Gathen and M. Sieveking. A bound on solutions of linear integer equalities and inequalities. *Proceedings of the American Mathematical Society,* 72(1):155-158, 1978.