Compile-Time Schedulability Analysis of Communicating Concurrent Programs

Cong Liu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-94
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-94.html

June 28, 2006

Compile-Time Schedulability Analysis of Communicating Concurrent Programs

by

Cong Liu

B.E. (Tsinghua University, China) 1997
M.E. (Tsinghua University, China) 2000
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Alberto Sangiovanni-Vincentelli, Chair
Professor Robert Brayton
Professor Jasmina Vujic

Fall 2006

The dissertation of Cong Liu is approved:

Chair Date

Date

Date

University of California, Berkeley

Fall 2006

Compile-Time Schedulability Analysis of Communicating Concurrent Programs

Cong Liu

Abstract

Compile-Time Schedulability Analysis of Communicating Concurrent Programs

by

Cong Liu

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

The use of concurrent models has become a necessity in embedded system design. This trend is driven by the growing complexity and inherent multitasking of embedded systems. Describing a system as a set of concurrently executed, relatively simple subtasks is more natural than using a single, complicated task. Embedded systems, however, have limited resources. They often have a few processors. This implies that several software subtasks (programs) have to share a CPU. Compile-time scheduling determines a sequential execution order of the program statements that satisfies certain constraint, e.g. bounded memory usage, at compile time.

We study compile-time schedulability of concurrent programs based on a Petri net model. We consider concurrent programs that asynchronously communicate with each other and the environment through unbounded first-in first-out (FIFO) buffers. The Petri net represents the control flow and communications of the programs, and models data-

dependent branches as non-deterministic free choices. A schedule of a Petri net represents a set of firing sequences that can be infinitely repeated within a bounded state space, regardless of the outcomes of the nondeterministic choices. Schedulability analysis for a given Petri net answers the question whether a valid schedule exists in the reachability space of this net. Due to the heuristics nature of existing scheduling algorithms, discovering powerful necessary condition for schedulability is important to gain efficiency in analysis. We propose a novel structural approach to schedulability analysis of Petri nets. Structural analysis often yields polynomial-time algorithms and is applicable for all initial states. We show that unschedulability can be caused by a structural relation among transitions modelling nondeterministic choices. Two methods for checking the existence of the relation are proposed. One uses linear programming, and the other is based on the generating sets of T-invariants. We also show a necessary condition for schedulability based on the rank of the incidence matrix of the Petri net. These theoretic results shed a light on the sources of unschedulability often found in the Petri net models of embedded multimedia systems. Our experiments show that these techniques are effective and efficient.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

To my wife and parents,

Contents

List of Figures

Acknowledgments

First of all, I would like to thank my advisor, Professor Alberto Sangiovanni-Vincentelli, for his great support, help, and guidance. I owe him a great deal. He introduced me to the exciting research topic, which totally changed my Ph.D. program path. He taught me everything that I need to know to start the research, though he gave me quite some freedom to choose a particular research direction. He provided me a lot of advices, inspiration, and encouragement throughout my Ph.D. study. He is more than an advisor to me, especially when some personal matters arise and need his help.

I also want to thank Dr. Alex Kondratyev and Dr. Yosinori Watanabe for spending their valuable time to hold discussions with me. I enjoyed every discussion, and learned a lot from their criticism. Their feedback and comments put me on the right track during my explorations.

I owe a special thank to Professor Robert Brayton. He served as the chair of my Ph.D. qualification exam committee, and later my Ph.D. dissertation committee. His enthusiasm for research and kindness to young researchers are respectable.

I want to thank Professor Jörg Desel for his comments that shaped one of the major theoretic results of my thesis. I want to thank Professor Jasmina Vujic for serving on my dissertation committee. I want to thank Professor Edward Lee for his valuable comments on my research. I want to thank my colleagues and the staff at the DOP center for their warm-hearted help.

Finally, I want to thank MARCO/DARPA Gigascale Systems Research Center for providing me the financial support.

Chapter 1

Introduction

1.1 Concurrent Models for Embedded Systems

An embedded system is a *special-purpose* computer system that performs dedicated tasks with stringent requirements (e.g. timing, reliability). It is distinguished from general-purpose computer systems such as personal computers. The stringent requirements come from the nature of embedded systems, which usually interact with physical environment and perform safety-critical tasks. Embedded systems are widely used in airplanes, automobiles, office and home appliances, consumer electronics, and medical equipments.

The complexity of embedded systems has increased dramatically. This forces designers to adopt formal models to describe the behavior of a system at a high-level of abstraction and hide implementation details. A formal model with rigorous semantics provide a mathematical means to analyze the characteristics and properties of the modelled system. Formal analysis helps designers to detect design errors at early design stages.

Due to the complexity of embedded systems, it is often necessary to decompose

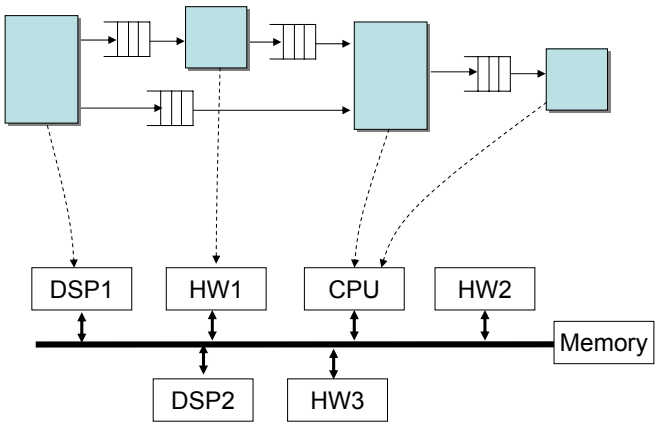


Figure 1.1: Mapping concurrent processes to architectures

a system into a set of functional entities so that the design complexity of each entity is manageable. Modular design facilitates the reuse of intellectual property (IP), which many believe is critical for successful designs.

Representing concurrency explicitly is essential in the models of embedded systems. This is due to the implementation of embedded systems typically consists of hardware, which is intrinsically concurrent, or programmable platforms with multiple computing engines. Exposing the task-level parallelism in a model facilitates experimentations with different hardware/software partitions and mappings onto architectures. It helps designers to find an optimal implementation.

For example, Figure 1.1 illustrates the mapping of a functional speciation described as a set of concurrent processes to a heterogenous multiprocessor architecture. A designer could use this framework to make trade-offs among different mappings.

Note that the entities (concurrent processes) typically need to communicate with each other to exchange data or synchronize. This introduces dependence among operations

in communicating entities, which complicates the design. Thus, it is often desired for a model to decouple the inter-dependence as much as possible so that the design of each entity can be done concurrently and independently.

Concurrent models, such as Kahn process networks [32][33], synchronous dataflow [36], heterochronous dataflow [26], cyclo-static dataflow [25], and communicating sequential processes [31], are suited for these purposes. In these models, a system consists of a set of concurrent autonomous processes. Each is described by a sequential program. The processes communicate with each other and the environment, and run at their own speed. In this chapter, we review the concurrent models that are closely related to our work.

1.1.1 Kahn Process Networks

Many digital signal processing and multimedia embedded systems are naturally represented by a set of cooperating sequential processes incrementally generating or transforming infinite data streams. Kahn and Mac Queen[24, 25] introduced a formal model, called *Kahn process networks*, which is particularly well suited for describing these systems.

A Kahn process network consists of a set of concurrent, autonomous processes that communicate exclusively via unidirectional, unbounded, first-in first-out (FIFO) channels, with blocking reads and non-blocking writes.

In a Kahn process network, a process is described by a sequential program in an infinite loop. Figure 1.2 shows an example. The program typically contains computation and communication instructions, i.e. *read* and *write*. A process often reads data from input channels, performs computations, and writes data to output channels. Reading consumes data. It means that once a data item is read from an input channel, it is no longer present

```
process TestPrime
(OutPort CHA, OutPort CHB)
begin
    loop
        for (int n=2; n<10; n++) {
            if (isPrime(n))
                write (CHA, n, 1)
            else
                write (CHB, n, 1)
        }
    end loop
end proc
```

```
process Reorder
(InPort CHA, InPort CHB)
begin
    int m
    loop
        read (CHA, m, 1)
        print (m)
        read (CHB, m, 1)
        print (m)
    end loop
end proc
```

Figure 1.2: A Kahn process network described by concurrent programs

in the channel. Reading is blocking. It means that if a process tries to read from an empty channel, the execution of the read instruction halts until there are enough data items in the channel to be consumed. Note that it is possible that a single read instruction consumes more than one data item a time. Writing produces data. Similarly, it is possible that more than one data item are produced a time. Writing is non-blocking. It means that a writing operation always succeeds by simply putting data items in the output channel. This is because channels are assumed to have unbounded capacity, thus there is no overflow. This is also because writing is asynchronous. The write operation does not need any form of acknowledgement or synchronization with a read operation to be complete.

In a Kahn process network, a process can not test whether data is present or absent in an input channel, and then decide whether to perform a read operation. It can not wait for data from one *or* another input channel. It means that if a process is blocked on reading, it is waiting for data exclusively from a single channel. No two process send

data to the same channel. If there are more than one process that read from the same input channel, each process gets an identical stream of the data in the channel.

For example, consider the Kahn process network shown in Figure 1.2. If process *TestPrime* first runs an iteration of the *for* loop, which produces integer 2 in channel CHA, and then process *Reorder* read the data and print it out, process *Reorder* will be blocked at the next read statement, since it tries to read from an empty channel, CHB. It could continue execution only after process *TestPrime* runs the third iteration, which produces integer 4 in channel CHB. Note that now channel CHA contains integer 3, which will be consumed in the next run of process *Reorder*.

An important characteristic of a Kahn process network is its *determinacy*. It means that the sequence of data items produced in each channel does not depend on how the processes are executed, either in parallel or sequentially. Thus, given a Kahn process network, there is no ambiguity about its behavior.

In the Kahn process network shown in Figure 1.2, the data streams appearing in channel *CHA* and *CHB* are $(2, 3, 5, 7)^\omega$ and $(4, 6, 8, 9)^\omega$, respectively. The data stream that is finally printed out is $(2, 4, 3, 6, 5, 8, 7, 9)^\omega$.

The determinacy of a Kahn process network makes it an attractive model for system design methodology. It cleanly separates specification from implementation. Given a system specified by a Kahn process network, whether processes are implemented by hardware (executed in parallel) or software (executed sequentially) will not change the input/output behaviors of a system. Since any implementation is functionally equivalent, it makes sense to reason about optimality of an implementation for certain metrics, e.g.

channel size.

The determinacy also implies that a Kahn process network could be viewed as a single bulk process within a larger Kahn process network, because the input/output behaviors remain the same no matter how the internal processes are executed. This means a Kahn process network is compositional. Compositionality is usually a desired property of a model. It allows a design to be performed at different levels of hierarchy. At each level, designers only deal with a network of relatively small size.

Since the execution of a process could be blocked on reading, it is often desired to know whether an execution of a Kahn process network could lead to a *deadlock*, where all processes are suspended. It is known that deadlock is a property of a Kahn process network. That is, if one execution leads to a deadlock, then any execution will lead to a deadlock. Meanwhile, if there exists one indefinite execution of a Kahn process network, then any finite executions can be extended without deadlock.

Though the channels capacities are assumed to be unbounded, it is often desired to know whether there exists a *bounded* execution of the Kahn process network. That is, the number of data items buffered on each channel is always less than a constant finite integer at any point of execution. It is known that boundedness is not a property of a Kahn process network. It means that it is possible that some executions of a Kahn process network require bounded channel capacity while others do not. We refer a sequential bounded execution of a Kahn process network, a bounded-memory schedule of the network, or simply a *schedule*.

For example, the Kahn process network shown in Figure 1.2 require unbounded memory if process *TestPrime* is executed indefinitely without executing process *Reorder*.

However, it is easy to see that if process *TestPrime* is executed only when process *Reorder* is waiting for data, the Kahn process network can be executed with bounded-memory. The bound for each channel is 1.

It has been shown that whether there exists a deadlock or a schedule of a Kahn process network is *undecidable* [43], respectively. In the sequel, we will focus on the schedulability problem. The decidability result implies that there exists no algorithm that can always tell whether there exists a schedule in a finite amount of time. Parks [43] devises a scheduling algorithm that could need an infinite amount of time. He argues that since an indefinite execution of a Kahn process network is wanted, there is no need to answer the existence question in a finite amount of time. Lee uses a different approach [36]. He introduced the synchronous dataflow model, a special kind of Kahn process network, to trade expressiveness for decidability. We will discuss synchronous dataflow model in details in next section. Buck [10] suggests another approach. Instead of restricting the model itself, one could restrict the notion of schedule, e.g. bounded-cycle-length schedule, such that the existence of a schedule can be answered in a finite amount of time. We will discuss Buck's work in details later this chapter.

Kahn process network has been widely used in embedded system design methodologies advocated by industrial and academic researchers. Examples of software packages that support the design methodologies are Ptolemy [1], Metropolis [8], and YAPI [20]. YAPI is developed at Philips research labs as an implementation of Kahn process network based on programming language C++. It introduces communication primitives to represent the interactions between processes. Using a standard C++ compiler, the processes are compiled

to run with a multi-threading package. The Kahn process networks used in the experiments of this thesis are mostly written in YAPI.

In summary, Kahn process network is an attractive model for embedded systems. It has a clear semantics of process interaction, which facilitates well-structured compositional design. However, whether there exists an indefinite execution of a Kahn process network that uses bounded-memory is an undecidable problem. The decidability result indicates that we need to restrict either the expressiveness of the model or the notion of schedule to check schedulability efficiently.

1.1.2 Synchronous Dataflow

Lee and Messerschmitt [36] proposed a programming model, called *synchronous dataflow*, for describing digital signal processing (DSP) systems. The model is a special case of Kahn process network. It assumes that the number of data items (called *tokens*) produced and consumed by a process (called *actor*) are constants and known *a priori*. The advantage of synchronous dataflow over general Kahn process network is that synchronous dataflow is *statically* schedulable. The question whether there exists a bounded-memory execution of a synchronous dataflow model can be answered conclusively at compile time or design time. If the answer is positive, such an execution can be constructed at compile time.

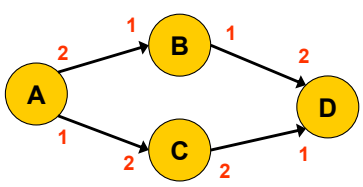
In synchronous dataflow, the execution of a process is an atomic action, called *firing*. An actor can be fired when all input channels have enough tokens to be consumed. If an actor has no input channel, it can be fired at any time. When an actor is fired, it will consume tokens in each input channel, and produce tokens in each output channel,

according to the fixed consumption and production rates, respectively.

The assumption of constant production and consumption rates has dramatic consequences. Firstly, it implies that synchronous dataflow can not model concurrent programs that contain data-dependent communications, e.g. a write operation inside a *if-then-else* construct and the condition depends on the value of data that is read from an input channel. This is a major restriction on the expressiveness of synchronous dataflow. Secondly, it implies that for scheduling purposes we can abstract out data values. Since bounded-memory scheduling concerns about the number of data items accumulated in channels, and in synchronous dataflow the number of data items consumed and produced is irrelevant to data values, we can use tokens that do not carry value to model data items. Consequently, the FIFO ordering of data items in a channel is trivially satisfied, because tokens are undistinguishable. The abstraction greatly simplifies the scheduling problem.

Based on the abstraction, we could represent a synchronous dataflow as a labelled directed graph, where a node represents an actor, and an arc represents a channel. Each input and output of an actor is associated with a constant number, which represents the number of tokens that will be consumed and produced once it is fired, respectively.

The diagraph in Figure 1.3 depicts a synchronous dataflow model, which consists of four actors A, B, C, D, and four arcs. The behavior of actor A, for example, can be interpreted as the following. Actor A has no input channels, thus can be fired at any time. Once it is fired, it will produce two tokens on the arc connected to actor B, and one token on the arc connected to actor C. Note that in order to fire actor D, there must exist at least two tokens on the arc connected from actor B *and* one token on the arc connected



$$\Gamma = \begin{bmatrix} 2 & -1 & 0 & 0 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 2 & -1 \end{bmatrix} \quad x = \begin{bmatrix} 2 \\ 4 \\ 1 \\ 2 \end{bmatrix}$$

Figure 1.3: A synchronous dataflow model, its topology matrix, and repetition vector

from actor C.

Since tokens do not carry values, the state of a synchronous dataflow model is fully determined by the number of tokens on each arc. Thus, we can use a nonnegative integer vector to represent a state, where each entry represents the number of tokens on an arc. The dimension of a state vector is equal to the number of arcs.

The structure of a synchronous dataflow model can be characterized by a *topology matrix* $\mathbf{\Gamma}$, where there is one column for each actor and one row for each arc. The entry $\mathbf{\Gamma}_{ij}$ represents the number of tokens that actor j produced on arc i each time it is fired. If the actor consumes tokens from the arc, the entry is negative. The entry is zero if the arc does not connect the actor. For example, the matrix shown in Figure 1.3 is the topology matrix of the synchronous dataflow shown on the left. Note that the topology matrix of a synchronous dataflow is not necessarily a square matrix.

With the topology matrix of a synchronous dataflow, we can compute the state s reached after a sequence of firings from the initial state s_0.

$$\boldsymbol{s} = \boldsymbol{s}_0 + \mathbf{\Gamma}\boldsymbol{x}$$

where $\boldsymbol{x}$ is the *firing count vector*, whose entries represent the number of times each actor

is fired. The dimension of a firing count vector is equal to the number of actors.

If a finite, non-empty firing sequence returns a synchronous dataflow to the initial state, i.e. $\boldsymbol{s} = \boldsymbol{s}_0$, the firing sequence is called a *finite complete cycle*. The firing count vector $\boldsymbol{x}$ of a finite complete cycle is called a *repetition vector*. By definition, $\boldsymbol{x}$ satisfies the *balance equations*, i.e.

$$\boldsymbol{\Gamma x} = 0$$

For example, the vector shown in Figure 1.3 is the repetition vector of the topology matrix shown on the left. Clearly, any multiple of a repetition vector is a repetition vector.

A schedule of a synchronous dataflow is defined as a finite complete cycle. By repeated executing the schedule, the synchronous dataflow is executed indefinitely without deadlock, requiring only bounded buffer capacity. For example, $(AABBBBCDD)$ represents a schedule of the synchronous dataflow shown in Figure 1.3.

Since the repetition vector is in the null space of the topology matrix, one can prove the following necessary condition for schedulability. If there exists a schedule of a synchronous dataflow model,

$$\text{rank}(\boldsymbol{\Gamma}) = |V| - 1$$

where $|V|$ is the number actors in the synchronous dataflow.

This is easy to see, since the topology matrix Γ has rank $|V| - 1$ implies that the balance equations have a non-trivial solution. The condition is not sufficient because even there exists a repetition vector, the synchronous dataflow could contain deadlocks, which prevent it from being executed indefinitely.

Note that the condition corresponds to a pure structural analysis. It does not

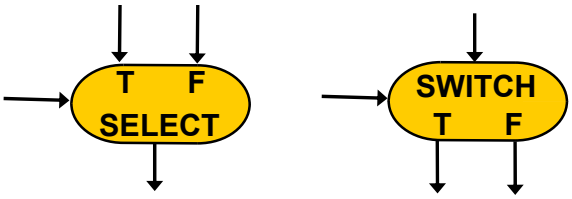


Figure 1.4: Select and Switch actor in Boolean dataflow graph

involve the initial state.

1.1.3 Boolean Dataflow

Buck and Lee proposed a model called *Boolean dataflow* [10], which is a superset of synchronous dataflow. Boolean dataflow model extends synchronous dataflow model by allowing conditional token consumption and production.

This is realized by two special actors called *Switch* and *Select*, which are shown in Figure 1.4. The Switch actor consumes a control token from the *control* input (the left) and then routes a token from the data input (the top) to the appropriate data output, which is determined by the Boolean value of the control token. The Select actor consumes a control token and routes a token from the appropriate input, which is determined by the Boolean value of the control token, to the output. A Boolean dataflow model still satisfies the Kahn conditions, and is just a special case of a Kahn process network.

Switch and Select actors could be used to model conditional constructs (e.g. if-then-else) in sequential programs. Figure 1.5 shows a Boolean dataflow model and its imperative equivalent program. Note that the control inputs of the Switch and Select actors share the same Boolean stream.

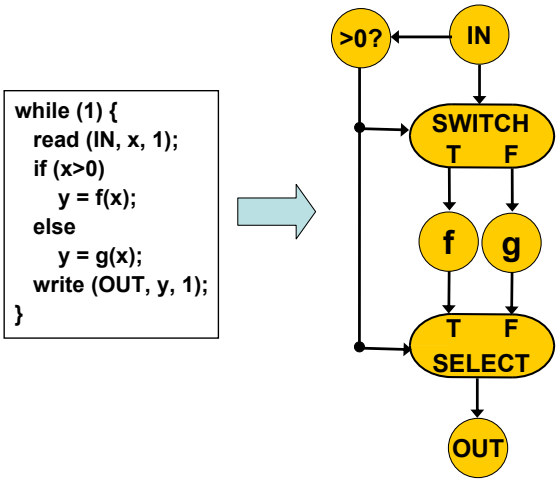


Figure 1.5: Boolean dataflow model of a sequential program

The state of a Boolean dataflow model is determined not only by the number of tokens on regular arcs, but also the values and orders of Boolean tokens on *control* arcs, which are connected to the control inputs of Switch and Select actors.

Lee defined the schedule of a Boolean dataflow as a finite list of guarded firings, where the state after executing the schedule is the same as before, regardless of the outcomes of the Booleans; and firing rules are satisfied at every point in the schedule. For example, $(1, 2, 3, b : 4, !b : 5, 6, 7)$ represents a schedule of the Boolean dataflow shown in Figure 1.5. The actors are referred using the numbers shown in Figure 1.6. The symbol b represents the Boolean value of control tokens.

Buck showed that whether there exists a bounded-memory schedule of a Boolean dataflow graph is *undecidable*. However, we still can adapt the analysis techniques used in synchronous dataflow models to Boolean dataflow models. We can introduce a symbolic variable to represent the consumption and production rates of the conditional inputs and

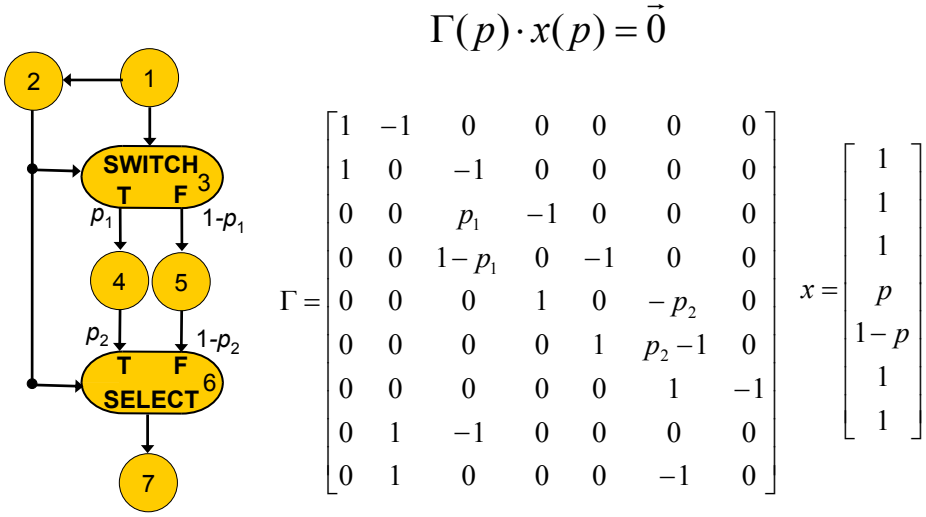


Figure 1.6: A Boolean dataflow model, the topology matrix, the balance equations, and the repetition vector.

outputs. For example, if the production rate of the T data output of a Switch actor is represented by $p, (0 \leq p \leq 1)$, then the production rate of the other data output is represented by $1 - p$. The symbolic variable p could be interpreted as the probability that a Boolean control token takes value *True*, or the proportion of *True* tokens at the control input in one complete cycle. Thus, we could obtain the topology matrix and the balance equations of a Boolean dataflow similar to that of a synchronous dataflow.

In order for the balance equations to have a non-zero solution, certain constraints may have to be imposed on the values of the symbolic variables. For example, Figure 1.6 shows a Boolean dataflow model and its topology matrix. The symbolic variable p_1 is the production rate of T output of the Switch actor, and p_2 is the consumption rate of T input of the Select actor. The balance equation has a non-trivial solution only if $p_1 = p_2 = p$. The repetition vector is also shown in Figure 1.6.

A Boolean dataflow model is called *strongly consistent* if its balance equations have a non-trivial solution no matter what values the symbolic variables may assume. It is called *weakly consistent* if its balance equations have a non-trivial solution only for certain values of the symbolic variables. Thus, the Boolean dataflow model shown in Figure 1.6 is weakly consistent, since it requires the values of two symbolic variables to be equal. However, it is strongly consistent if we only introduce one symbolic variable for the two control inputs. It is easy to see that the two control inputs are provided with the same Boolean stream.

Consistence is related to schedulability. It has been shown that strong consistence is a necessary but not sufficient condition for schedulability.

Lee's definition of schedule of Boolean dataflow is restrictive. If a Boolean dataflow has a schedule as Lee defined, then the schedule can be executed forever with bounded-memory. However, the reverse is not true in general. In other words, there exists a Boolean dataflow that can be executed forever with bounded-memory, but does not have a schedule as Lee defined.

Figure 1.7 shows a Boolean dataflow that is not schedulable according to Lee's definition. For example, a Boolean stream contains an odd number of *False* tokens will result in one token left in the arc connecting the Switch actor and actor 5. However, Figure 1.7 also gives an execution of the Boolean dataflow that can be repeated forever with bounded memory. Particularly, the bound on the arc connecting the Switch actor and actor 5 is two.

Lee's schedule definition, in fact, requires a Boolean dataflow to return to its initial state after a finite number of firings for arbitrary Booleans at the control inputs. Thus, I

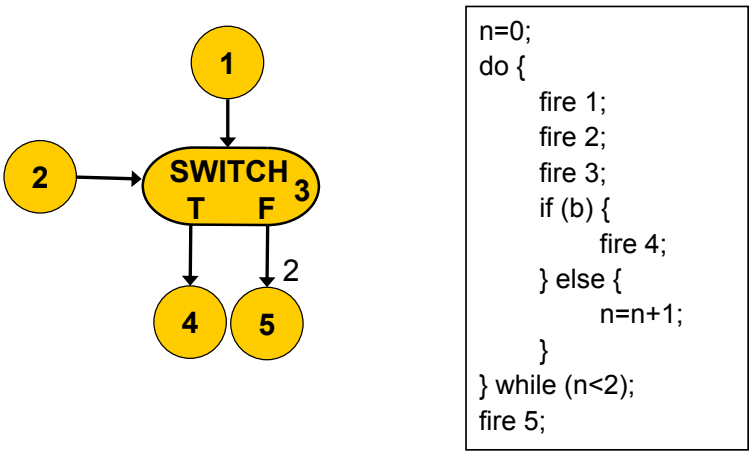


Figure 1.7: A Boolean dataflow and its bounded-memory schedule

called it bounded-cycle-length schedule of a Boolean dataflow. The rational behind the definition is that in many applications, the schedule gives an upper bound of an execution period of a system, given some worst-case execution time of each actor. Lee's schedule definition, however, provides a good example of trading generalness for efficiency.

1.1.4 Petri Nets

In 1962 Carl Adam Petri introduced the basic concepts of a net model, which is later called Petri net. As a graphical and mathematical model, Petri nets have been widely used to specify, model, and analyze distributed, asynchronous, communication systems. Although it has been shown that a Petri net is not equivalent to a Turing machine, its expressiveness is rich enough to capture many essential system characteristics, such as causality, concurrence, conflict, and nondeterminism. We discuss Petri net in details because our compile-time schedulability is based on Petri net models of communicating concurrent

programs.

Formally, a *Petri net* is a 4-tuple (P, T, F, M_0).

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*.
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions*.
- $F\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is the flow relation.
- $M_0\colon P \to \mathbb{N}$ is the *initial marking*, where $\mathbb{N}$ denotes the set of nonnegative integers. In general, $M : P \to \mathbb{N}$ is a *marking*, which represents a state of a Petri net.

We use $N = (P, T, F)$ to denote a Petri net structure without any specific initial marking, and (N, M_0) to denote a Petri net with a given initial marking.

A Petri net can be represented by a directed, weighted, bipartite graph consisting of two kinds of vertices, places and transitions, where edges are either from a place to a transition or from a transition to a place. In a graphical representation, places are drawn as circles, transitions as boxes or bars. There exists an edge from a vertex $u \in P \cup T$ to a vertex $v \in P \cup T$ if $F(u, v) > 0$, and $F(u, v)$ is labelled with the arc. If $F(u, v) = 1$, the label is usually omitted. A marking is depicted by a distribution of black dots in circles. For example, Figure 1.8 shows a Petri net graph with 4 places and 5 transitions. Places p_1, p_4 are initially marked. We could use a nonnegative integer $|P|$-vector to represent a marking, where the $k-$th entry equals $M(p_k)$, the number of tokens contained in place p_k. So the initial marking could be represented as (1001).

A Petri net is *connected* if there exists an undirected path between any pair of vertices in the bipartite graph. In the sequel, we assume a Petri net is connected unless

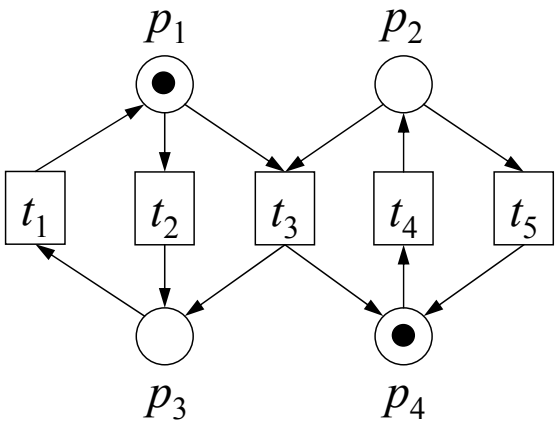


Figure 1.8: A Petri net

stated otherwise.

Let $v \in P \cup T$. Its preset and postset are given by ${}^\bullet v = \{u \in P \cup T | F(u,v) > 0\}$, $v^\bullet = \{u \in P \cup T | F(v,u) > 0\}$. We call ${}^\bullet t$ and $t^\bullet$, the input and output places of transition t, respectively. We call ${}^\bullet p$ and $p^\bullet$, the input and output transitions of place p, respectively. We call a transition without input places a *source* transition. For example, the Petri net shown in Figure1.8 has no source transitions. ${}^\bullet t_3 = \{p_1, p_2\}, t_3^\bullet = \{p_3, p_4\}$.

A transition t is *enabled* at a given marking M, if for each places $p \in P$, $M(p) \geq F(p,t)$. Thus, by definition source transitions are always enabled. When a transition is enabled it can *fire*. The new marking M' reached after the firing of transition t satisfies: for all place $p \in P$, $M'(p) = M(p) - F(p,t) + F(t,p)$. For example, in Figure 1.8, transitions $\{t_2, t_4\}$ are enabled at the initial marking (1001). If transition t_2 is fired, then the new marking is (0011).

A marking M' is *reachable* from the marking M if there exists a sequence of firings that transforms M to M'. It is denoted by $M[\sigma > M'$, where σ represents a *firing sequence*

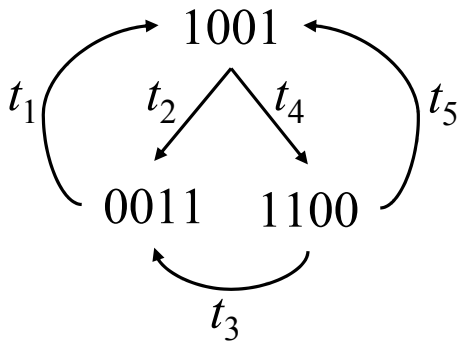


Figure 1.9: The reachability graph of the Petri net shown in Figure 1.8

$(t_{\sigma 1}, t_{\sigma 2}, \cdots, t_{\sigma k})$. A firing sequence is said to be *cyclic* if $M' = M$. The *firing count vector* $\bar{\sigma}$ of a firing sequence σ is a $|T|$-vector, where the i-th entry denotes the number of times transition t_i appears in σ. The set of markings reachable from the initial marking M_0 is denoted as $R(N, M_0)$. A Petri net could have an infinite set of reachable markings. The reachability graph of a Petri net is a labelled digraph (V, E), where vertices are associated with distinct reachable markings, and edges are associated with transitions. There exists edge (M, M') labelled with transition t if and only if $M[t > M'$. For example, Figure 1.9 shows the reachability graph of the Petri net shown in Figure 1.8. Note that the set of reachable markings of the net is finite.

The *incidence matrix* $\boldsymbol{A} = [a_{ij}]$ is a $|T| \times |P|$ matrix, where $a_{ij} = F(t_i, p_j) - F(p_j, t_i)$. For example, Figure 1.10 gives the incidence matrix of the Petri net shown in Figure 1.8. If a marking M' is reachable from M through a firing sequence σ then $M' = M + \boldsymbol{A}^{\mathrm{T}}\bar{\sigma}$. The incidence matrix plays an essential role in the structural analysis of Petri net properties.

A *T-invariant* is a nonnegative integer solution to $\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$. It is known that a

$$A = \begin{bmatrix} 1 & 0 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ -1 & -1 & 1 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 \end{bmatrix} \qquad x_1 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad x_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad x_3 = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 1.10: The incidence matrix and minimal T-invariants of the Petri net shown in Figure 1.8

$|T|$-vector $\boldsymbol{x}$ is a T-invariant if and only if there exists a marking M and a firing sequence σ from M back to M with $\bar{\sigma} = \boldsymbol{x}$. A T-invariant $\boldsymbol{x}$ is *minimal* if there exists no T-invariant $\boldsymbol{x}' \neq 0$ such that $\boldsymbol{x}' \leq \boldsymbol{x}$. It is easy to see that every T-invariant of a Petri net can be expressed as a linear combination of minimal T-invariants with nonnegative rational coefficients. The set of transitions corresponding to non-zero entries in a T-invariant x is called the *support* of an invariant and is denoted by $\|\boldsymbol{x}\|$. A support is said to be *minimal* if no proper nonempty subset of the support is also a support of a T-invariant. Given a minimal support of a T-invariant, there exists a unique minimal T-invariant corresponding to the minimal support, that is called a *minimal-support* T-invariant. It is known that every T-invariant of a Petri net can be expressed as a linear combination of minimal-support T-invariants with nonnegative rational coefficients. For example, Figure 1.10 also gives the minimal T-invariants of the Petri net shown in Figure 1.8. It is easy to check that the T-invariants are also minimal-support T-invariants.

We adopt the interleaving semantics of Petri nets. It means that when a Petri net is executed, at most one transition is fired at any time.

Petri nets can be used to analyze many interesting properties of a system. The

properties could be categorized as *behavioral properties* and *structural properties*. Behavioral properties refer to those properties that are dependent on the initial marking of a Petri net, while structural properties refer to those that are independent on the initial marking. Typically studied behavioral properties include reachability, boundedness, liveness, and reversibility.

- A Petri net (N, M_0) is said to be *bounded* if there exists a nonnegative integer k, such that for each reachable marking $M \in R(N, M_0)$, $M(p) \leq k$ for each place $p \in P$.

- A Petri net (N, M_0) is said to be *live*, if for each transition $t \in T$ and for each reachable marking $M \in R(N, M_0)$, there exists a firing sequence σ from M such that σ contains t.

- A Petri net is said to be *deadlock-free* if for each reachable marking, there exists at least one transition enabled. Note that a Petri net is deadlock-free does not necessarily mean that it is live. It is possible that there always exists a transition enabled at any reachable marking of a Petri net, but some particular transition is never enabled.

- A Petri net is said to be *reversible* if for each reachable marking M, the initial marking can be reached from M.

For example, the Petri net shown in Figure 1.8 is bounded, live, deadlock-free, and reversible, which can be verified by inspecting its reachability graph shown in Figure 1.9.

By definition, a place of a Petri net has infinite capacity. Thus, a Petri net is an infinite-state model in nature. However, it is well known that boundedness, reachability, deadlock-freedom, and reversibility problems are *decidable*. The computation complexity of

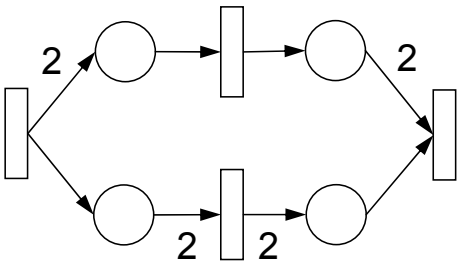


Figure 1.11: A marked graph

the liveness problem remains open.

Although most of the properties of a Petri net are decidable, checking these properties is computationally expensive. Thus, Petri nets in restrictive forms are often studied. Typical subclasses of Petri nets include marked graphs and free-choice Petri nets.

- A *marked graph* is a Petri net where each place has exactly one input transition and one output transition.

- A *free-choice Petri net* is a Petri net where every arc from a place is either a unique outgoing arc or a unique incoming arc to a transition.

Note that the synchronous dataflow model is equivalent to a marked graph. There exists a one-to-one mapping from actors and edges in a synchronous dataflow model to transitions and places in a marked graph, respectively. For example, Figure 1.11 shows a marked graph that is equivalent to the synchronous dataflow shown in Figure 1.3.

1.1.5 Free Choice Petri Net

Free-choice Petri nets have been well studied in Petri net theory. This is mainly because they maintain a good balance between modelling power and analysis complexity. On one hand, they are expressive enough to model many essential system characteristics, such as causality, concurrence, synchronization, and conflicts. On the other hand, their analysis algorithms often have a low computation complexity compared with general Petri nets.

Sgroi et al. [46] studied the scheduling problem of free-choice Petri nets. They aim to increase the modelling power of synchronous dataflow (marked graph) by including structures that model conflicts/choices, and extend the schedule of synchronous dataflow by considering each possible choice.

Their definition of schedule is based on two central notions: finite cyclic firing sequences and the equal conflict relation. The former correspond to the finite complete cycles in synchronous dataflow. The latter is a new notion. Two transitions are in *equal conflict relation* if they have the same nonempty set of input places. Equal conflict transitions model data-dependent choices which are unknown at compile time or design time. A *schedule* is defined as a nonempty finite set $\Sigma = \{\sigma_1, \sigma_2, ...\sigma_k\}$ of finite cyclic firing sequences satisfying the following two properties. First, each finite cyclic firing sequences $(\sigma_i, i = 1, 2, \ldots k)$ contains at least one occurrence of each source transition. Second, if σ_i contains a transition t_i, for each t_j such that t_i and t_j are in a equal conflict relation, there exists σ_j such that the prefix of σ_j up to the first appearance of t_j equals the prefix of σ_i up to the first appearance of t_i. This implies that if a free-choice Petri net contains a non-deterministic choice, then a

valid schedule contains one finite complete cycle for each possible choice.

In the schedule definition of free-choice Petri nets, the requirement of finiteness and cycle is natural, as we need a finite representation of executions that will be repeated infinitely. However, the requirement that each source transition must be contained in each cycle of a schedule impose restrictions on the notion of schedule. It does not only guarantee that the Petri net can be indefinitely executed with bounded-memory, but also ensures bounded response time. That is, in a schedule there is a finite number of firings between two firings of source transitions. If the execution of a transition takes a finite amount of time, then the time elapsed between two firing of source transitions is finite. Note that source transitions model the interactions with the environment. Thus, I call Marco's schedule *bounded-response-time schedule.*

The notion of bounded-response-time schedule is stronger than bounded-memory schedule. We could easily construct a free-choice Petri net that does not have a bounded-response-time schedule, but has a non-terminating bounded-memory execution. Figure 1.12 shows such an example. It is easy to check that there are two minimal cycles, $\{(t_1, t_2, t_4), (t_3, t_5)\}$. However, the second cycle does not contain the source transition t_1. Thus, in order to satisfy the first property, any finite complete cycle that is included in a schedule must contain the first minimal cycle. In order to satisfy Property 2, if a schedule contains (t_1, t_2, t_4), then it also contains $(t_1, t_3, t_5, t_2, t_4)$, then it also contains $(t_1, t_3, t_5, t_3, t_5, t_2, t_4)$, ... Clearly, the schedule would contain an infinite number of cyclic firing sequences. Thus, it is not schedulable according to the definition. But note that the execution $(t_1, (t_3, t_5)^n, t_2, t_4)$, where $n \in \mathbb{N} \cup \{\infty\}$, can be repeated infinitely with bounded-

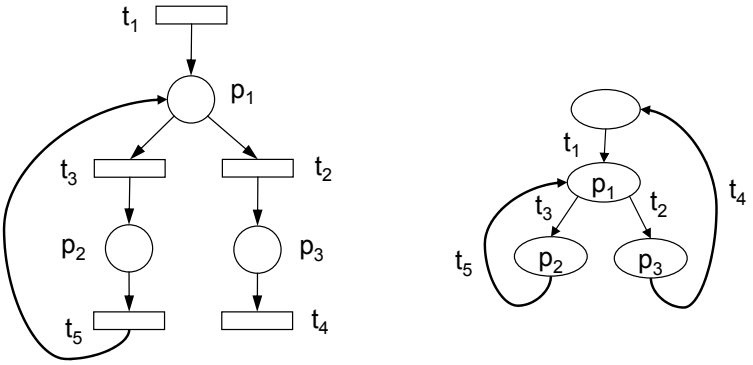


Figure 1.12: A free-choice Petri net and its state transition diagram representing a non-terminating bounded-memory execution.

memory. Specifically, each place contains at most one token during the non-terminating execution. As we will see, the Petri net shown in Figure 1.12 models a conditional loop construct. The firing sequence (t_3, t_5) corresponds to the iterative computation that does not interact with the environment directly.

However, the notion of bounded-response-time schedule is more powerful than bounded-cycle-length schedule. A bounded-response-time schedule may contain a cycle that has infinite length. We illustrate this with the example shown in Figure 1.13. The free-choice Petri net has a bounded-response-time schedule. But if at run-time one choice (transition t_2) is chosen first then the other choice (transition t_3) is chosen forever, the Petri net would never return to the initial state. In other words, if the Petri net returns to the initial state after some firing sequence, then there is no upper bound for the cycle length, because which choice is chosen at run-time is unknown at compile time.

Clearly, a bounded-response-time schedule does not directly give an execution order of a free-choice Petri net. Such an execution is derived from the finite cyclic firing sequences contained in a schedule.

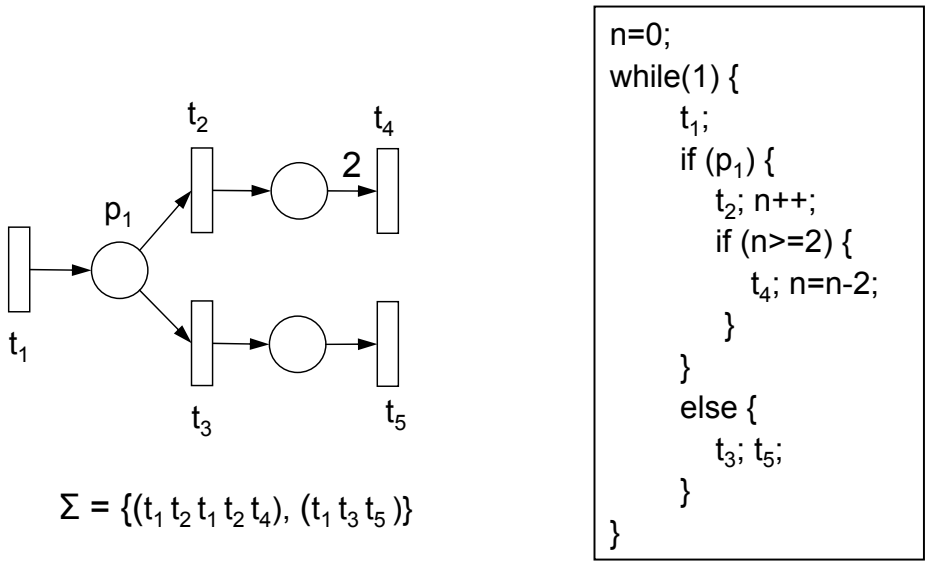


Figure 1.13: A free-choice Petri net, its bounded-response-time schedule and the bounded-memory execution.

The schedulability analysis of free-choice Petri net is based on finding the finite complete cycles for all possible choices. To find complete cycles, a net is decomposed into a set of *conflict-free* components, where each place has at most one output transition, one for each possible choice. It has been shown that a free-choice Petri net is schedulable if and only if each conflict-free component is schedulable. However, the number of conflict-free components of a free-choice Petri net is exponential in the number of conflicting transitions. Thus, the schedulability analysis has an exponential complexity.

It is worth noticing that there are some semantic subtleties in a free-choice Petri net. A free-choice Petri net introduces *nondeterministic choices* and *nondeterministic merge* structures. Figure 1.14 shows two Petri net structures that exhibit nondeterminism. The left structure models a conditional branch. Since tokens in Petri net are distinguishable, which transition to fire is chosen nondeterministically. The right structure models a merge

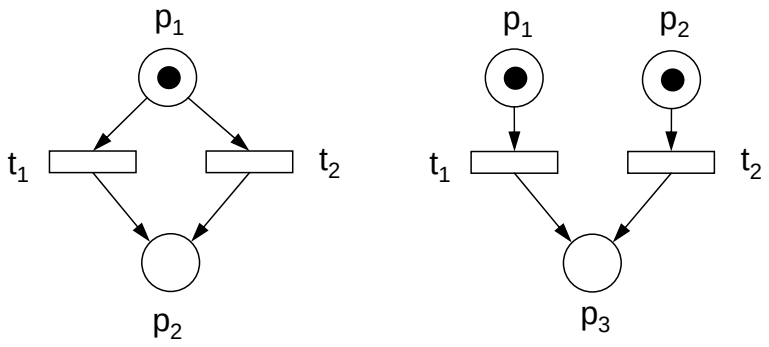


Figure 1.14: Petri net structures exhibiting nondeterminism.

of execution path. This merge is nondeterministic. It means that which transition fires and produces a token in place p_3 is chosen nondeterministically. It is natural to use a nondeterministic choice to model a data-dependent choice, which are unknown at compile-time. However, nondeterministic merge makes the stream of tokens in a merge place indeterminate. Certain restrictions have to be imposed on the scheduler to ensure determinacy. For example, one could enforce a scheduler not reach a state where there exist two enabled input transitions to a merge place. In a Petri net model of Kahn process network, such a state is, in fact, never reachable. This is because the incoming transitions to a merge place belong to the same process. Each channel has a single reader and a single writer. Since each process is a finite state machine, in a process, either a single transition or a free choice set is enabled. Thus, if two incoming transitions to a merge place are enabled, firing one will disable the other.

1.2 Quasi-Static Scheduling

We reviewed some concurrent models of embedded systems. Though exposing concurrency is essential for a model, in practice, each process or task in a concurrent model is not allocated a dedicated resource. This is because embedded systems have limited resources. They often have a few processors, small memories, and limited services provided by an operating system (if any). This implies that several concurrent processes have to share a physical resource (e.g. CPU or bus). Thus, scheduling their operations is inevitable. For embedded software, it means that the concurrent processes (programs) have to be sequentialized.

Depending on how and when the scheduling decision are made, scheduling algorithms could be classified as: static, quasi-static, and dynamic.

- Dynamic scheduling makes all scheduling decisions at run time. Depending on the run-time data, a process may halt or execute. When a process halts and another process executes, the scheduler has to store and restore the state information. It introduces context switch overhead.

- Static scheduling makes all scheduling decisions at compile time. It reduces the context switch overhead, because no run-time scheduling decision has to be made. Due to this static scheduling is restricted to systems without data-dependent choices, e.g. *if-the-else*.

- Quasi-static scheduling is applied to systems in which data-dependent choices occur. It perform static scheduling as much as possible while leaving data-dependent control

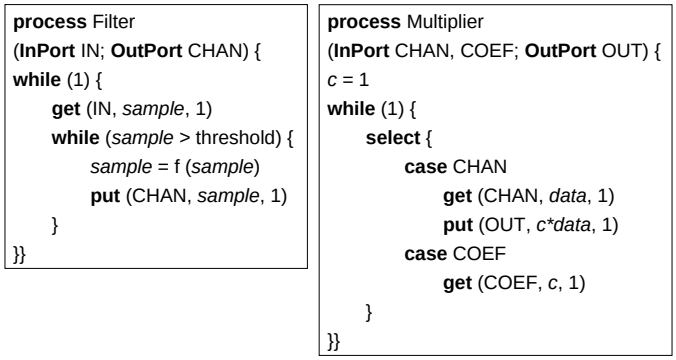
to be resolved at run-time [16], [9].

In the sequel, we use compile-time scheduling to refer both static scheduling and quasi-static scheduling, since all or most of the scheduling work is done at compile time. Clearly, the behaviors of a system is much more predictable when the system is scheduled at compile-time than when it is scheduled at run-time. In fact, as we will show, compile-time schedules can even guarantee certain property of the dynamic behaviors of a system, regardless of the run-time data. This makes compile-time scheduling particularly attractive for applications where predictability is highly desired.

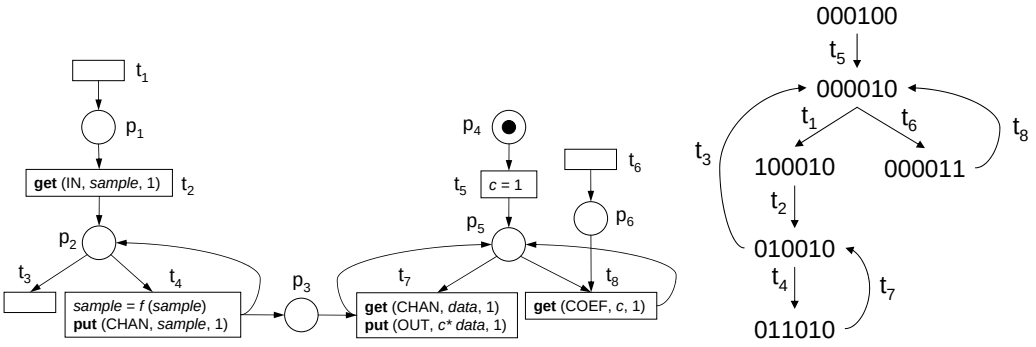
In general, quasi-static scheduling is implemented manually by rewriting the programs, i.e. properly interleaving statements in concurrent processes. Obviously, this is a tedious, time-consuming, and error-prone task. To automate the transformation, a number of synthesis algorithms [36] [46] [16] have been proposed.

In this work, we rely on a modelling framework [16] that uses Petri nets to represent concurrent processes, e.g. Kahn process network. The synthesis starts by transforming the processes that can be specified in a conventional programming language (e.g., C) into a Petri net. Then the Petri net is scheduled. Finally, the schedule is translated into a single sequential process.

Consider the example shown in Figure 1.15. Figure 1.15(a) describes two concurrent processes. Process *Filter* receives samples from the environment and then conditionally keeps sending processed samples to Process *Multiplier*. Depending on the availability of data, Process *Multiplier* either receives a processed sample and outputs the product of the sample and a coefficient, or updates the coefficient supplied by the environment. If both



(a)



(b) (c)

```
process Filter&Multiplier
(InPort IN, COEF; OutPort OUT) {
c = 1
while (1) {
    select {
        case IN
            get (IN, sample, 1)
            while (sample > threshold) {
                sample = f (sample)
                data = sample
                put (OUT, c*data, 1)
            }
        case COEF
            get (COEF, c, 1)
    }
}}
```

(d)

Figure 1.15: (a) Communicating concurrent processes. (b) Its Petri net model. (c) A schedule of the Petri net with initialization. (d) The single sequential process translated from the schedule.

sample and coefficient are present, the choice of which one to consume is done nondeterministically.

Figure 1.15(b) shows the Petri net model, which represents the control flow and communications of the processes. Specifically, a token in place p_2 models the program counter of Process *Filter*, and a token in place p_4 or p_5 models the program counter in Process *Multiplier*. Places p_1, p_3, and p_6 model unbounded buffers *IN, CHAN, and COEF*, respectively. Transitions t_1 and t_6 model environment stimuli, and other transitions model the inlined operations. Note that the data-dependent choice is modelled by a *free choice* (Definition 1).

Figure 1.15(c) depicts the initialization and a *schedule* (Definition 2) of the Petri net. The single process shown in Figure 1.15(d) is generated from the schedule by substituting transition with inlined operations. Note that although originally buffers are assumed to be unbounded, the sequential process can repeatedly execute using buffers of size one.

1.3 Major Results

The synthesis method suggested in [16] is based on heuristics because the existence of a solution for the scheduling problem is proven only for simple subclasses of Petri nets (such as Marked Graphs, see [36]). For general Petri nets the decidability of the scheduling problem remains open. Therefore, discovering powerful sufficient conditions for unschedulability of Petri nets is important for gaining efficiency in analysis.

We observe that regular and repeated structural patterns exist in schedules of Petri nets generated from many applications. These patterns result in infinite paths that

prevent a schedule to return to a designated set of states (initial state in particular), which causes unschedulability. The phenomenon is driven by the definition of a schedule and, interestingly enough, a structural property of the Petri net. Thus, we suggests a structural approach to schedulability analysis of Petri nets. This approach is chosen due to two reasons. First, the underlying algorithms have polynomial-time complexity. Second, the results are applicable to all initial markings.

Our main contribution can be summarized as follows.

- We introduce *cyclic dependence relation*, a structural property defined on transitions of Petri net. To the best of our knowledge, this property provides the most general sufficient condition for Petri net unschedulability.

- Though cyclic dependence is defined through T-invariants, which are nonnegative *integer* solutions to a homogenous linear system, we show that it is not necessary to solve an integer programming problem to check the property. We propose an exact algorithm that is based on linear-programming.

- We prove another sufficient condition for unschedulability based on the *rank* of the incidence matrix. Checking this condition is computationally efficient.

- We demonstrate the effectiveness and efficiency of our approach by applying it to check unschedulability of Petri nets generated from concurrent models of industrial JPEG and MPEG codecs.

Chapter 2

Schedule of Petri Net

2.1 Related Work

The definition of bounded-memory schedule informally describes a property of an indefinite execution of a Kahn process network. However, it is difficult to directly apply the definition to scheduling and schedulability analysis. Parks' scheduling algorithm [43] is based on the basic definition. It starts with predetermined bounds of the buffers. Then it executes the concurrent processes with blocking reads and blocking writes. It increases the buffer capacity when there is a deadlock due to a blocking write. It continues execution and repeatedly checks for deadlocks. Though the algorithm does not rely on a restricted notion of bounded-memory schedule, it could take an infinite amount of time. Thus, it is not practical, particularly for compile-time schedulability analysis.

The schedule of a synchronous dataflow is defined as a finite complete cycle. It can be shown to be exact to bounded-memory schedule. A finite complete cycle is much more operational than the basic definition of a bounded-memory schedule. Thus, the scheduliability

analysis of synchronous dataflow is focused on finding a finite complete cycle. Though the schedule definition of a synchronous dataflow can not be directly applied to the schedule of a general Kahn process network, it gives two important suggestions. First, an operational definition of schedule is needed to gain efficiency in schedulability analysis. Second, a schedule is a finite representation of cyclic behaviors.

The bounded-cycle-length schedule of a Boolean dataflow and the bounded-response-time schedule of a free-choice Petri net are both natural extensions of the schedule definition of a synchronous dataflow. Their definitions are based on the notion of complete cycles. However, as we have shown, the two definitions are restrictive and do not cover many interesting applications. In other words, in practice many bounded-memory schedules do not have bounded cycle length or bounded response time.

In this chapter, we formally define schedule based on a general Petri net. We believe that Petri net is a good model of communicating concurrent program. It is not Turing-complete. Thus, it gives hope for a low computation complexity schedulability analysis. Meanwhile, it is expressive enough to handle many interesting system characteristics. The rich Petri net theory provides a solid foundation where we can build our theory of schedulability analysis. We also believe that abstracting away data value is essential to obtain efficient analysis methods. The rational behind is that bounded-memory schedule, by definition, only concerns about the maximum number of data items ever accumulated in a channel. The values and order of data items are not directly related. As we have shown in a Boolean dataflow, the system state is much more complicated when data values are considered. This prevents an efficient way to characterize the evolution of system state. For

example, in a Petri net, if a marking M' is reached from the initial marking M_0 through a firing sequence σ then $M' = M_0 + \boldsymbol{A}^{\mathrm{T}}\bar{\sigma}$. Thus, any cyclic firing sequence σ_c must satisfy $\boldsymbol{A}^{\mathrm{T}}\bar{\sigma_c} = 0$. While a Boolean dataflow lacks such characterizations. As we will show later, the efficiency of the schedulability analysis of Petri nets comes from the algebraic characterization.

We define the schedule of a Petri net as a labelled, strongly-connected diagraph. As we will see, the definition is still stronger than bounded-memory schedule. It means that a system may have a bounded-memory schedule but no schedule as we defined. But fortunately, in the applications we have seen so far, if a system does not have a schedule as we defined, it does not have a bounded-memory schedule. Our schedule definition is more powerful and general than bounded-cycle-length schedule and bounded-response-time schedule. Our schedules could contain a cycle with unbounded length or infinite firing sequences between the firings of two source transitions.

2.2 Free Choice Set

First, we introduce free choice sets, a key concept in the definition of a schedule.

Definition 1 (Free choice) *Two distinct transitions t and t' are in free choice relation $\mathbb{F}$, if for each place $p \in {}^\bullet t \cup {}^\bullet t'$ and for each transition $t'' \in p^\bullet$, $F(p,t) = F(p,t') = F(p,t'')$.*

The free choice relation is a binary relation from the set of transitions T of a Petri net to T. By definition it is *symmetric*. That is, $t\mathbb{F}t'$ implies $t'\mathbb{F}t$. We show it is also *transitive*.

Lemma 1 *If $a\mathbb{F}b$ and $b\mathbb{F}c$, then $a\mathbb{F}c$.*

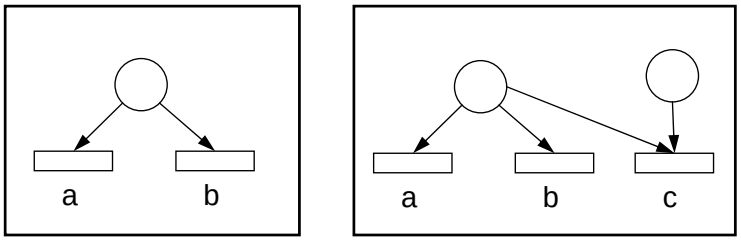


Figure 2.1: Free choice set differs from equal conflict set.

Proof: Choose a place $p_b \in {}^\bullet b$. aFb implies that for each place $p \in {}^\bullet a$ and for each transition $t \in p^\bullet$, $F(p,t) = F(p,a) = F(p_b,b)$. Similarly, bFc implies that for each place $p \in {}^\bullet c$ and for each transition $t' \in p^\bullet$, $F(p,t') = F(p,c) = F(p_b,b)$. Thus, for each place $p \in {}^\bullet a \cup {}^\bullet c$ and for each transition $t \in p^\bullet$, $F(p,t) = F(p,a) = F(p_b,b) = F(p,c)$. □

We call the maximal set of transitions that are pairwise in free choice relation a *free choice set* (FCS). It is easy to see that the set of source transitions trivially satisfies the condition since they have no input places, thus are in a FCS. The notion of FCS differs from that of equal conflict set (ECS). Figure 2.1 shows an example. Transition a, b in the left Petri net are in a FCS and a ECS. Transition a, b in the right Petri net are in a ECS, but not in a FCS.

We introduce the notion of FCS mainly to model environmental stimuli and run-time data-dependent choices, which are unknown at design time. For example, the Petri net shown in Figure 1.15(b) has exactly two binary FCSs $\{t_1, t_6\}$ (which are source transitions) and $\{t_3, t_4\}$. For the sake of simplicity, when referring to FCSs we only refer to those containing exactly two transitions. The assumption can be satisfied by representing an arbitrary FCS with n transitions as a n-leaf tree of binary FCSs. The transformation is

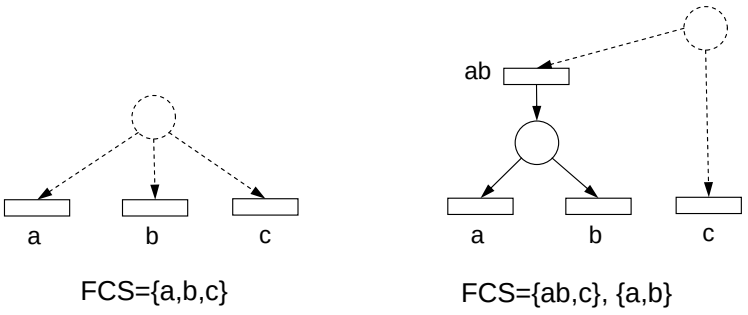


Figure 2.2: Transformation of general FCSs to binary FCSs.

illustrated by an example shown in Figure 2.2. The FCS contains three transitions a, b, c. It is transformed into two binary FCSs $\{a, b\}, \{ab, c\}$. Note that this changes the true concurrent semantics for source transitions to interleaving. But it is not harmful because we anyway use interleaving semantics in constructing a schedule.

Modelling a data-dependent choice as a nondeterministic free choice is an abstraction. It enables us to study the run-time behaviors of a system at compile-time, without knowing which branch is actually taken during execution. As we will show, a schedule of a Petri net considers that all outcomes are possible when a choice has to be made.

2.3 Definition of Schedule

Definition 2 (Schedule) *A* schedule *of a Petri net (N, M) with a set T_s of source transitions is a digraph (V, E, r) with the following properties:*

1. *V and E are finite and nonempty.*

2. *There exists $\mu\colon V \to R(N, M)$ with $\mu(r) = M$. For each $(u, v) \in E$, there exists $t \in T$*

such that $\mu(u)[t > \mu(v)$. *This is denoted by* $u \xrightarrow{t} v$.

3. *Given* $u \xrightarrow{t} v$, *there exists* $u \xrightarrow{t'} v'$ *if and only if* t, t' *are in a FCS.*

4. *For each* $v \in V$, *there exists a directed path to an* await *vertex* v_a *satisfying* $\forall t \in T_s, v_a \xrightarrow{t}$.

5. *For each* $v \in V$, *there exists a directed path from* r *to* v *and from* v *to* r.

The requirement in Property 1 ensures that any execution of a schedule visits a finite subset of a state space: the underlying system can be executed with bounded memory.

Property 2 states that a vertex in a schedule (called schedule state or simply state) corresponds to a reachable marking of the Petri net, an edge corresponds to a transition, and a path corresponds to a firing sequence. Note that it is allowed that several schedule states are mapped to the same marking, and these states fire different transitions at their output edges.

Property 3 has two implications. First, if an outgoing edge of a vertex corresponds to a transition in a FCS, then there must exist another outgoing edge of the vertex that corresponds to another transition in the FCS. It ensures that a schedule is complete, because all possible outcomes of FCSs are considered. In this case, the FCS is said to be *involved* in the schedule. Second, if a state of a schedule has more than one outgoing edge then these edges must correspond to transitions in a FCS. Other enabled transitions are not considered at the state. This property ensures that the schedule can be sequentially executed.

Property 4 guarantees the progress of a schedule, i.e. from any state of a schedule one can reach an await state in which source transitions fire and therefore inputs from the

environment are served. This is a necessary condition for reactive systems. The progress notion could be formulated with respect to any set of transitions that are in FCS (not necessarily source ones). This makes it possible to treat Petri nets without source transitions within the same framework. Note that the theoretic results presented in this thesis are independent of the notion of progress.

Property 5 implies that a schedule is *strongly connected*. Thus, following a schedule, the underlying system can be infinitely executed without deadlock. Note that Property 5 of returning to the root makes the definition of schedule more stringent than the one suggested in [16]. We believe that this is acceptable because most practical applications have a designated reset state that is reachable from any other state (emergency exit) and from which the operation restarts. Such reset state could play a role of the root state in the suggested definition of a schedule.

2.4 Schedule Unfolding

In practice, it is often convenient to study the unfolding of a schedule instead of the schedule itself. In fact, the scheduling algorithm [16] first generates such an unfolding. In this section, we define the unfolding of a schedule and propose a procedure to construct such an unfolding from a schedule.

By schedule Property 5, vertex r has a directed path to every vertex of a schedule. Thus, starting from vertex r we can unfold a schedule $G(V, E, r)$ into a rooted, directed tree $G'(V', E', r')$, as described in Procedure 1. In future, objects in the unfolding of a schedule graph G will be denoted by decorating the corresponding objects from a schedule with $'$.

Procedure 1 Schedule unfolding

INPUT: a schedule (V, E, r) of a Petri net (N, M).

OUTPUT: returns the unfolding (V', E', r') of the schedule.

$V' \Leftarrow \emptyset, E' \Leftarrow \emptyset$

for all vertex $v \in V$ **do**

$visited(v) \Leftarrow false$

end for

$explore(r)$

Procedure *explore* (vertex v)

$visited(v) \Leftarrow true$

$V' \Leftarrow V' \cup \{v'\}$, $\theta(v') \Leftarrow v$

for all edge $(v, u) \in E$ **do**

$V' \Leftarrow V' \cup \{u'\}$, $E' \Leftarrow E' \cup \{(v', u')\}$, $\theta(u') \Leftarrow u$

if $visited(u) = false$ **then**

$explore(u)$

end if

end for

$visited(v) \Leftarrow false$

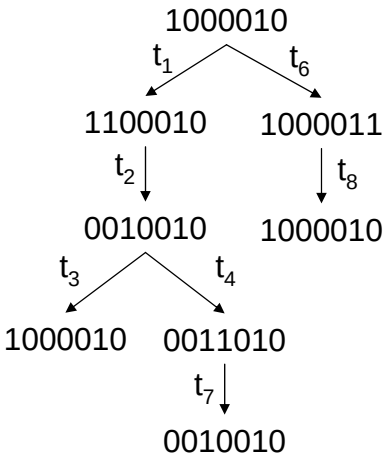


Figure 2.3: The unfolding of the schedule in Figure 1.15.

The unfolding uses a mapping $\theta : V' \to V$ with $\theta(r') = r$, and an extension of mapping μ (from schedule Property 2) as $\mu(v') = \mu(\theta(v'))$. There exists $v' \xrightarrow{t} u'$ in G' only if there exists $\theta(v') \xrightarrow{t} \theta(u')$ in G. The unfolding proceeds as a depth-first search. It terminates at a leaf vertex v' when there exists an ancestor u' of v' such that $\mu(v') = \mu(u')$. We call the object obtained by applying Procedure 1 the *unfolding* of a schedule.

For example, Figure 2.3 shows the unfolding of the schedule in Figure 1.15.

Lemma 2 *A schedule unfolding $G'(V', E')$ for a given schedule $G(V, E)$ is finite.*

Sketch of proof: The proof immediately follows from the finiteness of a schedule. One can make only a finite number of steps in the unfolding procedure before seeing the same marking repeating with some ancestor in the tree. Repeating the marking terminates the unfolding and keeps the generated prefix finite. □

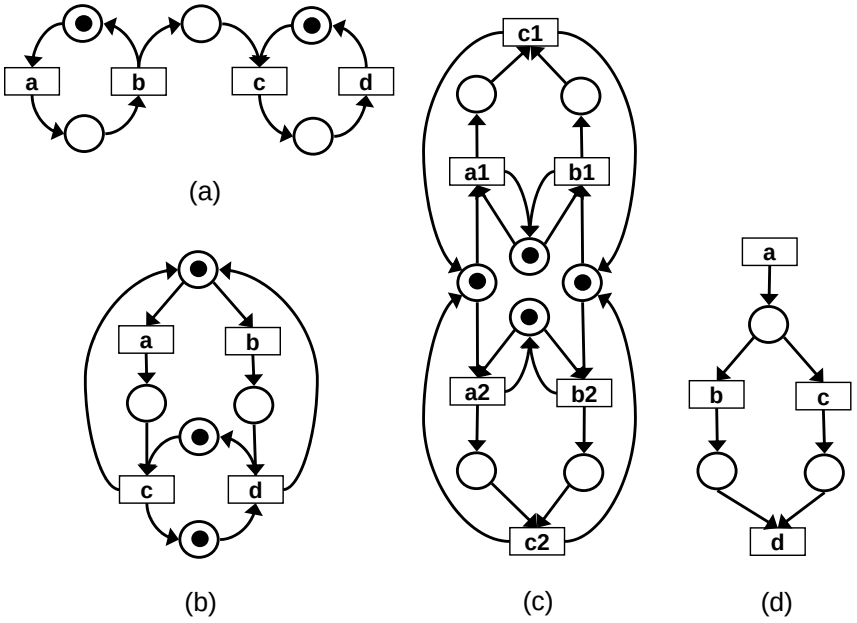


Figure 2.4: (a) an unbounded, schedulable Petri net, (b) a bounded, unschedulable Petri net, (c) a non-live, schedulable Petri net, (d) a live, reversible, unschedulable Petri net.

2.5 Schedulability

In this section, we formally define schedulability, and briefly discuss its relationship with other Petri net properties.

Definition 3 (Schedulability) *A Petri net* (N, M_0) *is said to be schedulable if there exist a reachable marking* $M \in R(N, M_0)$ *and a schedule of* (N, M)*. A Petri net* N *is said to be schedulable, if there exist a marking* M *of* N *and a schedule of* (N, M)*.*

Thus, a Petri net N is said to be unschedulable, if for any marking M of N there exits no schedule of (N, M).

In general, schedulability is independent of other Petri net properties, such as boundedness, liveness, and reversibility. We illustrate this with the examples shown in

Figure 2.4. Figure 2.4(a) shows an unbounded Petri net. It is unbounded because the firing sequence $(ab)^{\omega}$ will produce an infinite number of tokens in the place connecting transition b and c. However, it is schedulable and $(abcd)$ represents the cyclic firing sequence in the schedule of the Petri net. Figure 2.4(b) shows a bounded Petri net. For all reachable marking, the maximum number of tokens in a place is no greater than two. We will show later that it is not schedulable. Figure 2.4(c) models two dining philosophers. If both philosophers pick one chop stick, which is modelled by firing transition a_1, b_2 or a_2, b_1, then the system deadlocks and no one can obtain a pair. However, the Petri net is schedulable. The sequence $(a_1b_1c_1)$ and $(a_2b_2c_2)$ are cyclic firing sequences that could be contained in its schedules. Figure 2.4(d) shows a reversible, live Petri net. It is easy to check that no matter how many tokens are accumulated in a place, there always exists a firing sequence that consumes all of them. The Petri net is live because transition a is a source transition, thus always enabled. Firing a will enable transition b and c. Transition d can be enabled from any marking by the firing sequence (a, b, a, c). The Petri net is not schedulable, as we will show later.

2.6 False Path Problem

In practice, the unschedulability of a Petri net is often caused by a scheduler exploring false paths. False paths are the firing sequences of a Petri net that do not have corresponding execution traces of the modelled program. False paths exist because data-dependent controls in a program are modelled by non-deterministic free choices in a Petri net. Thus, once a scheduler reaches a choice, it will consider all possible outcomes.

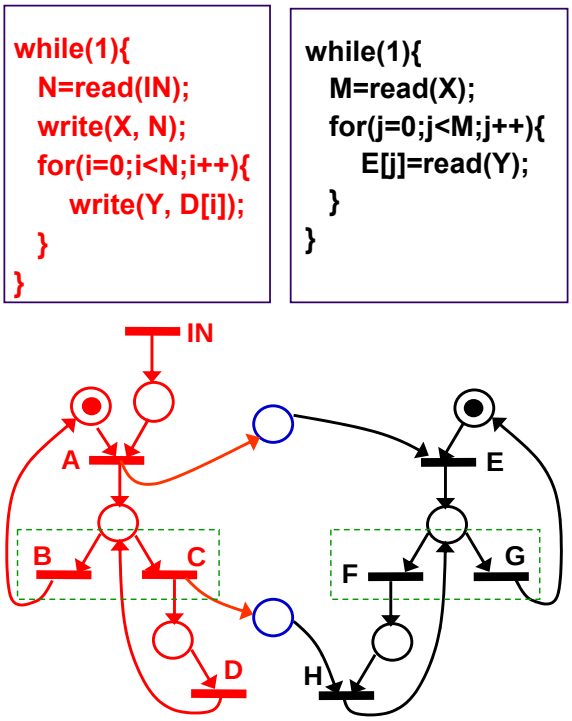


Figure 2.5: A concurrent program and its Petri net model.

We illustrate the false path problem with the example shown in Figure 2.5. The concurrent program describes a producer and a consumer process. The producer generates a data stream through a for-loop, and the consumer read the data stream also through a for-loop. The producer reads data, the number of iterations, from the environment, and passes it to the consumer. So both for-loops have the same number of iterations. Therefore, it is not necessary to consider all possible true/false combinations of the two conditions.

However, in the Petri net model, the two conditions are modelled as two independent non-deterministic choices. A scheduler will explore all possible outcomes of the two choices. This leads to false paths. As we will show later, the Petri net is not schedulable.

Arrigoni et al. [7] proposed an approach to the false path problem. It suggests

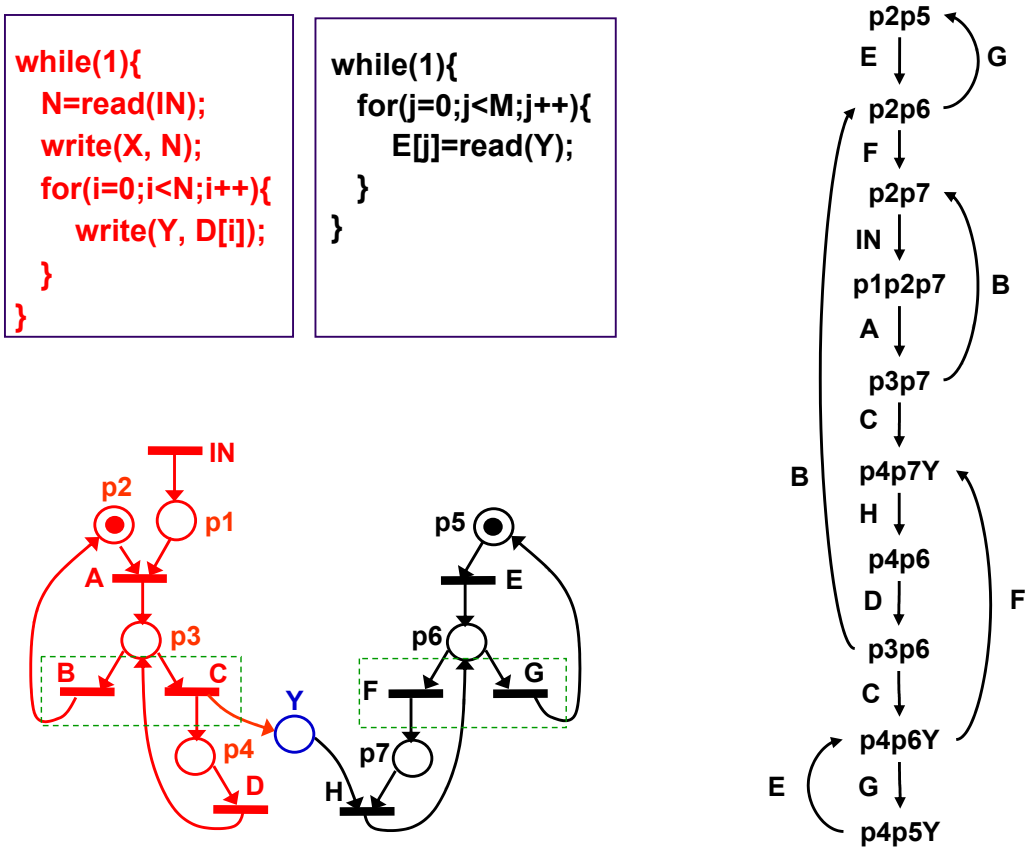


Figure 2.6: A concurrent program, its Petri net model, and its schedule.

to modify the original programs so that one of the two control structures having the same condition would not be modelled by a free choice. However, it requires designers to manually identify the correlated conditions in programs. In practice, such conditions are distributed in a number of concurrent processes. Identifying these conditions is a tedious, error-prone task. Thus, the approach is not practical. Standard static program analysis techniques, such as abstract interpretation [19], could be applicable. But the technique is known to have a high computation complexity, and can only deal with arithmetic operations on variables of a system.

Note that false paths do not always cause unschedulability. In other words, a schedule could contain a firing sequence that does not correspond to an execution trace of the underlying program. From scheduling point of view, such false paths are not harmful. We illustrate this with the following example. Figure 2.6 shows a concurrent program similar to the one in Figure 2.5. The difference is that the producer process does not pass the number of iterations to the consumer process. Thus, the two for-loops could have different number of iterations. The Petri net model of the concurrent program is shown at the bottom. The net, however, is schedulable, and one schedule is shown on the right. The schedule contains false paths. It is easy to see that a feasible execution of the program always first enters a for-loop and then exits the loop. Thus, starting from the initial state of the schedule, any firing sequence beginning with (E, G) does not have a corresponding execution trace of the program.

Our approach is different from the known ones. We explore the implication of the correlated conditions in the original programs to their structures in the corresponding Petri net models. That is, we study the structure of a Petri net and identify a structural property that causes unschedulability. By exposing the structural correlation among a set of free-choice sets, we get hints on the correlated conditions in original programs.

Chapter 3

The Cyclic Dependence Theorem

In this chapter, we introduce one of our key theoretic results regarding the schedulability analysis of Petri nets. The central concept is the dependence relation among transitions in FCSs. The connection between the mutual dependence of transitions and the behavior property of a Petri net was observed by Chen et al.[13], where the dependence relation was introduced to analyze the boundedness and liveness of a Petri net that is constructed by connecting live, bounded Petri nets through additional places. However, as we showed before, schedulability is independent of boundedness and liveness. Our dependence relation is defined, extended, and studied in a different context. Nevertheless, it is interesting to see the similarities of the central concept between two Petri net analysis techniques.

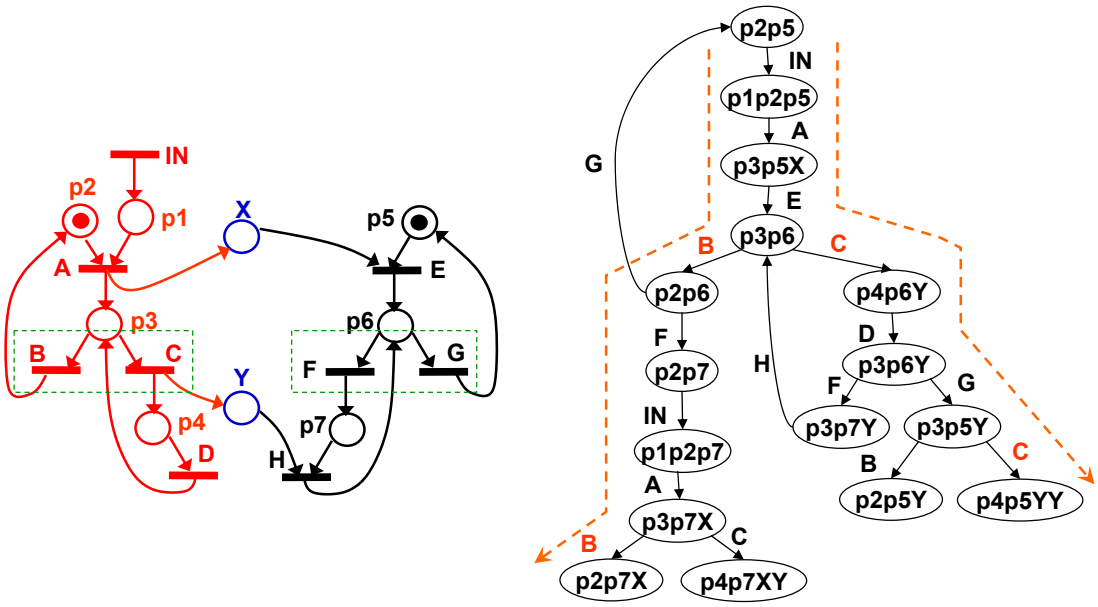


Figure 3.1: A Petri net and its "schedule".

3.1 A Motivational Example

We observe that regular and repeated structural patterns exist in schedules of Petri nets models of many applications. These patterns result in infinite paths that prevent a schedule to return to a designated set of states (initial state in particular), which causes unschedulability.

We illustrate our observations with an example. Figure 3.1 shows a Petri net, the same as the one shown in Figure 2.5, and its "schedule". The labelled diagraph is not strictly a schedule. It is constructed by a scheduling algorithm. We noticed that there exist two infinite paths starting from the initial state. They are indicated by the two dotted lines. The infinite paths exhibit certain repeated patterns. For example, transition B, F alternately appear on the left infinite path. It seems that there exits a underlying force

that drives a scheduler to enter an infinite state space. Firing transition B at marking $p3p6 \Rightarrow$ firing transition G at marking $p2p6 \Rightarrow$ firing transition F at marking $p2p6 \Rightarrow$ firing transition C at marking $p3p7X \Rightarrow$ firing transition B at marking $p3p7X \Rightarrow \ldots$.

As we will show, the phenomenon is driven by the definition of a schedule and, interestingly enough, a structural property of the Petri net.

3.2 Pairwise Transition Dependence Relation

We first introduce the pairwise transition dependence relation to give readers some intuition, and prove a proposition that relates the dependence relation to schedulability of a Petri net. We will extend the dependence relation later.

Definition 4 (Pairwise transition dependence) *A transition t of a Petri net N is said to be* dependent *on a transition t', if for each T-invariant $\boldsymbol{x}$ of N, $t \in \|\boldsymbol{x}\|$ implies $t' \in \|\boldsymbol{x}\|$. This is denoted by $t \rightarrowtail t'$.*

Since the set of T-invariants of a Petri net is often an infinite set, it is difficult to directly use the definition to prove the existence of dependence relation. We propose a necessary and sufficient condition for dependence, which allows us only to examine a finite set of T-invariants.

Lemma 3 *A transition t of a Petri net N is dependent on a transition t', if and only of for all minimal T-invariant $\boldsymbol{x} \in X_{min}$ of N, $t \in \|\boldsymbol{x}\|$ implies $t' \in \|\boldsymbol{x}\|$.*

Proof: (only-if): We proceed by contradiction. If there exists a minimal T-invariant $\boldsymbol{x}$ that contains t but not transition t', then $\boldsymbol{x}$ is a counterexample for $t \rightarrowtail t'$.

(if): By definition, each T-invariant $\boldsymbol{x} = \sum_{i=1,k} a_i \boldsymbol{x}_i$, where $a_i \geq 0, \boldsymbol{x}_i \in X_{min}$. If $t \in \|\boldsymbol{x}\|$, there exist a rational a_j and a minimal T-invariant $\boldsymbol{x}_j$ such that $a_j > 0$ and $t \in \|\boldsymbol{x}_j\|$. Note that $t \in \|\boldsymbol{x}_j\|$ implies $t' \in \|\boldsymbol{x}_j\|$. Since $a_j > 0, a_i \geq 0, i = 1, 2, \cdots, k$, $t' \in \|\boldsymbol{x}\|$ follows. □

Obviously, the pairwise transition dependence is a binary relation over the set of transitions of a Petri net.

Lemma 4 *The pairwise transition dependence relation is reflexive and transitive, but not symmetric in general.*

Proof: It is trivial to show that any transition is dependent on itself.

Now we show that the relation is reflexive by contradiction. Assume $t_1 \rightarrowtail t_2$ and $t_2 \rightarrowtail t_3$. If there exists a T-invariant x such that $t_1 \in \|\boldsymbol{x}\|$ and $t_3 \notin \|\boldsymbol{x}\|$, then there are two cases possible.

Case 1. $t_2 \in \|\boldsymbol{x}\|$.

$t_2 \in \|\boldsymbol{x}\|$ and $t_3 \notin \|\boldsymbol{x}\|$ conflict with $t_2 \rightarrowtail t_3$.

Case 2. $t_2 \notin \|\boldsymbol{x}\|$.

$t_1 \in \|\boldsymbol{x}\|$ and $t_2 \notin \|\boldsymbol{x}\|$ conflict with $t_1 \rightarrowtail t_2$.

Thus, no such x exists. That is $t_1 \rightarrowtail t_3$.

Now we show the dependence relation is not symmetric. Let N be a ordinary Petri net that contains one place and three transitions t_1,t_2, and t_3, where t_1 is the input transition of p and t_2, t_3 are both output transitions of p. There are two minimal T-invariants with support $\{t_1, t_2\}, \{t_1, t_3\}$. Clearly, $t_2 \rightarrowtail t_1$, but $t_1 \rightarrowtail t_2$ does not hold. Any T-invariant with support $\{t_1, t_3\}$ is a counterexample. □

3.3 Proof of a Special Case of Cyclic Dependence Theorem

Now we prove a proposition that relates the pair-wise dependence with the schedulability of a Petri net. Note that the proposition is formulated simple enough to illustrate the basic idea. Another proof based on schedule unfolding is provided in the Appendix. We will extend the notion of dependence and reformulate its relation with schedulability in the next two sections.

Proposition 1 *Given a Petri net N with two FCSs $S_1 = \{t_1, t_1'\}$, $S_2 = \{t_2, t_2'\}$, if $t_1' \rightarrowtail t_2$ and $t_2' \rightarrowtail t_1$, then for any marking M of N, there exists no schedule of (N, M) involving S_1 or S_2.*

Proof: We prove by contradiction. We assume that there exists a schedule $G(V, E, r)$ of N. The proof proceeds by showing the validity of at least one of the two following statements:

I1: G contains an infinite path $v_1 \xrightarrow{t_1'} y_1 \rightsquigarrow v_2 \xrightarrow{t_1'} y_2 \cdots$, such that the path from v_1 to v_k $(k = 2, 3, \dots)$ does not contain transition t_2. For succinctness, we say a path contains a transition t if the path contains vertices u, v such that $u \xrightarrow{t} v$.

I2: G contains an infinite path $u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow u_2 \xrightarrow{t_2'} z_2 \cdots$, such that the path from u_1 to u_k $(k = 2, 3, \dots)$ does not contain transition t_1.

Without loss of generality, we may assume that G involves $S_1 = \{t_1, t_1'\}$. It implies that there exists $v_1 \in G$ such that $v_1 \xrightarrow{t_1'} y_1$.

Step 1: By schedule Property 5, there exists a directed cycle $\pi_1 : r \rightsquigarrow v_1 \xrightarrow{t_1'} y_1 \rightsquigarrow r$. Since the firing count vector of a cyclic firing sequence is a T-invariant, $t_1' \rightarrowtail t_2$ implies that π_1 contains at lease one vertex u such that $u \xrightarrow{t_2}$. Let u_1 be the u that is first reached by

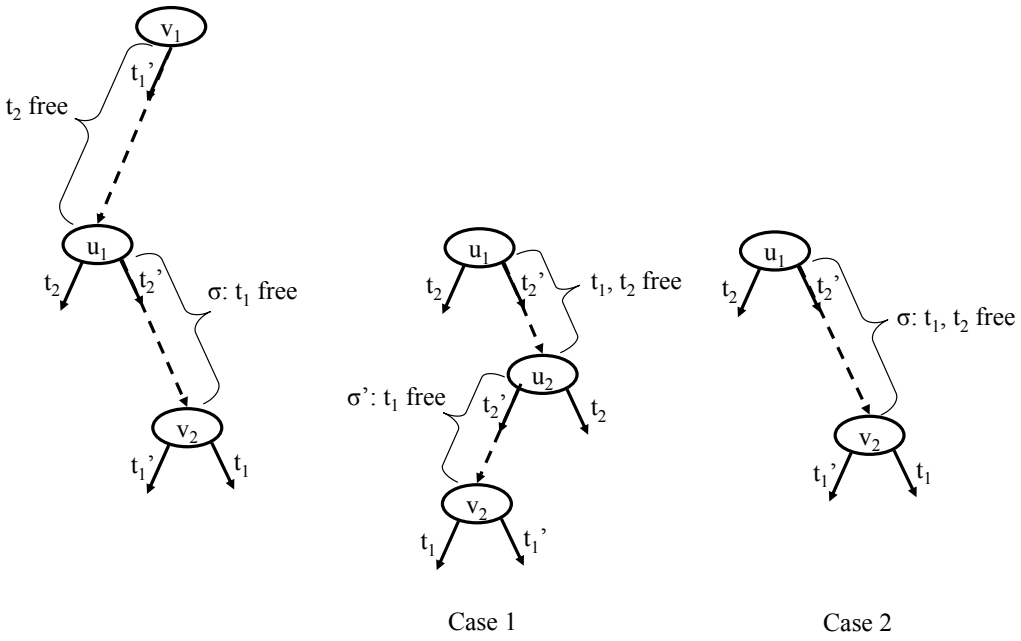


Figure 3.2: Illustration of Proof.

a traversal from v_1 along π_1. Clearly, the subpath $v_1 \rightsquigarrow u_1$ of π_1 does not contain transition t_2.

Step 2: Since t_2 and t_2' are in a FCS, by schedule Property 3 there exists $u_1 \xrightarrow{t_2'} z_1$. Similar to the reasoning in Step 1, by Property 5, there exists a directed cycle $\pi_2 : r \rightsquigarrow u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow r$. $t_2' \rightarrowtail t_1$ implies that π_2 contains at least one vertex v such that $v \xrightarrow{t_1}$. Let v_2 be the v that is first reached by a traversal from u_1 along π_2. Clearly, the subpath $u_1 \rightsquigarrow v_2$ of π_2 does not contain transition t_1.

Step 3: Let us consider the path σ from u_1 to v_2. Two cases are possible.

Case 1: σ contains t_2.

That is, σ contains at least one vertex u such that $u \xrightarrow{t_2}$. Let u_2 be the u that is first reached by a traversal from u_1 along σ. Clearly, the subpath $u_1 \rightsquigarrow u_2$ of σ does not contain

transition t_1 and t_2. By schedule Property 3, $S_2 = \{t_2, t_2'\}$ implies there exists $u_2 \xrightarrow{t_2'} z_2$. By schedule Property 5, there exists a directed cycle $\pi_3 : r \rightsquigarrow u_2 \xrightarrow{t_2'} z_2 \rightsquigarrow r$. $t_2' \rightarrowtail t_1$ implies that π_3 contains at least one vertex v such that $v \xrightarrow{t_1}$. Redefine v_2 be the v that is first reached by a traversal from u_2 along π_3. Now we examine the subpath $\sigma' = u_2 \rightsquigarrow v_2$ of π_3. Just like the path σ, there are two cases possible. σ' either contains t_2 or does not. If it contains t_2, then *Case 1* is repeated. Clearly, the same procedure can be repeated forever. *Case 1* is infinitely repeated implies that there exists an infinite path $u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow u_2 \xrightarrow{t_2'} z_2 \rightsquigarrow \ldots$ does not contain t_1. Thus, I2 is proved. If *Case 2* occurs at u_k, then there exists a path $u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow \ldots u_k \xrightarrow{t_2'} z_k \rightsquigarrow v_2$ that does not contain t_2. Since the subpath $v_1 \rightsquigarrow u_1$ of π_1 does not contain t_2, there exists a path $v_1 \xrightarrow{t_1'} y_1 \rightsquigarrow u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow \ldots u_k \xrightarrow{t_2'} z_k \rightsquigarrow v_2 \xrightarrow{t_1'} y_2$ that does not contain t_3. Repeat the same procedure of v_1 at v_2, I1 is proved.

Case 2: σ does not contain t_2.

Immediately follows that there exists a path $v_1 \xrightarrow{t_1'} y_1 \rightsquigarrow u_1 \xrightarrow{t_2'} z_1 \rightsquigarrow v_2 \xrightarrow{t_1'} y_2$ that does not contain transition t_2. Repeat the same procedure of v_1 at v_2, I1 is proved.

If I1 is valid, we show that $v_1, v_2, \ldots$ are distinct vertices in G. If there exist two vertices $v_i, v_j, 1 \leq i < j$ corresponding to the same vertex in G, there exists a directed cycle $v_i \xrightarrow{t_1'} y_i \rightsquigarrow v_{i+1} \xrightarrow{t_1'} y_{i+1} \cdots \rightsquigarrow v_i$, which contains transition t_1' but not t_2. This contradicts with $t_1' \rightarrowtail t_2$. $v_1, v_2, \ldots$ are distinct implies that there are an infinite number of vertices in G, which contradicts with the finiteness of a schedule. Similar arguments hold if I2 is valid.

□

Note that the proof does not rely on schedule Property 4, the notion of progress. In other words, even if in certain circumstances the progress of a schedule is defined differently

from what we defined, the proposition still holds.

We could use Proposition 1 to prove that some Petri nets are not schedulable. Figure 2.5 shows a Petri net model that satisfies the condition of Proposition 1. It has two FCSs, $\{B, C\}$ and $\{F, G\}$, and two minimal T-invariants with supports $\{IN, A, B, E, G\}$ and $\{C, D, F, H\}$. It is easy to verify that $C \rightarrowtail F$ and $G \rightarrowtail B$. Thus, according to Proposition 1 there exists no schedule of the Petri net that involves either $\{B, C\}$ or $\{F, G\}$. Note that in this example, there exist much more dependence relations between the transitions in the two FCSs than what is needed to prove unschedulability. In fact, $C \rightarrowtail F$, $F \rightarrowtail C$, $G \rightarrowtail B$ and $B \rightarrowtail G$. Either $(C \rightarrowtail F) \wedge (G \rightarrowtail B)$ or $(F \rightarrowtail C) \wedge (B \rightarrowtail G)$ is sufficient to prove that no schedule of the Petri net involves either $\{B, C\}$ or $\{F, G\}$.

3.4 Extending Pairwise Dependence

By observing the "chain-like" dependence formulation in Proposition 1, we extend the dependence relation to more than two FCSs.

Definition 5 (Cyclic pairwise dependence relation) *A set of FCSs* $\{t_1, t_1'\}, \ldots, \{t_k, t_k'\}$ *of a Petri net is said to be in cyclic pairwise dependence relation, if there exist dependence relations* $(t_1' \rightarrowtail t_2) \wedge (t_2' \rightarrowtail t_3), \cdots, (t_k' \rightarrowtail t_1)$.

In the case of a single FCS $\{t_1, t_2\}$, the formulation is $t_1 \rightarrowtail t_2$. In the case of three FCSs $\{t_1, t_2\}, \{t_3, t_4\}, \{t_5, t_6\}$, the formulation is $(t_2 \rightarrowtail t_3) \wedge (t_4 \rightarrowtail t_5) \wedge (t_6 \rightarrowtail t_1)$.

Theorem 1 *Given a Petri net N with a set $\mathbb{S}$ of FCSs, if $\mathbb{S}$ is in cyclic pairwise dependence relation, for any marking M of N there exists no schedule of (N, M) involving a FCS in $\mathbb{S}$.*

Sketch of proof: The proof can be done in a way similar to the proof of Proposition 1. Given FCSs $\{t_1, t_1'\}, \ldots, \{t_k, t_k'\}$ with $t_1' \rightarrowtail t_2 \wedge t_2' \rightarrowtail t_3, \cdots, t_k' \rightarrowtail t_1$. We assume that there exists a schedule $G(V, E, r)$ of N. The proof proceeds by showing the validity of at least one of the following statements:

I1: G contains an infinite path $v_1 \xrightarrow{t_1'} y_1 \rightsquigarrow v_2 \xrightarrow{t_1'} y_2 \cdots$, such that the path from v_1 to v_k $(k = 2, 3, \ldots)$ does not contain transition t_2.

...

Ik: G contains an infinite path $u_1 \xrightarrow{t_k'} z_1 \rightsquigarrow u_2 \xrightarrow{t_k'} z_2 \cdots$, such that the path from u_1 to u_k $(k = 2, 3, \ldots)$ does not contain transition t_1.

Then, we show that each statement implies the existence of an infinite number of distinct vertices in G. □

The Petri net shown in Figure 2.4(b) has one FCS $\{a, b\}$ and one minimal T-invariant with support $\{a, b, c, d\}$. $\{a, b\}$ is in cyclic dependence relation because $a \rightarrowtail b$. Since every T-invariant contains a transition in $\{a, b\}$, we conclude that the Petri net is not schedulable for any marking of the net. Similarly, in Figure 2.4(d) there is only one FCS $\{b, c\}$, one minimal T-invariant with support $\{a, b, c, d\}$. Any T-invariant of the net contains a transition from the FCS. Thus, the Petri net is not schedulable.

3.5 General Transition Dependence Relation

We defined pairwise dependence relation and based on that a sufficient condition for unschedulability is proposed and proved. However, the power of pairwise dependence relation is limited. Figure 3.3 shows a Petri net that contains four FCSs $\{B, C\}$, $\{F, G\}$, $\{I, J\}$, $\{L, M\}$, and five minimal T-invariants with supports $\{IN, A, B, I, E, G, M\}$, $\{C, D, F, H\}$, $\{C, D, L, N\}$, $\{J, K, F, H\}$, $\{J, K, L, N\}$. Note that there exists no cyclic pairwise dependence relation among the transitions in the FCSs. But as we show later, the transitions are in a general notion of cyclic dependence, and consequently we can prove unschedulability of the net. It is worth noticing that the Petri net shown in Figure 3.3 resembles the Petri net shown in Figure 2.5. It models a producer and a consumer process which include an additional loop structure, compared with the program shown in Figure 3.3. Also note that the Petri net is not a free-choice Petri net. The arc from the place modelling the loop-communication channel to transition H is neither a unique outgoing arc of the place, nor a unique incoming arc of transition H.

Definition 6 (General transition dependence relation) *A transition t of a Petri net N is said to be* dependent *on a set S of transitions, if for each T-invariant $\boldsymbol{x}$ of N, $t \in \|\boldsymbol{x}\|$ implies $\exists t' \in S : t' \in \|\boldsymbol{x}\|$. This is denoted by $t \rightarrowtail S$.*

The pairwise transition dependence relation can be viewed as a special case of the general transition dependence relation, or simply *dependence relation* with $|S| = 1$.

Lemma 5 *The general transition dependence relation is monotonically non-decreasing.*

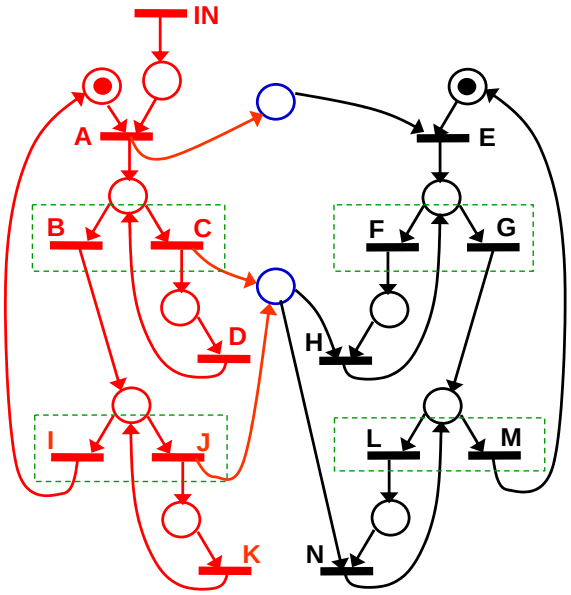


Figure 3.3: A Petri net containing FCSs in cyclic (general) dependence relation.

Proof: We want to show that if $t \rightarrowtail S$, then for each S' such than $S \subseteq S'$, $t \rightarrowtail S'$. $t \rightarrowtail S$ implies for all T-invariant x such that $t \in \|\boldsymbol{x}\|$ there exists $t' \in S$ such that $t' \in \|\boldsymbol{x}\|$. $S \subseteq S'$ implies that $t' \in S'$. Thus, there exists $t' \in S'$ such that $t' \in \|\boldsymbol{x}\|$. □

This implies a transition t of a Petri net is dependent on any set of transitions that contains t. Clearly, the dependence relation alone can not characterize the cyclic dependence relation used in Proposition 1. We introduce the concept of cover of FCSs.

Definition 7 (Cover of a set of FCSs) *A cover S of a set $\mathbb{S}$ of FCSs is a minimum subset of transitions such that for each FCS $F \in \mathbb{S}$, there exists a transition $t \in S \cap F$.*

Lemma 6 *A cover of a set of FCSs contains exactly one transition from each FCS.*

Proof: Clearly, a cover S of $\mathbb{S}$ must contain at least one transitions in each FCS in $\mathbb{S}$. If S contains two transition $\{t_i, t_i'\}$ of a FCS $F_i \in \mathbb{S}$, then $S' = S \backslash \{t\}$ is a subset of S that

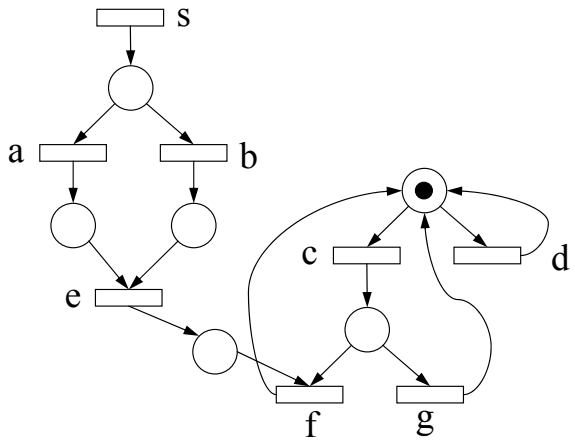


Figure 3.4: A Petri net containing a FCS in cyclic dependence relation.

satisfies for each FCS $F \in \mathbb{S}$, there exists a transition $t \in S' \cap F$. This conflicts with the definition that S is a minimum set that satisfies the condition. □

Definition 8 (Cyclic dependence relation) *A set $\mathbb{S}$ of FCSs of a Petri net N is said to be in cyclic dependence relation, if there exists a cover S of $\mathbb{S}$, such that for each transition $t \in S$, $t \rightarrowtail \mathbb{S} \backslash S$.*

A FCS is said to be *cyclic dependent* if there exists a set $\mathbb{S}$ of FCSs, such that $\mathbb{S}$ contains the FCS, and is in cyclic dependence relation. Note that although the general dependence relation is monotonic, the cyclic dependence relation is not monotonic in general. Hence we can not prove the existence of the cyclic dependence relation in a set of FCSs by proving the existence of the relation for its subsets and vice versa.

We illustrate this with the following examples. Figure 3.4 shows a Petri net with two FCSs $\{a, b\}, \{c, d\}$. There are three minimal T-invariants with supports $\{s, a, b, e, c, f\}$, $\{c, g\}, \{d\}$. Clearly, $a \rightarrowtail b$. Thus, $\{a, b\}$ is in a cyclic dependence relation. However, it is

easy to verify that the two FCSs are not in a cyclic dependence relation.

Figure 2.5 shows a Petri net with two FCSs, $\{B, C\}$ and $\{F, G\}$. There are two minimal T-invariants with supports $\{IN, A, B, E, G\}$ and $\{C, D, F, G\}$. It is easy to verify that there exists no cyclic dependence relations in either of two FCSs. But we showed earlier the two FCSs together are in a cyclic dependence relation.

It implies that given a set of FCSs, in order to find if the set contains a cyclic dependent FCS one needs to examine potentially all subsets of the set, which is an exponential of the cardinality of the set. We will discuss this in more details in next chapter.

Theorem 2 (Cyclic dependence theorem) *No schedule of a Petri net involves a cyclic dependent FCS.*

Sketch of proof: The proof can be done in a way similar to the proof of Proposition 1 and Theorem 1. Given FCSs $\{t_1, t_1'\}, \ldots, \{t_k, t_k'\}$ and a cover $S = \{t_1, t_2, \ldots t_k\}$, $(t_1 \rightarrowtail S') \wedge (t_2 \rightarrowtail S'), \cdots, (t_k \rightarrowtail S')$, where $S' = \{t_1', t_2', \ldots t_k'\}$. We assume that there exists a schedule $G(V, E, r)$ of N. The proof proceeds by showing the validity of at least one of the following statements:

I1: G contains an infinite path $v_1 \xrightarrow{t_1} y_1 \rightsquigarrow v_2 \xrightarrow{t_1} y_2 \cdots$, such that the path from v_1 to v_k $(k = 2, 3, \ldots)$ does not contain any transition in S'.

$\ldots$

Ik: G contains an infinite path $u_1 \xrightarrow{t_k} z_1 \rightsquigarrow u_2 \xrightarrow{t_k} z_2 \cdots$, such that the path from u_1 to u_k $(k = 2, 3, \ldots)$ does not contain any transition in S'.

Then, we show that each statement implies the existence of an infinite number of distinct vertices in G. □

Clearly, Proposition 1 and Theorem 1 can be viewed as special cases of Theorem 2. Thus, Theorem 2 is more powerful than them to establish unschedulability. Consider the Petri net in Figure 3.3. We have showed that Proposition 1 and Theorem 1 can not be used to prove that it is not schedulable. However, there exists general dependence relations, and furthermore a cyclic dependence relation for the set of all FCSs in the net. Let $S = \{C, G, J, M\}$ be a cover of the $\mathbb{S}$, then $\mathbb{S} \backslash S = \{B, I, F, L\}$. For each transitions t in S, $t \rightarrowtail \mathbb{S} \backslash S$. Thus, by Theorem 2, for any initial marking there exists no schedule of the Petri net involving any of the FCSs.

Chapter 4

Algorithms Checking Cyclic Dependence

The cyclic dependence relation is central to the schedulability analysis we have presented so far. To apply the theoretic results, efficient methods to check the existence of cyclic dependence relation are needed. In this chapter, we discuss two methods. One is based on the linear programming (not integer linear programming). The other is based on generating sets of T-invariants.

4.1 Checking Cyclic Dependence with Linear Programming

Although the dependence relation is defined on the set of T-invariants of a Petri net, interestingly enough, such a set does not have to be explicitly computed to check the relation. We propose an exact algorithm based on linear programming to check the cyclic dependence relation. Note that though T-invariants are nonnegative *integer* solutions to a

homogenous linear system, it is not necessary to solve an integer programming problem to check the relation.

Algorithm 2 Checking cyclic dependence relation using linear programming

INPUT: $\boldsymbol{A}$: the incidence matrix of a Petri net, $\mathbb{S}$: the set of FCSs to be checked.

OUTPUT: returns TRUE if there exists a cyclic dependence relation in $\mathbb{S}$, FALSE otherwise.

1: **for all** covers S of $\mathbb{S}$ **do**
2: $dependent \Leftarrow$ TRUE
3: **for all** $t_i \in S$ **do**
4: $LP \Leftarrow (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0) \cap (\boldsymbol{x} \geq 0) \cap (x_i > 0) \cap (x_j = 0, \forall j, t_j \in \mathbb{S} \backslash S)$
5: **if** $LP \neq \emptyset$ **then**
6: $dependent \Leftarrow$ FALSE
7: break
8: **end if**
9: **end for**
10: **if** $dependent$ = TRUE **then**
11: return TRUE
12: **end if**
13: **end for**
14: return FALSE

Given a Petri net N, its incidence matrix $\boldsymbol{A}$, and a set $\mathbb{S}$ of FCSs of N to be checked, the algorithm iterates through all possible covers of $\mathbb{S}$ till one cover leads to a cyclic dependence relation. For each cover, a feasibility problem of linear programming

is constructed. As proved in Theorem 3, a solution to the feasibility problem provides a counterexample to the dependence relation. If no solution is found, the dependence relation holds. Note that whether a cover S leads to a cyclic dependence relation for $\mathbb{S}$ can be checked in polynomial-time.

Theorem 3 *A transition t_i is dependent on a set S of transitions if and only if the following linear system has no solution:*

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0$$

$$\boldsymbol{x} \geq 0$$

$$x_i > 0$$

$$x_j = 0 \ \forall j, t_j \in S$$

Proof: (if): We prove the contrapositive. If $t_i \rightarrowtail S$ does not hold, by definition, there exists a T-invariant $\boldsymbol{x} \in \mathbb{N}^{|T|}$, such that $x_i \geq 1$ and for each $t_j \in S, x_j = 0$.

(only-if): We prove the contrapositive. Since the incidence matrix $\boldsymbol{A}$ is an integer matrix, if there exists a real vector that satisfies all the constraints, then there exist a rational vector $\boldsymbol{x}$ also satisfying the constraints. Let θ be a common multiple of all the denominators of the elements of $\boldsymbol{x}$ and let $\boldsymbol{x'} = \theta\boldsymbol{x}$. By definition, $\boldsymbol{x'}$ is a T-invariant, and $\boldsymbol{x'} \geq 0, x'_i > 0$, for each $t_j \in S, x'_j = 0$. Thus, $\boldsymbol{x'}$ is a counterexample for $t_i \rightarrowtail S$. □

The complexity of checking a cyclic dependence using Algorithm 2 is exponential on the number of FCSs. This comes from the need to check explicitly all possible covers of the set $\mathbb{S}$. The next chapter presents a more efficient way to establish unschedulability based on checking the rank of the incidence matrix of a Petri net.

4.2 Checking Cyclic Dependence with Generating Sets

In this section, we show that the cyclic dependence relation among a set of FCSs could be established by simply counting the number of minimal T-invariants that contains transitions in the FCSs. If the number is less or equal to the number of FCSs, then the FCSs contains a cyclic dependence FCS.

Definition 9 (Generating set) *A set X_g of T-invariants $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k\}$ is a generating set of all T-invariants over Q^+, the set of non-negative rationals, if for all T-invariant $\boldsymbol{x}$, there exist $a_1, a_2, \ldots a_k \in Q^+$, such that $\boldsymbol{x} = a_1\boldsymbol{x}_1 + a_2\boldsymbol{x}_2 + \cdots + a_k\boldsymbol{x}_k$.*

We call the T-invariants in a generating set *generators*. It is easy to verify that both the set of minimal T-invariants and the set of minimal-support T-invariants are generating sets of all T-invariants over Q^+. The two generating sets can be efficiently computed using algorithms proposed by Martinez and Silva [39], Anna and Trigila [6].

We show that dependence relation could be derived by examining a generating set.

Lemma 7 *A transition t is dependent on a set S of transitions, $t \rightarrowtail S$, if and only if there exists a generating set X_g of T-invariants over Q^+, such that for all generators $\boldsymbol{x}_g \in X_g, t \in \|\boldsymbol{x}_g\|$ implies $\exists t' \in S, t' \in \|\boldsymbol{x}_g\|$.*

Proof: (only-if): We proceed by contradiction. If there exists a generating set X_g and a generator $\boldsymbol{x}_g \in X_g$ that contains t but none of the transitions in S, then $\boldsymbol{x}_g$ is a counterexample for $t \rightarrowtail S$.

(if): By definition, each T-invariant $\boldsymbol{x} = \sum_{i=1,k} a_i\boldsymbol{x}_i$, where $a_i \geq 0, \boldsymbol{x}_i \in X_g$. If $t \in \|\boldsymbol{x}\|$, there exist a rational a_j and a generator $\boldsymbol{x}_j$ such that $a_j > 0$ and $t \in \|\boldsymbol{x}_j\|$. Note

that $t \in \|\boldsymbol{x}_j\|$ implies $\exists t' \in S, t' \in \|\boldsymbol{x}_j\|$. Since $a_j > 0, a_i \geq 0, i = 1, 2, \cdots, k$, $t' \in \|\boldsymbol{x}\|$ follows. □

In the sequel, we say a FCS is *involved* in a T-invariant or a T-invariant *involves* a FCS, if there exists a transition in the FCS that corresponds to a non-zero entry in the T-invariant.

Theorem 4 *A set $\mathbb{S}$ of FCSs contains a cyclic dependent FCS if there exists a generating set X_g of all T-invariants over Q^+, such that $|X_g(\mathbb{S})| \leq |\mathbb{S}|$, where $X_g(\mathbb{S})$ is the set of generators in X_g that involves at least one FCS in $\mathbb{S}$.*

Proof: We proceed by induction on $|\mathbb{S}|$.

For $|\mathbb{S}| = 1$, $|X_g(\mathbb{S})| \leq |\mathbb{S}|$ implies $X_g(\mathbb{S}) = \{\boldsymbol{x}_g\}$ or $\emptyset$. Let $\{t, t'\}$ be the FCS in $\mathbb{S}$. If $X_g(\mathbb{S}) = \emptyset$, $t' \rightarrowtail t$ trivially holds. If $X_g(\mathbb{S}) = \{\boldsymbol{x}_g\}$, then without loss of generality, assume $t \in \|\boldsymbol{x}_g\|$. Clearly, no matter $t' \in \|\boldsymbol{x}_g\|$ or not, $t \in \|\boldsymbol{x}_g\|$ holds. By Lemma 7, this implies $t' \rightarrowtail t$. Thus, $FCS = \{t, t'\}$ is in a cyclic dependence relation.

Assume the theorem holds for $|\mathbb{S}| = 2, ..., n-1$, we show it also holds for $|\mathbb{S}| = n$. There are two cases possible.

Case 1. There exists $\mathbb{S}' \subset \mathbb{S}$ such that $|X_g(\mathbb{S}')| \leq |\mathbb{S}'|$. $\mathbb{S}' \subset \mathbb{S}$ implies $|\mathbb{S}'| \leq n-1$. By the assumption from previous induction steps, $|X_g(\mathbb{S}')| \leq |\mathbb{S}'|$ implies that $\mathbb{S}'$ contains a cyclic dependent FCS. Since $\mathbb{S}' \subset \mathbb{S}$, $\mathbb{S}$ contains a cyclic dependent FCS.

Case 2. For all $\mathbb{S}' \subset \mathbb{S}, |X_g(\mathbb{S}')| > |\mathbb{S}'|$.

1. First, we show that $|X_g(\mathbb{S})| = |\mathbb{S}|$. Choose $\mathbb{S}' \subset \mathbb{S}$ such that $|\mathbb{S}'| = |\mathbb{S}| - 1$. Then $|X_g(\mathbb{S}')| > |\mathbb{S}'|$ implies $|X_g(\mathbb{S}')| \geq |\mathbb{S}|$. And $\mathbb{S}' \subset \mathbb{S}$ implies $|X_g(\mathbb{S}')| \leq |X_g(\mathbb{S})|$. Thus, we have $|X_g(\mathbb{S})| \geq |\mathbb{S}|$. Since $|X_g(\mathbb{S})| \leq |\mathbb{S}|$, $|X_g(\mathbb{S})| = |\mathbb{S}|$ follows.

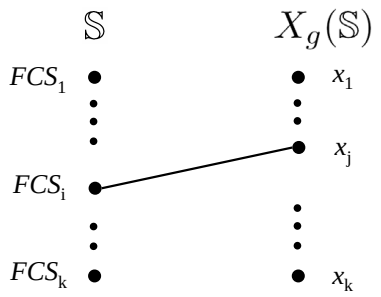


Figure 4.1: Bipartite graph of a set of FCSs and the generators that involves the FCSs.

2. Then, we show that $\mathbb{S}$ is in cyclic dependence relation. For all $\mathbb{S}' \subset \mathbb{S}, |X_g(\mathbb{S}')| > |\mathbb{S}'|$ and $|X_g(\mathbb{S})| = |\mathbb{S}|$ implies that for all $\mathbb{S}' \subseteq \mathbb{S}, |X_g(\mathbb{S}')| \geq |\mathbb{S}'|$.

   Construct a bipartite graph $G(\mathbb{S}, X_g(\mathbb{S}), E)$ as illustrated by Figure 4.1. There exists an edge $(FCS_i, x_j) \in E$ if and only if generator x_j involves a free-choice set FCS_i. Since $|X_g(\mathbb{S})| = |\mathbb{S}|$ and for all $\mathbb{S}' \subseteq \mathbb{S}, |X_g(\mathbb{S}')| \geq |\mathbb{S}'|$, by *Philip Hall's Theorem*[29], there exists a *complete (perfect) matching* for $\mathbb{S}$ and $X_g(\mathbb{S})$. By definition, for each match $(FCS, \boldsymbol{x})$ in a complete matching, there exists a transition t that is both contained in FCS and $\|\boldsymbol{x}\|$. Let S denote the set of such transitions for a complete matching. Clearly, every generator $\boldsymbol{x} \in X_g(\mathbb{S})$ contains at least one transition in S, and S contains exactly one transition from each FCS. Since each FCS contains two transitions, $\mathbb{S} \backslash S$ is a cover of $\mathbb{S}$. Denote it S'. Since for all $\boldsymbol{x} \in X_g(\mathbb{S})$ there exists t in S such that $t \in \|\boldsymbol{x}\|$, then

$$\forall t \in S', \forall \boldsymbol{x} \in X_g(\mathbb{S}), t \in \|\boldsymbol{x}\| \Rightarrow \exists t' \in S, t' \in \|\boldsymbol{x}\|.$$

   By Lemma 7, it implies that there exists a cover S' of $\mathbb{S}$ such that for $\forall t \in S', t \rightarrowtail S$. Thus, $\mathbb{S}$ is in cyclic dependence relation. □

Theorem 4 provides a sufficient condition for the existence of a cyclic dependent FCS in a set of FCSs. Once a set of FCSs of a Petri net and a generating set satisfying the condition are identified, then by Theorem 2 no schedule of the net involves such a set of FCSs. Thus, we could use Theorem 4 to prove unschedulability. In practice, we usually examine two generates sets of T-invariants of a Petri net, the set of minimal T-invariants and the set of minimal-support T-invariants.

Consider the Petri net shown in Figure 2.5. It has two FCSs and two minimal T-invariants. Thus the two FCSs contain a cyclic dependent FCS, and consequently no schedule of the Petri net involves the two FCSs.

Although checking the condition of Theorem 4 is easy once a generating set is computed, Theorem 4 is weaker than Theorem 3 to prove unschedulability. There are cases where there exists a cyclic dependent FCS in a set of FCSs, but can not be derived by examining the set of minimal (or minimal-support) T-invariants. Consider the Petri net shown in Figure 3.3. It has four FCSs $\{B, C\}$, $\{F, G\}$, $\{I, J\}$, $\{L, M\}$, and five minimal T-invariants with supports $\{IN, A, B, I, E, G, M\}$, $\{C, D, F, H\}$, $\{C, D, L, N\}$, $\{J, K, F, H\}$, $\{J, K, L, N\}$. Note that the set of minimal T-invariants is also the set of minimal-support T-invariants. It is easy to verify that no subset of FCSs satisfy the condition of Theorem 4. However, the existence of a cyclic dependent FCS in the Petri net can be established by Theorem 3 and the rank theorem, which we discuss in next chapter.

Chapter 5

The Rank Theorem

So far our schedulability analysis is based on the cyclic dependence relation. In this chapter, we introduce another important approach. We show how the incidence matrix, particularly its rank, is related to schedulability.

5.1 Related Work

The connection between behavioral properties of Petri nets and the rank of the incidence matrix was first observed for free-choice Petri nets [22]. The rank property shown there states that a Petri net of a specific class of free-choice nets has a live and bounded marking if and only if the number of conflict clusters exceeds the rank of the incidence matrix by one. In our terminology, conflict clusters of free-choice nets are either free-choice sets or transitions which are not in conflict with any other transition. Since we assume that free-choice sets have exactly two transitions, the number of all transitions equals the number of conflict clusters plus the number of conflict sets. In other words, the number

of conflict clusters can be written as $|T| - k$, where T is the set of all transitions and k is the number of free-choice sets. So, the rank condition translates to $\text{rank}(\boldsymbol{A}) = |T| - k - 1$. The same property is shown in [22] to be sufficient for the existence of a live and bounded marking in case of certain non-free-choice nets.

Our setting is quite different to the one considered in [22]. The class of nets is different, and so is the behavioral property under investigation. Nevertheless, the condition provided in our rank theorem looks pretty similar to the characterization mentioned above.

5.2 Proof of the Rank Theorem

Theorem 5 (Rank theorem) *For a Petri net N with $|T|$ transitions and incidence matrix $\boldsymbol{A}$, if there exists a schedule of N that involves k FCSs, then* $\text{rank}(\boldsymbol{A}) \leq |T| - k - 1$.

Proof: Let *sch* be a schedule of a Petri net $N = (P, T, F)$ for some marking M of N. Let $\mathbb{S}$ be the set of FCSs involved in *sch*. $|\mathbb{S}| = k$. First, we construct a Petri net $N' = (P', T, F')$ by adding places and edges to N. Then, we show that $\text{rank}(\boldsymbol{A'}) \leq |T| - 1$, where $\boldsymbol{A'}$ is the incidence matrix of N'. Finally, we show that $\text{rank}(\boldsymbol{A'}) = \text{rank}(\boldsymbol{A}) + k$. Therefore, $\text{rank}(\boldsymbol{A}) \leq |T| - k - 1$.

1. Construct a Petri net $N' = (P', T, F')$ by adding places and edges to N. As illustrated in Figure 5.1, for each $FCS_i = \{t_i, t_i'\} \in \mathbb{S}$, add 2 places $p_{ti}, p_{ti'}$ and 4 edges $F'(p_{ti}, t_i) = n_i, F'(t_i, p_{ti'}) = n_i, F'(p_{ti'}, t_i') = n_i', F'(t_i', p_{ti}) = n_i'$.

2. Determine n_i, n_i'. Since for each edge (u, v) in *sch* there exists a directed path from r to u and from v to r, there must exist a closed walk W in *sch* from r to r, such that

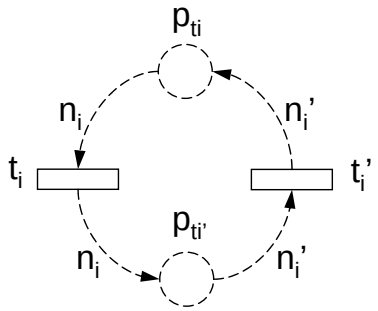


Figure 5.1: Illustration of the construction of N'

W traverses each edge in *sch* at least once. Since the length of any path in *sch* is bounded by $|E|$, the length of W is bounded by $2|E|^2$. Also note that E is nonempty. Thus, W is a walk of finite and non-zero length.

Let σ be the corresponding firing sequence of W. Clearly, σ is finite and nonempty. Let c_i, c_i' be the number of times of t_i, t_i' of FCS_i appears in the σ, respectively. Since W traverses each edge at least once, $c_i \geq 1, c_i' \geq 1$. Let c be a common multiple of $\{c_1, c_1', \ldots, c_k, c_k'\}$, and $n_i = c/c_i, n_i' = c/c_i'$.

3. Show that $\text{rank}(\boldsymbol{A}') \leq |T| - 1$. First, define M' of N' as the following.

$$M'(p) = \begin{cases} M(p) & \text{if } p \in P \cap P' \\ c & \text{otherwise} \end{cases}$$

It is easy to see that the definition of n_i, n_i' and M' ensures that σ can be fired at M' in N', and the number of tokens in added places $p_{ti}, p_{ti'}, i = 1, \ldots, k$ are unchanged after the firing sequence. That is, σ is a nonempty firing sequence of N' from M' to M'. It implies that $\boldsymbol{A'}^{\mathrm{T}}\bar{\sigma} = 0$, where $\bar{\sigma}$ is the firing count vector of σ. Since $\bar{\sigma}$ is not

a vector full of zeros, $\text{rank}(\boldsymbol{A'}^{\text{T}}) < |T|$. Thus, $\text{rank}(\boldsymbol{A'}) \leq |T| - 1$.

4. Compare the incidence matrix $\boldsymbol{A'}$ of N' and the incidence matrix $\boldsymbol{A}$ of N. As illustrated by the following graph, for each $FCS_i = \{t_i, t_i'\} \in \mathbb{S}$, N' contains two additional column vectors $\boldsymbol{y_i}, \boldsymbol{y_i'}$ corresponding to $p_{ti}, p_{ti'}$, respectively. Let $Y = [\boldsymbol{y_1} \dots \boldsymbol{y}_{|\mathbb{S}|}]$ and $Y' = [\boldsymbol{y_1'} \dots \boldsymbol{y}'_{|\mathbb{S}|}]$. Then $\boldsymbol{A'} = [\boldsymbol{A}|Y|Y']$.

	p_{ti}		$p_{ti'}$	
	$\ddots$	0	$\ddots$	0	$\ddots$
ti	$\dots$	$-n_i$	$\dots$	n_i	$\dots$
	$\ddots$	0	$\ddots$	0	$\ddots$
ti'	$\dots$	n_i'	$\dots$	$-n_i'$	$\dots$
	$\ddots$	0	$\ddots$	0	$\ddots$

In vector $\boldsymbol{y_i}$, there are exactly two non-zero entries $-n_i, n_i$, corresponding to transition t_i, t_i' respectively. Similarly, in vector $\boldsymbol{y_i}'$, there are entries $n_i', -n_i'$, corresponding to transition t_i, t_i' respectively. A' is the incidence matrix of N'

5. Show that $\text{rank}(\boldsymbol{A'}) = \text{rank}(\boldsymbol{A}) + |\mathbb{S}|$.

   Since $\boldsymbol{y_i} + \boldsymbol{y_i}' = 0$, $\boldsymbol{y_i}, \boldsymbol{y_i}'$ are linearly dependent. Thus, $\text{rank}(\boldsymbol{A'}) = \text{rank}([\boldsymbol{A}|Y])$.

   Then, we show that each column vector $\boldsymbol{y_i}$ of Y is linearly independent with respect to other column vectors in $[\boldsymbol{A}|Y]$. We proceed by contradiction. It is known that vector $\boldsymbol{y_i}$ is linearly dependent with respect to other columns vectors in $[\boldsymbol{A}|Y]$ if and

only if $\boldsymbol{y_i}$ is a linear combination of other vectors.

$$\boldsymbol{y_i} = \sum_{1 \le j \le |P|} a_j \boldsymbol{A}_j + \sum_{\substack{1 \le k \le |\mathbb{S}| \\ k \ne i}} b_k \boldsymbol{y_k}$$

where $\boldsymbol{A} = [\boldsymbol{A}_1 \dots \boldsymbol{A}_{|P|}]$, $a_j \in Q$, $b_k \in Q$. Let vector $\boldsymbol{a} = [a_1 \dots a_{|P|}]^{\mathrm{T}}$, and $\boldsymbol{z} = \boldsymbol{y_i} - (\sum_{\substack{1 \le k \le |\mathbb{S}| \\ k \ne i}} b_k \boldsymbol{y_k})$. That is, $\boldsymbol{z} = \boldsymbol{A}\boldsymbol{a}$. Then for each T-invariant $\boldsymbol{x}$ of N,

$$\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{a}^{\mathrm{T}}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{x} = 0 \Rightarrow (\boldsymbol{A}\boldsymbol{a})^{\mathrm{T}}\boldsymbol{x} = 0 \Rightarrow \boldsymbol{z}^{\mathrm{T}}\boldsymbol{x} = 0.$$

We then show that $\boldsymbol{z}^{\mathrm{T}}\boldsymbol{x} = 0$ implies the existence of a cyclic dependence relation. Clearly, the non-zero entries in $\boldsymbol{z}$ corresponds to transitions in FCSs, and for a FCS $\{t_a, t_b\}$, $z_a > 0$ if and only $z_b < 0$. Let $\mathbb{S}' \subseteq \mathbb{S}$ be the set of FCS that has non-zero entries in $\boldsymbol{z}$. Obviously, $FCS_i \in \mathbb{S}'$. Let $S = \{t_a | z_a > 0\}$, $S' = \{t_b | z_b < 0\}$. It is easy to see that S is a cover of $\mathbb{S}'$, and $S' = \mathbb{S}' \backslash S$. Since $\boldsymbol{z}^{\mathrm{T}}\boldsymbol{x} = 0$, for each transition $t_j \in S$, $x_j > 0$ implies that there exists at least one transition $t_k \in S'$ such that $x_k > 0$. That is, for each transition $t \in S, t \rightarrowtail S'$. Since $S' = \mathbb{S}' \backslash S$, $\mathbb{S}'$ is in cyclic dependence relation. It implies that the schedule *sch* involves a cyclic dependent FCS. This contradicts Theorem 2. Thus, each column vector $\boldsymbol{y_i}$ of Y is linearly independent with respect to other column vectors in $[\boldsymbol{A}|Y]$, where $Y = [\boldsymbol{y_1} \dots \boldsymbol{y}_{|\mathbb{S}|}]$. Therefore, $\mathrm{rank}(\boldsymbol{A'}) = \mathrm{rank}([\boldsymbol{A}|Y]) = \mathrm{rank}(\boldsymbol{A}) + |\mathbb{S}|$. □

Theorem 5 is equivalent to the statement: a schedule of a Petri net involves at most $(|T| - \mathrm{rank}(\boldsymbol{A}) - 1)$ FCSs. It implies that if $\mathrm{rank}(\boldsymbol{A}) > |T| - |\mathbb{S}| - 1$, where $\mathbb{S}$ is the set of all FCSs of a Petri net N, then for any marking M of N, there exists no schedule of (N, M) that involves all FCSs of the net. Thus, we could use the rank theorem to prove unschedulability of a Petri net.

Consider the Petri net shown in Figure 2.5. $|T| = 9, \text{rank}(\boldsymbol{A}) = 7, |\mathbb{S}| = 2$, thus no schedule involves all FCSs. For the Petri net shown in Figure 3.3, $|T| = 15, \text{rank}(\boldsymbol{A}) = 11, |\mathbb{S}| = 4$, thus no schedule involves all FCSs.

5.3 Comparison with Cyclic Dependence

Since computing the rank of a matrix has a polynomial-time complexity (Gaussian elimination), checking unschedulability by the rank of incidence matrix is more efficient than by a cyclic dependence, because the latter requires to iterate through potentially all possible covers of a set of FCSs. The number of covers of a set of FCSs is an exponential of the number FCSs.

However, the rank condition is weaker in establishing the unschedulability as is illustrated by the following example. Figure 5.2 shows a Petri net with $|T| = 15$, $\text{rank}(\boldsymbol{A}) = 12$, and $|\mathbb{S}| = 2$. Thus, we can not prove unschedulability by the rank theorem. Note that the left part of the Petri net is identical to the Petri net shown in Figure 2.5. There exists a cyclic dependence relation in FCSs $\{B, C\}$, $\{F, G\}$. Thus, we can prove unschedulablity by the cyclic dependence theorem.

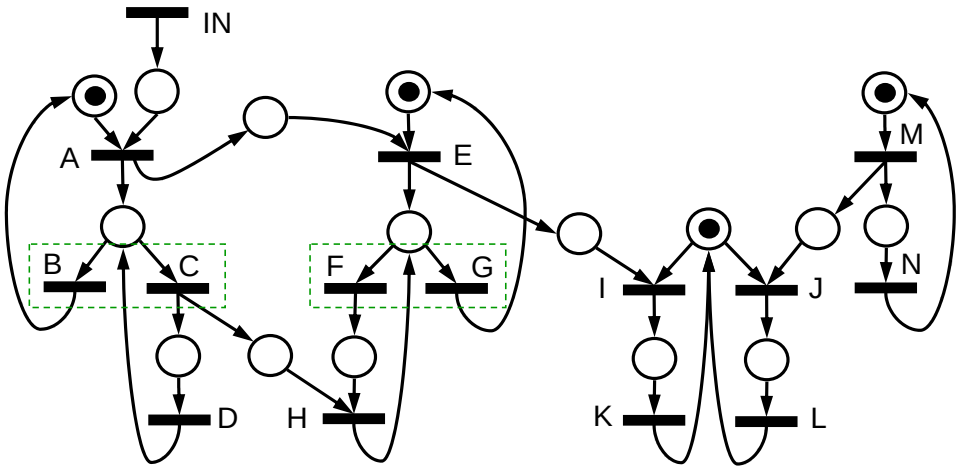


Figure 5.2: A Petri net whose unschedulability can be proved by Theorem 2, but not by Theorem 5.

Chapter 6

Experiments

In this Chapter, we show that the sufficient conditions introduced in the cyclic dependence theorem and the rank theorem hold for a wide class of real-life industrial applications. It means that the Petri nets generated from their system specifications are not schedulable, and our approach can be effectively applied to establish that. Note that to prove unschedulability of a Petri net based on the cyclic dependence theorem, we need to assert that each schedule involves at least one cyclic dependent FCS. To prove unschedulability based on the rank theorem, we need to assert that each schedule involves all FCSs of a Petri net. We use some public available JPEG and MPEG codecs as our test bench. The codecs used in our experiments are modelled as Kahn process networks.

6.1 MPEG-2 Decoder

We use an MPEG-2 decoder [51] developed by Philips Research Laboratories. The decoder is written in about 5,000 lines of YAPI [20] code, a system specification language

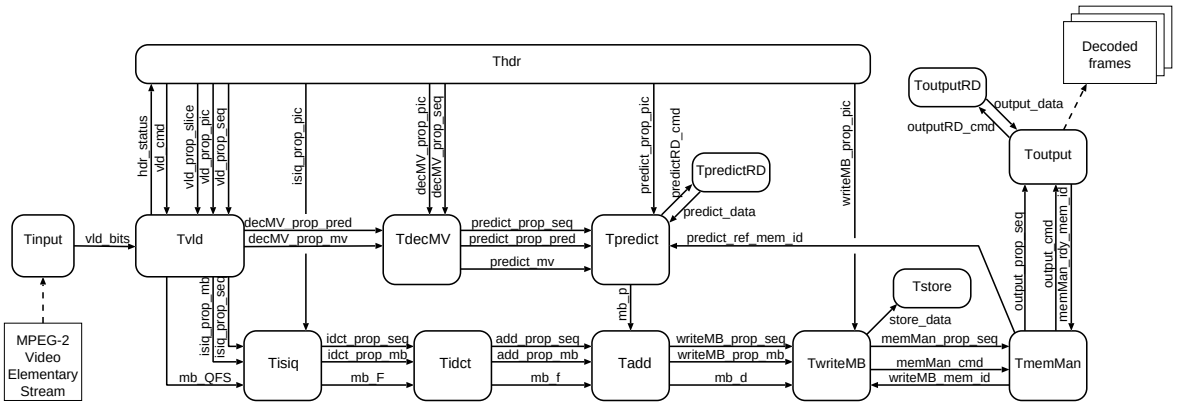


Figure 6.1: An MPEG-2 decoder modeled as a KPN.

based on C++. As shown in Figure 6.1, the system consists of 11 concurrent processes communicating through 45 channels. We perform schedulability analysis on 5 processes: *TdecMV*, *Tpredict*, *Tisiq*, *Tidct*, and *Tadd*. The first two processes implements the spatial compression decoding. The last three processes implements the temporal compression decoding and image generation. In total, the 5 processes have 10 channels, and 13 interfaces (communicating ports with other processes or the environment).

6.2 M-JPEG* Encoder

We use an M-JPEG* [38] encoder also developed by Philips. The source code is obtained through the Sesame [2] project public release. The encoder is written in about 2,000 lines of YAPI code. As shown in Figure 6.2, the system consists of 8 processes communicating through 18 channels. Video data are parsed by Process *DMUX* and are sent directly to Process *DCT* or via Process *RGB2YUV*, depending on the video format. Video parameters are send to Process *OB-Control*, which controls the video processing

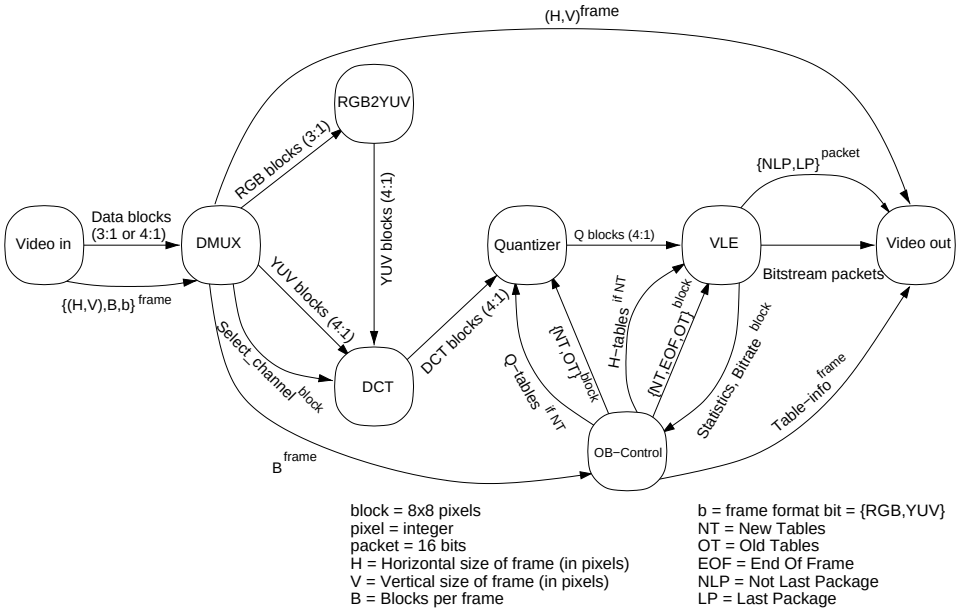


Figure 6.2: An M-JPEG* encoder modeled as a KPN.

in Process *DCT*, *VLE*, and Vudeo Out. It also collects statistics information to adjust Huffman coding tables and quantization tables. We perform schedulability analysis on the entire system.

6.3 XviD MPEG4 Encoder

Our model of a XviD MPEG4 encoder is based on the Kahn process network described in [45] and the C source code from [4]. XviD is a open source implementation of the ISO MPEG-4 video codec. The Kahn process network model was originally developed as a Sesame [2] application. As illustrated by Figure 6.3, the system consists of 15 processes with 40 channels. The input to the system is a video frame with parameters, which is modelled by a producer process *Video In*. The output is the encoded video data stream,

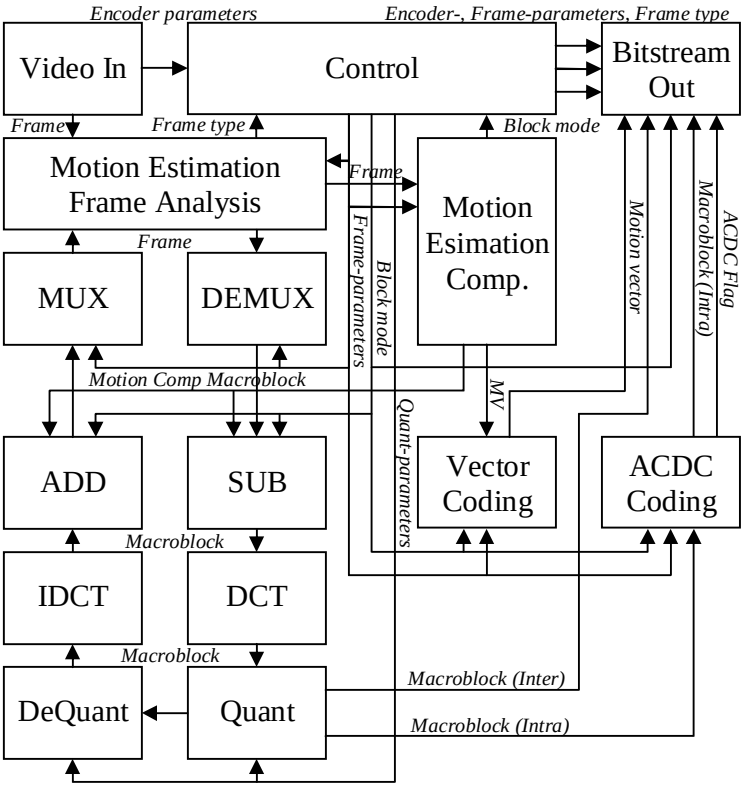


Figure 6.3: An XviD encoder modeled as a KPN.

which is modelled by a consumer process *Video Out*.

The encoder supports two types of frames: I-frame and P-frame. It performs motion estimation analysis to determine if an incoming frame will be treated as I-frame or P-frame. Consequently two types of frame will go through different processing paths. An I-frame will be split into macro-blocks and encoded independently. In a P-frame, a macro-block could be an intra-block, an inter-block, or an not-coded-block, depending the value of Sum of Absolute Differences (SAD). The granularity of tokens passing between processes is macroblock.

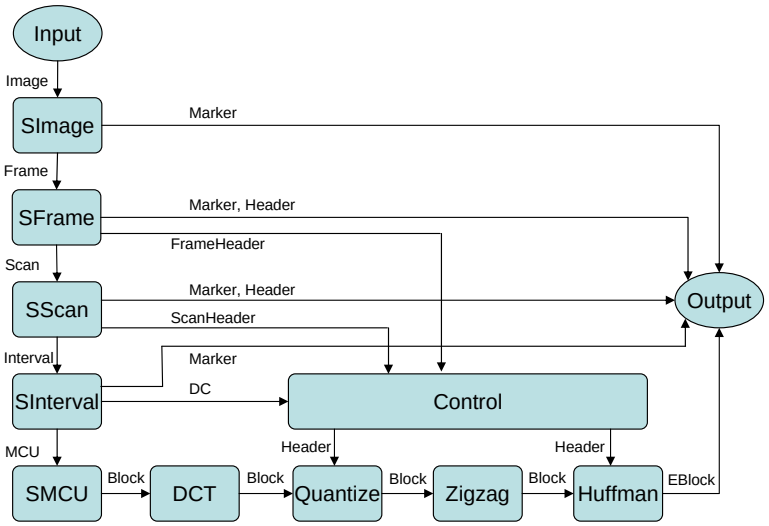


Figure 6.4: A baseline JPEG encoder as a Kahn process network.

6.4 PVRG JPEG Encoder

We obtain the Stanford Portable Video Research Group (PVRG) JPEG codec source code from [3]. We adjusted it based on JPEG baseline standard. Our model consists of 10 processes and 21 channels. As shown in Figure 6.4, the input image first goes through a set of decomposition processes. An image is first decomposed into frames, then each frame is decomposed into scans, then each scan is decomposed into intervals, and finally each interval is decomposed into Minimum Coded Units (MCU). The functional core part consists of 4 processes implementing Discrete Cosine Transform (DCT), quantization, Huffman coding, and control. The granularity of tokens passing between the four processes is 8×8 block. We preform schedulability analysis of the Petri nets generated from the model of the encoder and its functional core.

We also experiment different modelling decisions in this test case. For example,

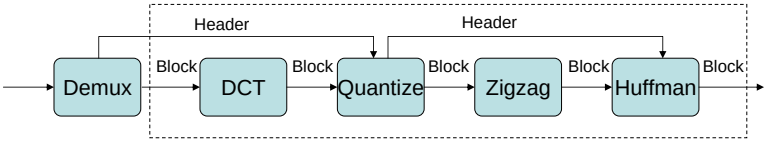


Figure 6.5: A simplified JPEG encoder as a Kahn process network using distributed control.

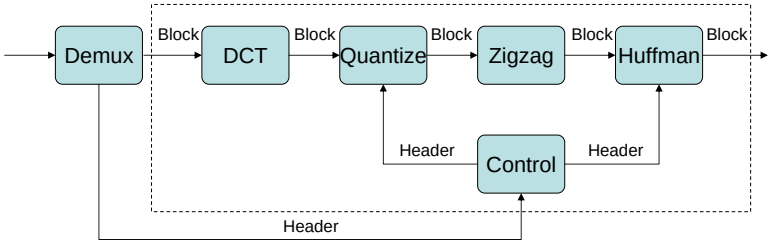


Figure 6.6: A simplified JPEG encoder as a Kahn process network using central control.

the control of synchronization between concurrent processes could be managed by a single master process, or distributed among processes. As shown in Figure 6.5 and Figure 6.6, Process Demux splits the input bit stream into two token streams: header stream and block stream. The header contain all the parameters and tables necessary to transform a block. It could be passed to a control process then sent to quantization and encoding processes. It could also be sent directly to quantization process, and then passed to encoding process.

6.5 Worst-Case Tests

Additionally, we create a set of instances to test the worst case performance of our analysis. They are listed in Figure 6.8. Each Petri net instance is a chain of free choices and contains no cyclic dependence. It is relatively easy to find a schedule for this kind of Petri nets. However, it is computationally expensive to prove there is no cyclic dependent

Instance	Place	Tran.	Arc	FCS	Rank	Schedulable	Runtime		
							Check Rank	Check Dependence	Scheduling
JPEGenc1	26	27	64	6	21	N	<0.01s	0.19s	523.75s
JPEGenc2	67	68	167	14	57	N	<0.01s	0.54s	>24hr
MJPEGenc	117	124	330	25	108	N	<0.01s	0.04s	>24hr
MPEG2dec1	116	144	358	38	111	N	<0.01s	0.25s	>24hr
MPEG2dec2	115	106	309	8	97	Y	<0.01s	17.28s	6.91s
MPEG4enc	72	72	184	15	63	N	<0.01s	0.16s	>24hr

Figure 6.7: Statistics of schedulability analysis of Petri net models of JPEG and MPEG codecs

Instance	Place	Tran.	Arc	FCS	Rank	Schedulable	Runtime		
							Check Rank	Check Dependence	Scheduling
choice3	6	9	20	3	5	Y	<0.01s	0.01s	<0.01s
choice4	7	11	25	4	6	Y	<0.01s	0.02s	0.01s
choice5	8	13	30	5	7	Y	<0.01s	0.05s	0.01s
choice6	9	15	35	6	8	Y	<0.01s	0.14s	0.04s
choice7	10	17	40	7	9	Y	<0.01s	0.45s	0.05s
choice8	11	19	45	8	10	Y	<0.01s	1.42s	0.09
choice9	12	21	50	9	11	Y	<0.01s	4.45s	0.29s
choice10	13	23	55	10	12	Y	<0.01s	14.06s	0.87s
choice11	14	25	60	11	13	Y	<0.01s	44.86s	1.06s
choice12	15	27	65	12	14	Y	<0.01s	141.55s	4.43s

Figure 6.8: Statistics of schedulability analysis of Petri nets in the worst-case test suite

FCSs. The analyzer has to check all subsets of FCSs and all covers of each subset.

6.6 Results and Analysis

We implemented our schedulability analyzer in C. All experiments were run on a 3.0 GHz Intel Pentium CPU with 512 MB memory. Since no other schedulability analyzer is available, we compare its performance with a scheduler [16]. The scheduler performs a schedulability analysis via heuristic construction of a schedule. Figure 6.7 summarizes the experiment results of Petri nets modelling JPEG MPEG codecs. Our analyzer typically proves a Petri net is not schedulable within a second, while the scheduler often fails to

terminate in 24 hours. Note that there are two instances modelling MPEG2 decoders, *MPEG2dec1* and *MPEG2dec2*. The former, generated from the original YAPI source code, is not schedulable. The second, generated from a modified source code, is schedulable. The modification removes the correlated control structures that result in cyclic dependence using the technique described in [7]. Note that our schedulability analyzer computes the minimum set of FCSs that has a cyclic dependence relation, once it proves a Petri net is unschedulable. The minimum set of FCSs provides useful information to find the correlated control structures.

Figure 6.8 shows that for schedulable Petri nets, the run time of checking cyclic dependence grows exponentially as the number of FCSs increases. However, checking rank remains efficient. In fact, checking rank takes less than 10 milliseconds for all of our experiments.

Our schedulability analyzer is effective because there exist certain program structures in the codecs. We illustrate it using a Huffman coding process and quantization process of a JPEG encoder. Figure 6.9 shows a simplified description of the processes. The process first reads from a control process a header which includes all parameters necessary to perform Huffman coding on a block. Then it iterates through all blocks of a component in a Minimum Coded Unit (MCU). The Huffman coding and reading from a zigzag process are performed at the block level inside the loop. Since JPEG standard requires samples of a component must use the same Huffman coding table, and multiple component samples could be interleaved within a compressed data stream, it needs to update the Huffman table from time to time. Also note that the loop has a variable number of iterations. The

vertical and horizontal sampling factors could be different for different components and only known at run-time. All the above requires communications inside and outside a loop structure. The quantization process has a similar program structure to the Huffman coding process, because samples of a component are required to use the same Quantization table and processing and communication data is performed at block level. The control process synchronizes the two concurrent processes such that a block is processed in the quantization process and later in a Huffman coding process with the set of parameters (e.g. vertical and horizontal sampling factors) of the same component.

Synchronized communications inside and outside a loop structure are common in the codecs. However, the controls of loops are abstracted as non-deterministic free choices in a Petri net. These two factors cause unschedulability that can be efficiently checked by our structural analysis. Note that this particular code structure is just a special case that result in unschedulability. Our analysis can be applied to general Petri net models.

```
PROCESS Huffman(
              In_DPORT Control_headerIn,
              In_DPORT Zigzag_blockIn,
              Out_DPORT Output)
{
  while(1) {
    READ_DATA(Control_headerIn, header, 1);
    Vi = getVSF(header);
    Hi = getHSF(header);
    Htable = getHtable(header);
    for(v=0; v<Vi; v++) {
      for(h=0; h<Hi; h++) {
        READ_DATA(Zigzag_blockIn, block, 1);
        block = HuffmanEncoding(block, Htable);
        WRITE_DATA(Output, block, 1);
} } } }

PROCESS Quantization(
              In_DPORT Control_headerIn,
              In_DPORT DCT_blockIn,
              Out_DPORT Zigzag_blockOut)
{
  while(1) {
    READ_DATA(Control_headerIn, header, 1);
    Vi = getVSF(header);
    Hi = getHSF(header);
    Qtable = getQtable(header);
    for(v=0; v<Vi; v++) {
      for(h=0; h<Hi; h++) {
        READ_DATA(DCT_blockIn, block, 1);
        block = Quantization(block, Qtable);
        WRITE_DATA(Zigzag_blockOut, block, 1);
} } } }
```

Figure 6.9: A simplified Huffman coding process and a simplified quantization process.

Chapter 7

Conclusion and Future Work

We formulated the compile-time schedulability problem of concurrent programs as the schedulability problem of a general Petri net. We formally defined the schedulability of Petri nets and provided several sufficient conditions for checking unschedulability using linear algebra and linear programming techniques. Our experimental results indicate that these techniques effectively and efficiently detect unschedulability of Petri nets for practical examples. They help to identify the correlated control structures in original programs that cause unschedulability.

Our work provides a sound foundation for further investigation of compile-time schedulability based on Petri nets. We list below some interesting research directions.

7.1 Dependence Considering Firability

We defined a dependence relation between transitions statically based on the existence of T-invariants. However, not every T-invariant may be firable in a Petri net. If the

dependency relation is violated because of non-firable T-invariants, our approach would fail to establish unschedulability. Taking into account firability of T-invariants would increase the resolution power of the proposed method.

7.2 Petri Net Subclasses

The cyclic dependence theorem and the rank theorem provide only sufficient conditions for unschedulability. One possible research direction is to look for subclasses of Petri nets (e.g. free choice Petri net) for which these conditions are also necessary, or weaker sufficient conditions can be proved. Petri net models of concurrent programs in many applications are free-choice Petri nets. Due to the existing rich free-choice Petri net theory, the research direction is particularly promising.

7.3 Cyclic Dependence Theorem and Rank Theorem

Though we treat the cyclic dependence theorem and the rank theorem as two independent approaches to schedulability analysis, we believe that they have connections. It would be interesting if one could establish a direct relation between the rank of the incidence matrix and the existence of cyclic dependence relation.

7.4 Folding Infinite Schedule

Our results can be used to represent the unbounded parts of the schedule in an implicit but finite form. Hence, we believe an implementation could be derived that guarantees boundedness as long as a system functions in a "good" part of the schedule and raising

the flag when it enters a potentially unbounded part. This approach would significantly extend the applicability of quasi-static scheduling for practical applications.

Bibliography

[1] The ptolemy project. URL: `http://ptolemy.eecs.berkeley.edu`.

[2] Sesame project public release. URL: `http://sesamesim.sourceforge.net`.

[3] Stanford PVRG JPEG codec. URL: `http://www.dclunie.com/jpegge.html`.

[4] XviD MPEG-4 video codec. URL: `http://www.xvid.org`.

[5] Spw 5-xp - the dsp workbench. In *CoWare, Inc*, 2004.

[6] M. D. Anna and S. Trigila. Concurrent system analysis using petri nets: an optimized algorithm for finding net invariants. *Comput. Commun.*, 11(4):215–220, 1988.

[7] G. Arrigoni, L. Duchini, L. Lavagno, C. Passerone, and Y. Watanabe. False path elimination in quasi-static scheduling. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2002.

[8] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[9] Shamik Bandyopadhyay. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Master's thesis, University of California, Berkeley, 2005.

[10] J. T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California, Berkeley, 1993.

[11] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer valued control streams. In *Proceedings of the 28th Asilomar conference on Signals, Systems, and Computer*, October 1994.

[12] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[13] Y. Chen, W. T. Tsai, and D. Chao. Dependency analysis-a petri-net-based technique for synthesizing large concurrent systems. *IEEE Trans. Parallel Distrib. Syst.*, 4(4):414–426, 1993.

[14] **Citation removed to allow blind review**.

[15] J. Cortadella, A. Kondratyev, L. Lavagno, M. Massot, S. Moral, C. Passerone, Y. Watanabe, and A. Sangiovanni-Vincentelli. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. In *Proceedings of the 37th Design Automation Conference*, June 2000.

[16] J. Cortadella, A. Kondratyev, L. Lavagno, C. Passerone, and Y. Watanabe. Quasi-

static scheduling of independent tasks for reactive systems. *IEEE Transactions on Computer-Aided Design*, 2004.

[17] J. Cortadella, A. Kondratyev, L. Lavagno, and A. Taubin Y. Watanabe. Quasi-static scheduling for concurrent architectures. *Fundamenta Informaticae*, 62, 2004.

[18] Jordi Cortadella, Alex Kondratyev, Luciano Lavagno, C. Passerone, and Y. Watanabe. Quasi-static scheduling of independent tasks for reactive systems. In *Proceedings of the 23rd International Conference on Applications and Theory of Petri Nets*, pages 80–100, 2002.

[19] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages*, Los Angeles, January 1977.

[20] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. YAPI: Application Modeling for Signal Processing Systems. In *Proceedings of the* 37^{th} *Design Automation Conference*, June 2000.

[21] Joerg Desel. Private communication, August 2004.

[22] Jorg Desel and Javier Esparza. *Free choice Petri nets*, volume 40 of *Cambridge Tracts In Theoretical Computer Science*. Cambridge University Press, New York, NY, USA, 1995.

[23] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3):366–390, March 1997.

[24] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Petri Nets*, pages 374–428, 1996.

[25] R. Lauwereins G. Bilsen, M. Engels and J. A. Peperstraete. Cyclostatic dataflow. *IEEE Trans. Signal Processing*, 44:397–408, 1996.

[26] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(6):742–760, June 1999.

[27] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, 1993.

[28] N. Halbwachs, Y.-E. Proy, and P. Raymond. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *International Symposium on Static Analysis, SAS'94*, 1994.

[29] Philp Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935.

[30] D. Har'el, H. Lachover, A. Naamad, A. Pnueli, et al. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[31] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[32] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475, Aug 1974.

[33] G. Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *Information processing*, pages 993–998, Aug 1977.

[34] A. Kalavade and E.A. Lee. A hardware-software codesign methodology for DSP applications. *IEEE Design and Test of Computers*, 10(3):16–28, September 1993.

[35] H. Kopetz and G. Grunsteidl. TTP – A protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1), January 1994.

[36] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow graphs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[37] Y.T.S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, June 1995.

[38] Paul Lieverse, Todor Stefanov, Pieter van der Wolf, and Ed Deprettere. System level design with spade: an m-jpeg case study. In *Proceedings of IEEE/ACM International Conference on Computer Aided Design*, pages 31–88, Nov 2001.

[39] J. Martinez and M. Silva. A simple and fast algorithm to obtain all invariants of a generalized petri net. In *Informatik-Fachberichte 52: Application and Theory of Petri Nets*, pages 301–310. Springer-Verlag, 1982.

[40] H. Mathony. Universal Logic Design Algorithm and its Application to the Synthesis of Two-level Switching Circuits. *IEE Proceedings*, 136 Pt. E(3), May 1989.

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

[42] T. Murata. Petri Nets: Properties, analysis and applications. *Proceedings of the IEEE*, pages 541–580, April 1989.

[43] T. M. Parks. *Bounded scheduling of process networks.* PhD thesis, University of California, Berkeley, 1995.

[44] C. Passerone, Y. Watanabe, and L. Lavagno. Generation of minimal size code for schedule graphs. In *Proceedings of the Design Automation and Test in Europe Conference*, March 2001.

[45] Andy D. Pimentel. Modeling XviD as a Kahn process network, a Sesame application design document. URL: `http://staff.science.uva.nl/~andy/apps/xvid.pdf`.

[46] Marco Sgroi, Luciano Lavagno, Yosinori Watanabe, and Alberto Sangiovanni-Vincentelli. Synthesis of embedded software using free-choice petri nets. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 805–810, New York, NY, USA, 1999. ACM Press.

[47] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System Design using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of the Design Automation and Test in Europe Conference*, February 2004.

[48] K. Strehl, L. Thiele, D. Ziegenbein, R. Ernst, and et al. Scheduling hardware/software systems using symbolic techniques. In *International Workshop on Hardware/Software Codesign*, 1999.

[49] Karsten Strehl, Lothar Thiele, Dirk Ziegenbein, Rolf Ernst, and Jürgen Teich. Scheduling hardware/software systems using symbolic techniques. In *Proceedings of the seventh international workshop on Hardware/software codesign*, pages 173–177, 1999.

[50] F. Thoen, M. Cornero, G. Goossens, and H De Man. Real-time multi-tasking in software synthesis for information processing systems. In *Proceedings of the International System Synthesis Symposium*, 1995.

[51] P. van der Wolf, P. Lieverse, M. Goel, D.L. Hei, and K. Vissers. An MPEG-2 Decoder Case Study as a Driver for a System Level Design Methodology. In *Proceedings of the* 7^{th} *International Workshop on Hardware/Software Codesign*, May 1999.

[52] P. Wauters, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-dynamic dataflow. In *Proceedings of the 4th EUROMICRO Workshop on Parallel and Distributed Processing*, January 1996.

Appendix A

Another Proof of Proposition 1

Given a Petri net N with two FCSs $S_1 = \{t_1, t_2\}$, $S_2 = \{t_3, t_4\}$, if $t_1 \rightarrowtail t_3$ and $t_4 \rightarrowtail t_2$, then for any marking M of N, there exists no schedule of (N, M) involving S_1 or S_2.

Proof: We show that the unfolding $G'(V', E', r')$ of a valid schedule $G(V, E, r)$ with S_1 or S_2 involved is infinite, which violates finiteness of a schedule.

The proof proceeds by showing the validity of at least one of the two statements:

I1: G' contains an infinite path $r' \rightsquigarrow v'_1 \xrightarrow{t_1} y'_1 \rightsquigarrow v'_2 \xrightarrow{t_1} y'_2 \cdots$, such that the firing sequence corresponding to the path from r to v_k $(k = 1, 2, \dots)$ does not contain transition t_3.

I2: G' contains an infinite path $r' \rightsquigarrow u'_1 \xrightarrow{t4} z'_1 \rightsquigarrow u'_2 \xrightarrow{t4} z'_2 \cdots$, such that the firing sequence corresponding to the path from r to u_k $(k = 1, 2, \dots)$ does not contain transition t_2.

In G' let us choose vertex v' in which transitions from S_1 or S_2 are enabled and v'

to be the closest vertex to the root r' with this property. This vertex exists because S_1 or S_2 is involved in a schedule. Without loss of generality we may assume that S_1 is enabled in v'. Then $v'_1 = v'$ in proving I1.

From Property 5 of a schedule, follows that there exists $w', v' \in V'$ such that the path from v' to w' contains t_1 and $\mu(w') = \mu(v')$. This path corresponds to a firing sequence σ that makes a cycle from marking $\mu(v')$ back to itself and hence $\bar{\sigma}$ is a T-invariant. $t_1 \rightarrowtail t_3$ implies $t_3 \in \sigma$. Therefore, σ contains a vertex u' such that $u' \xrightarrow{t_3}$. Let u' be the closest descendant of v' with t_3 enabled.

Let us consider path $\sigma_1 \subset \sigma$ that goes from v' to u'. Two cases are possible.

Case 1. If $t_2 \in \sigma_1$ then σ_1 goes through vertex v'_2 with enabled t_2. t_1 and t_2 are from the same FCS and hence t_1 is also enabled in v'_2. Clearly the path from r' to v'_2 does not contain t_3 and therefore v'_2 satisfies the conditions of I1 and is a descendant of v'_1. Repeating the consideration for v'_2 one can conclude about the existence of infinite path satisfying I1.

Case 2. Suppose that $t_2 \notin \sigma_1$. Then the path from r' to u' does not contain transition t_2. In addition t_4 is enabled in u' (being in the same FCS as t_3) and therefore one can use u' as u'_1 in proving I2.

Bearing in mind that $t_4 \rightarrowtail t_2$ and applying the same arguments for closing the cycle from u'_1 one can conclude that there must exist a path δ from u'_1 to v'_2 in which t_2 is enabled. If δ does not contain t_3 then v'_2 satisfies the conditions of I1, which is the basis for constructing an infinite path. If δ contains t_3 then by choosing the first firing of t_3 in δ, one can obtain a vertex u'_2 in which t_3 is enabled together with t_4, and u'_2 is a descendants

of u_1'. This proves I2. $\square$