用於高階合成之派翠網路式系統階層驗證技術

Petri-Net based System-level Verification Techniques on High-level Synthesis

研 究 生：江宗錫

指導教授：董蘭榮　教授

中 華 民 國 九 十 六 年 七 月

National Chiao Tung University

Hsinchu City, Taiwan

Petri-Net based System-level Verification Techniques on High-level Synthesis

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy

in

Electrical and Control Engineering

by

Tsung-Hsi Chiang

July, 2007

To my father and mother …

~ among so many other things ~

Vita

1976	Born, Taichung City, Taiwan, ROC.
1994–1998	B.S. (Electrical Engineering), I-Shou University, Kaohsiung County, Taiwan.
1998–2000	M.S. (Computer Science and Engineering), Yuan-Ze University, Taoyuan County, Taiwan.
2000–2007	Ph.D. (Electrical and Control Engineering), National Chiao Tung University, Hsinchu City, Taiwan.

推薦函

一、事由：推薦電機與控制工程系博士班研究生江宗錫提出論文以參加國立交通大學博士論文口試。

二、說明：本校電機與控制工程系博士班研究生江宗錫已完成博士班規定之學科及論文研究訓練。

有關學科部分，江君以修必應修學分（請查學籍資料），通過資格考試；有關論文方面，江君已完成“用於高階合成之派翠網路式系統階層驗證技術”初稿。

三、總言之，江君已具備國立交通大學電機與控制工程系博士班研究生應有之教育及訓練水準，因此推薦江君參加國立交通大學電機與控制工程系博士論文口試。

國立交通大學電機與控制工程學系

電機與控制工程學系副教授

董　蘭　榮

[illegible]

中華民國 96 年 6 月

國立交通大學
論文口試委員會審定書

本校電機與控制工程學系博士班江宗錫君

所提論文：用於高階合成之派翠網路式系統階層驗證技術

合於博士資格水準，業經本委員會評審認可。

口試委員：[illegible] [illegible]

[illegible] [illegible]

[illegible]

指導教授：[illegible]

系 主 任：[illegible]

中華民國 96 年 7 月 9 日

Department of Electrical and Control Engineering
National Chiao Tung University
Hsinchu, Taiwan R.O.C.

Date: July 9, 2007

We have carefully read the dissertation entitled Petri-Net based System-level Verification Techniques on High-level Synthesis submitted by Tsung-Hsi Chiang in partial fulfillment of the requirements of the degree of **DOCTOR OF PHILOSOPHY** and recommend its acceptance.

[illegible] 陳慶瀚

[illegible] [illegible]

[illegible]

Thesis Advisor: [illegible]

Chairman: chiou, jin-chern

用於高階合成之派翠網路式
系統階層驗證技術

學生：江宗錫　　　　　　　　　　　　指導教授：董蘭榮 教授

國立交通大學電機與控制工程研究所 博士班

摘要

本論文提出創新的系統階層驗證(system-level verification)技術，驗證的目標是高階合成(high-level synthesis)的設計結果，驗證的範圍是針對資料流系統(dataflow)或數位信號處理導向(DSP-driven)的演算法，驗證的理論是基於派翠網路(Petri net)圖型理論與技術。系統階層驗證的意義，主要在於驗證系統架構(architecture)的正確性，其目的與在 RTL 階層所做的功能性(functionality)驗證是不同的。通常，資料流或數位信號處理導向的演算法需要利用系統階層之演算法轉換技術(arithmetic transformation)作系統架構的最佳化設計，目的是為了使受設系統能夠符合系統規範，或藉由改善系統架構，來達成效能最佳化；另一方面，將演算法實現到硬體架構之前，所有運算工作也必須明確的被排程或將工作指派給某些運算單元依序執行。然而，系統階層的演算法轉換技術與運算元排程需要仰賴複雜的演算法來達成，而這樣的設計流程是非常容易出錯的。為了能夠偵測系統階層設計時候的設計錯誤，在系統階層的設計流程中，我們引入了系統階層驗證技術來驗證高階合成結果的正確性。不同於傳統複雜邏輯推論的驗證方法，我們提出以 Petri net

圖型理論與技術作為基礎，用來偵測資料流系統在高階合成階段可能發生的設計錯誤。首先，我們會利用所提出的圖型轉換法則，將原始的資料流圖型轉換成 Petri net 圖型。然後，以我們所提出的兩個系統階層驗證架構，使受測系統能在 Petri net 圖型領域中進行系統階層的驗證工作。與傳統的驗證技術比較起來，我們將驗證的工作從 RTL 階層提升到系統階層，因此，在系統階層設計初期，就能偵測可能產生的設計錯誤。越早在設計初期偵測設計的錯誤，就越能夠減少重新設計的成本並且縮短產品上市的時間。利用本論文所提出的驗證方式，我們也實現了幾種不同的系統階層的驗證演算法用來驗證高階合成的設計結果，並從實驗數據中，歸納出最佳的驗證演算法。

Petri-Net based System-level Verification Techniques on High-level Synthesis

Student: Tsung-Hsi Chiang Advisors: Dr. Lan-Rong Dung

Department of Electrical and Control Engineering

National Chiao Tung University

ABSTRACT

This thesis presents a novel verification technique for detecting design faults of high-level synthesis (HLS) results of the dataflow or DSP-driven algorithms by using Petri net (PN) theory. System-level or high-level verification means architecture verification, which is different from functional verification in register transfer level (RTL). Generally, dataflow or DSP-driven algorithms need algorithmic transformations to achieve optimal goals and also need scheduling algorithms to allocate processor resources for all execution tasks before mapping on a silicon. However, the optimization procedures of the algorithmic transformations and the design scheduling in HLS are error-prone and complex. In order to detect high-level faults, PN based high-level verification flow is introduced and applied in HLS. Instead of applying Boolean algebra in conventional verification, the PN based verifier adopts mathematical matrix manipulation of linear algebra. In the proposed verification flow, we first convert a dataflow or DSP-driven design in the form of FSFG graph into PN domain by using proposed graph based conversion method. Then, two PN based verifier frameworks are proposed to verify the correctness of the HLS results. While comparing to the conventional verification techniques, we put the verification work beyond the RTL or gate level, thus high-level design faults can be detected in the early stage of the design flow. Early fault detection in system-level design flow is helpful for reducing redesigning cost and time-to-market cycle time. Base on the proposed method, we also implement several algorithms of HLS verifiers. From the experimental results of the verification algorithms, we conclude the best approach algorithm of the verifier.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Significance of the Research

The growing advances in semiconductor technique and the great complexity of DSP-driven hardware design along with stringent time-to-market requirements pose serious challenges for verification to ensure that designs are relatively correct. A complete verification is not only the important issue for less redesigning costs and shorter tape out cycle time, it is a prerequisite for successful system design. Therefore, there is up to 80% of the overall design costs are due to verification efforts for a modern circuit. In industrial, practical simulation and test still remain the mainstream for verification issues. The scalability of these methods is easily applicable at any level from abstraction to physical of the design flow. Under a stand-alone environment, practical methods can detect simple bugs easily and quickly. However, simulation and test can provide the presence of bugs rather than the absence and face low error coverage problem. Unlike practical method, formal verification technique, which gain confidence through mathematical reasoning, has been getting much attention for its ensuring 100% design error coverage [1]. By using mathematical model, formal method conducts exhaustive exploration of all possible behaviors of design and proves or disproves the correctness of design intention underlying system specification or property.

Various formal verification methods have matured over the last decades to put the state of art [2–5]. In lectures [6,7], the authors give excellent survey of formal verification techniques, and the major trends can be classified into two categories of *equivalence checking* [7–9] and *model checking* [10, 11]. Equivalence checking technique is used to prove functional equivalence of two design representations modeled at the same or different levels of abstraction. While model checking is an automatic technique for checking a design model that is satisfied for a given specification covers all the properties that the system should hold. The theory and algorithms of both techniques have become available for today's commercial EDA (Electronic Design Automation) tools. However, these approaches tend to verify designs at RTL (Register Transfer Level) or gate level rather than high level.

This thesis concerns formal verification of detecting high-level faults for high-level synthesis (HLS) of DSP-driven or dataflow design. High-level design also means architecture design which is crucial to the success of modern hardware. In order to map DSP algorithms on a circuit, DSP algorithms may require arithmetic transformation to improve performance to meet specification under architecture constraints, such as silicon area, throughput rate and power consumption. Then, scheduling, allocation and assignment techniques may proceed iteratively to find optimal or suboptimal tasks schedules in HLS. The HLS design procedures usually require complex recursive processes and evaluations. Even without low level techniques, HLS still dominates the performance of a system. However, one may suffer from several design faults at high level in that. For instance, designer may make incomplete description on Integer Linear Programming (ILP) to apply retiming or scheduling technique and cause serious design faults. To unveil high-level faults, high-level verification must be taken into consideration in HLS design flow.

The proposed high-level verification is based on Petri Net (PN) theory [12–18]. The PN based verifier can be considered as a model checker. The inputs to the verifier include a PN model and a set of design properties. Given a DUV (Design-Under-Verification) of Dataflow Graph (DFG) or Fully-Specified Signal Flow Graph (FSFG) of a design, the PN model is obtained by using proposed conversion method that converts a FSFG graph of a design to PN domain. Each converted PN model is expressed by a PN characteristic matrix, which is sufficient to represent the characteristic of original design. The design specifications or properties to the verifier can be derived from the relationships of each node pair of PN graph and expressed by matrix equations. In PN domain, each HLS design procedures of the DSP algorithms can be seen a series PN token movements of a PN model. The HLS result is valid if and only if all the PN matrix equations are satisfied.

Based on such scenario, novel verification framework based on PN theory is presented in this thesis. Before introducing the proposed high-level verification method, some review works are made in Section 1.2.

1.2 Review of Previous Works

While surveying the related works, most existential verification methods usually utilize classical techniques, such as BDD (Binary Decision Diagrams) [19–24], Boolean SAT (Satisfiability) solver [25–27], symbolic model checking [28–34] and theorem proving [35, 36]. These techniques are extremely powerful but must be applied in RTL level. In order to verify the correctness of HLS result, most literatures focused on developing a strategy for RTL validation between the synthesized RTL result and its abstract level description. In [37,38], the verification task is partitioning into two subtasks, verifying the validity of register sharing and checking the correctness of RTL for the interconnection and control signals. Similarly in [39–42], a high-level design is decomposed into the control part and the datapath part and modeled by using FSMD (Finite State Machine with Data Path) [43]. These verification methods basically decompose and divide a high-level scheduled design into control and datapath. Therefore, the existential equivalent checking [7, 44] can be applied to check the correctness of datapath, and the model checking [10] can be used to verify the correctness of the control part.

Unlike traditional approaches, the proposed verification method is based on PN theory. Instead of using Boolean algebra, PN based method checks the correctness of HLS result by using mathematical matrix equations. When comparing with the existential methods, the proposed verification is to check the validity of high-level design in abstract level rather in synthesized RTL level. Therefore, high-level design faults that violate the non-preemption, job completion and precedence properties can be found, and reworks may be performed in the early stage of the design flow.

1.3 Contributions of this research

We list several contributions of this research in following items.

- *The proposed verification is capable of detect high-level faults in system-level rather than in RTL or gate level.*
  Conventionally, high-level faults may not be detected before the RTL or gate level results being generated. In this research, we put the verification tasks beyond the low level to the higher abstraction level. Since, early fault detection in system level design flow may take benefits from reducing re-

designing cost and also speeding up the time-to-market cycle time especially for the consumer productions.

- *The verification flow starts from PN modeling by using proposed conversion method, then the HLS results are verified in PN domain.*
  Unlike applying conventional Boolean algebra, this research presents a novel PN based method to verify the DSP-driven algorithms or dataflow designs. A given original dataflow graph is first modeled and converted into a PN graph. In PN domain, the HLS procedures represents a series PN token-movements in PN domain. Thus, we can detect design errors of HLS result by checking the validity of the PN token-movements [16].

- *The proposed verification frameworks can be used to verify both static architecture and dynamical behavior of the dataflow design.*
  PN theory provides a different domain viewpoint and analytical method to verify the design system. The proposed PN based verification method concerns both static architecture and dynamical behavior of the dataflow design. In static verification, PN based verifier is used to check the correctness of the high-level arithmetic transformations, such as retiming and unfolding techniques. In dynamical verification, PN based verifier checks the correctness of a dataflow schedule by checking the validity of PN tokens firing rules [12–15, 18].

- *The performance of the proposed PN based verifier is efficient.*
  In conventional approaches, a high-level design is first synthesized in RTL or gate level, then the existential Boolean based tools can be used in verification flow. Instead of using Boolean algebra, PN based verifier gains confidence in mathematical equations derived from PN characteristic matrix and verifies the correctness of the design in higher level. By using

mathematical linear algebra, high-level faults of the HLS results can be detected in PN based verification method. While comparing to the conventional approaches, PN based verifier does not need complex Boolean manipulations. And hence, PN based verifier is more efficient than traditional approaches [12–15, 18].

- *The proposed hybrid verification frameworks can be integrated into existential verification tools.*
  The concept of the proposed hybrid verification framework is to verify HLS arithmetic transformations in the first stage and check the correctness of HLS scheduling in the second stage. Such hybrid framework can be integrated into existential verification tools, such as the SMV model checker [14, 15].

1.4 Outline

This thesis is organized as following. In Chapter. 2, some background and review of the related researches are given. In order to verify dataflow design in PN domain, the dataflow modeling and conversion methods are introduced in Chapter. 3. In Chapter. 4, the hybrid verification framework combines PN theory and SMV-model checker is proposed. The PN-based two-folded verification technique for the dataflow graph is addressed in Chapter. 5. At last, some conclusions are made in Chapter. 6.

CHAPTER 2

Background Review

2.1 Levels of Abstraction

The terminology of *design synthesis* is defined by [45] as a translation process from a behavioral description into a structural description. Usually, the tripartite representation of Y-chart [46,47] is used to represent different types of design synthesis. As showing in Figure. 2.1, the Y-chart has three axes including behavioral, structural and physical domains in it. Along each axis, as we move farther away from the center of the Y-chart, the level of description becomes more abstract. Each circle intersects the Y axis at a particular level of representation within a domain. The outer circle is the system, the next is the register-transfer (RTL) level, followed by the logic and circuit levels. Table. 2.1 lists these levels and design objects on different abstraction levels.

To distinguish different tasks of synthesis works, we bravely describe the synthesis subtasks as follows [48]. As showing in Figure. 2.1, the most outside arc is the *System synthesis* or *high-level synthesis* (HLS). HLS consists of generating an optimized structural view of an architectural level model of a design. The process corresponds to determining resources assignment or interconnection of modules as well of executing schedules. On *register-transfer synthesis*, time is divided into intervals called control states or steps. Designers use a register-transfer description, which specifies for each control state the condition to be tested, all register

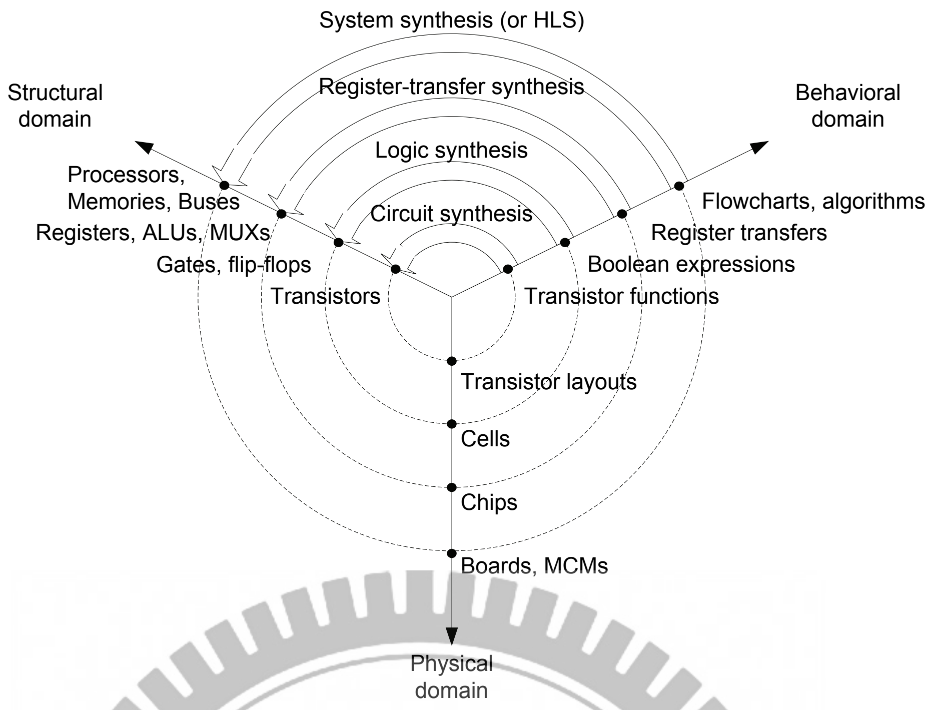


Figure 2.1: The Y-chart

Table 2.1: Design objects on different abstraction levels [45]

Level name	Behavioral domain	Structural domain	Physical domain
System level	Spec. charts Flowcharts Algorithms	Processors Controllers Memories Bues	Cabinets Boards MCMs Chips
RTL level	Register transfers	ALUs Multipliers MUXs Registers Memories	Chips Floorplans Module floorplans
Logic level	Boolean equations Sequencers	Gates Flip-flops	Modules Cells
Circuit level	Transfer functions	Transistors Connections	Transistor layouts Wire segments Contacts

transfers to be executed, and the next control state to be entered. *Logic synthesis* is the task of generating a structural view of a logic-level model. Logic synthesis determines the gate-level structure of a circuit, and transforming a logic model into an interconnection of instances of library cells. *Circuit synthesis* consists of creating a physical view of the circuit level. It entails the specification of all geometric patterns defining the physical layout of the chip.

In this thesis, we only concern the verification on system-level synthesis of dataflow algorithms. Several major HLS optimization techniques are addressed in the next section.

2.2 HLS: High-Level Synthesis

As mentioned in previous section, the goal of system-level synthesis or HLS is to generate optimized structural architecture of design. The typical design flow of a high-level DSP synthesis system is illustrated in Figure. 2.2 [49]. The first step in HLS is to convert the behavioral description into a graph-based internal representation in the form of a FSFG (Fully-Specified Flow Graph) or DFG (Dataflow Graph). The intermedia tasks in HLS include high-level optimization, scheduling and resource allocation, module binding, and control generation. The final architecture produced by HLS is typically at the RTL and can be represented by a netlist that consists of a network of functional units, registers, multiplexers, and buses. Several major optimization HLS techniques are utilized in system-level synthesis and are introduced in the following subsections.

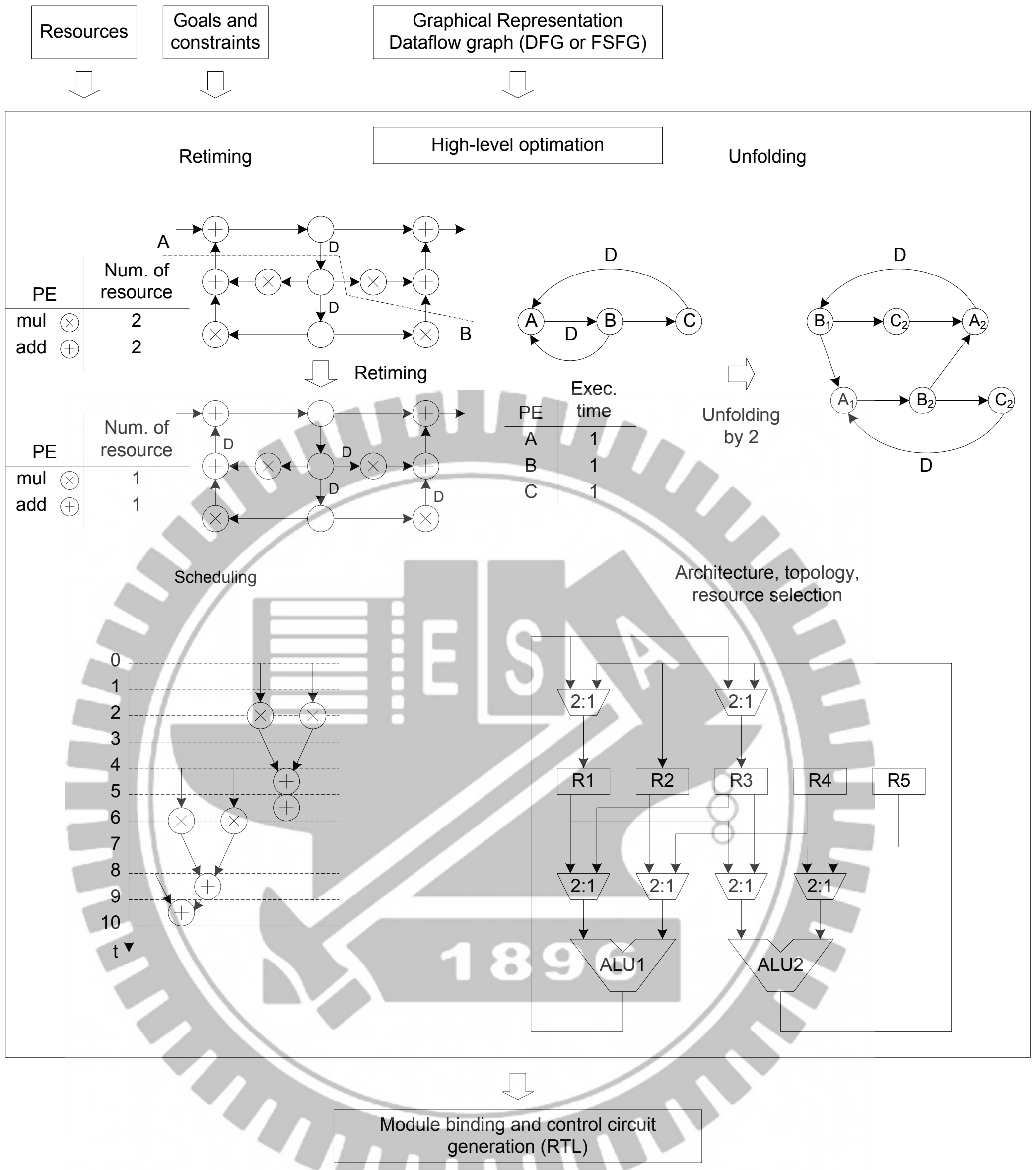


Figure 2.2: The high-level DSP synthesis system

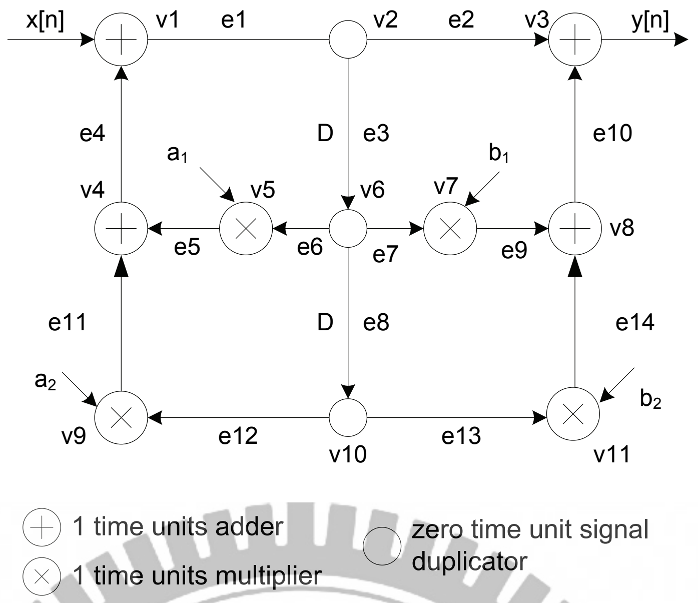
2.2.1 FSFG: Fully-Specified Signal Flow Graph

Before addressing the HLS optimization techniques, we first state the graph representation for dataflow design and several measurement matrices for evaluating performance of dataflow graph.

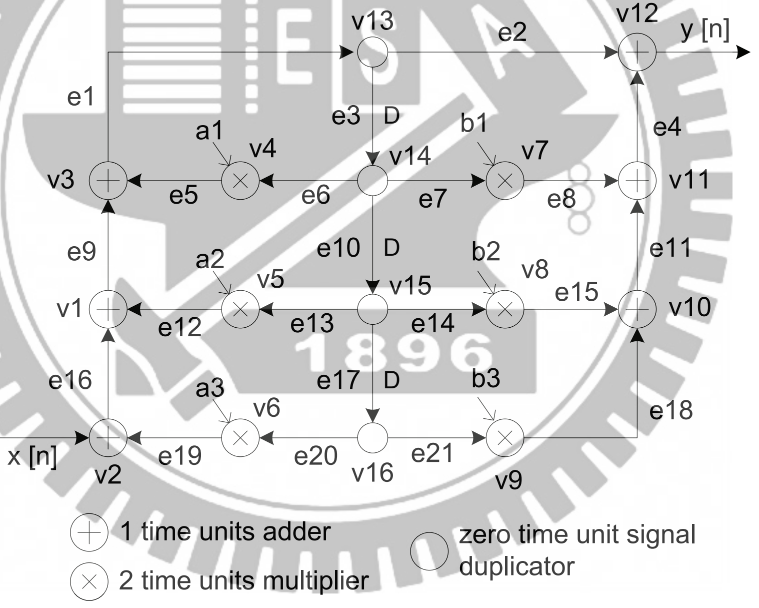
Usually, the dataflow graph design can be represented by Fully-Specified Signal Flow Graph (FSFG) [50]. Let $G_{FSFG}(V, E, D)$, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, be a three-tuple directed and edge-weighted graph. Vertex set V represents atomic operation of functional units. A vertex may have a zero execution delay, such as the signal duplicator; or may be assumed to take non-zero unit time, such as adder or multiplier. Directed edge set E describes the direction of flow of data between functional units. Inter data dependencies between functional units are denoted by weighted edges or mapping function $D : e \to \{0, 1, 2, \cdots\}, e \in E$. If a weighted edge $e \in E$ is labeled by D while $D(e) = 1$, we say that edge e has a delay element with it. Figure. 2.3(a), for instance, shows a second order IIR filter in the form of FSFG. Another example of FSFG of the third order IIR filter is also shown in Figure. 2.3(b).

To measure the efficiency of the implementation of a FSFG on a synchronous multiprocessor system, a number of bounds have been used [50–54]. Three commonly used flow graph bounds are IPB (Iteration Period Bound), PDB (Period Delay Bound) and PB (Processor Bound). We redefine these matrices as following for convenience:

- Iteration Period Bound (IPB)
  Let d_i be the executing delay of vertex $v_i \in V$. The IPB is defined as the minimum achievable latency between iterations of the given dataflow graph. If L is the set of loops and n_l is the number of delay elements around loop



(a) A second order IIR filter in the form of FSFG



(b) A third order IIR filter in the form of FSFG

Figure 2.3: The example FSFG graphs

$l \in L$, then the IPB is formulated by:

$$IPB = \max \left\lceil \frac{\sum_{j \in l} d_j}{n_l} \right\rceil, \; l \in L. \tag{2.1}$$

Note that, a FSFG is called *perfect rate* if and only if it has one delay in every loop. All perfect rate FSFGs can be statically scheduled at the iteration period bound.

- Period Delay Bound (PDB)

  The Period Delay Bound represents the minimum average throughput delay from input to output at a given iteration period. Let P be the set of paths from input to output and n_p be the number of delay elements in path p, then the PDB is defined by:

$$PDB = \max \left\lceil \sum_{j \in p} d_j - n_p \cdot IPB \right\rceil, \; p \in P. \tag{2.2}$$

- Processor Bound (PB)

  The Processor Bound represents a lower bound on the number of processors needed to meet the IPB. Let N be the set of all nodes, then processor bound is defined by:

$$PB = \left\lceil \frac{\sum_{j \in N} d_j}{IPB} \right\rceil. \tag{2.3}$$

Figure. 2.4 illustrates the performance matrices of the second order IIR filter. As showing in figure, the IPB is 20 on loop $2 \rightarrow 4 \rightarrow 2$, the periodic delay $D_{i/o}$ is 30 on path $1 \rightarrow 2 \rightarrow 8$, and the processor bound PB is 4.

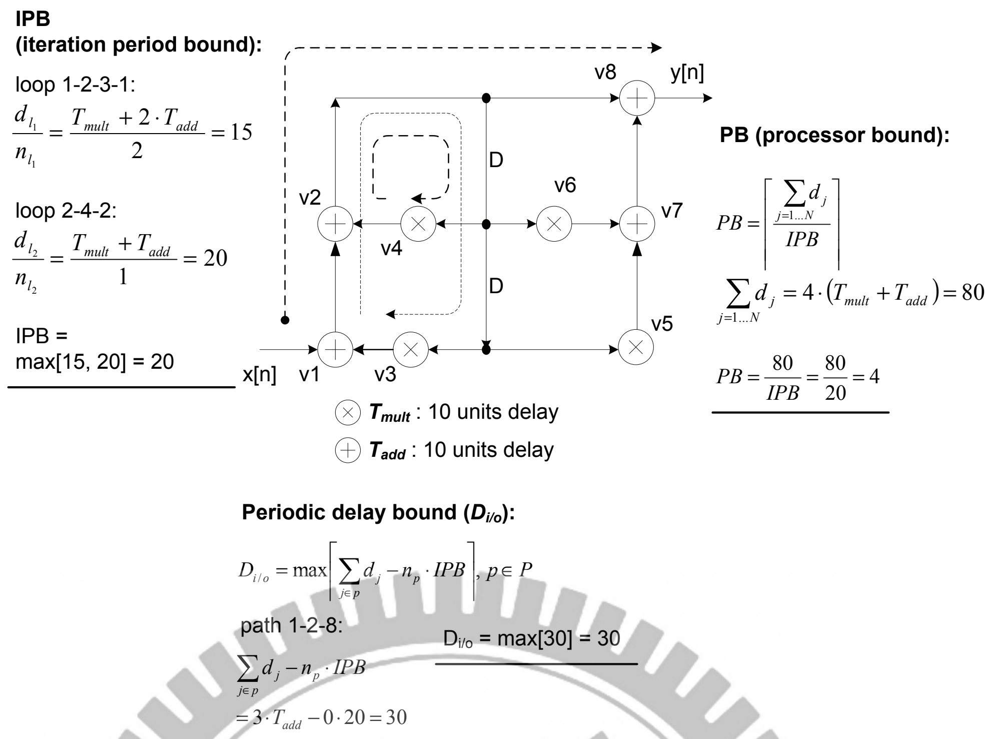

IPB
(iteration period bound):
loop 1-2-3-1:
$\frac{d_{l_1}}{n_{l_1}} = \frac{T_{mult} + 2 \cdot T_{add}}{2} = 15$
loop 2-4-2:
$\frac{d_{l_2}}{n_{l_2}} = \frac{T_{mult} + T_{add}}{1} = 20$
IPB =
max[15, 20] = 20
x[n]
v1
v2
v3
v4
v5
v6
v7
v8
y[n]
D
D
T_{mult} : 10 units delay
T_{add} : 10 units delay
PB (processor bound):
$PB = \left\lceil \frac{\sum_{j=1...N} d_j}{IPB} \right\rceil$
$\sum_{j=1...N} d_j = 4 \cdot (T_{mult} + T_{add}) = 80$
$PB = \frac{80}{IPB} = \frac{80}{20} = 4$
Periodic delay bound ($D_{i/o}$):
$D_{i/o} = \max\left[\sum_{j \in p} d_j - n_p \cdot IPB\right], p \in P$
path 1-2-8:
$\sum_{j \in p} d_j - n_p \cdot IPB$
$= 3 \cdot T_{add} - 0 \cdot 20 = 30$
$D_{i/o}$ = max[30] = 30

Figure 2.4: The performance matrices of the second order IIR filter

In the follows of the optimization procedures of HLS, these performance matrices of the dataflow can be used to measure the efficiency of the dataflow graph meeting the iteration period, delay and processor bounds of optimal goals.

2.2.2 Retiming

Retiming algorithms address the problem of moving the structural location of delay elements in a dataflow graph to improve its execution performance, silicon area, or power consumption in such a way that preserves the functional behavior of a design. The retiming algorithm was first described by Leiserson and Saxe [55, 56] using a graph model that abstracts the computation performed at each vertex of the dataflow graph design.

A *retiming* of a dataflow graph $G(V, E, D, w)$ is an integer-valued vertex labeling $r : V \mapsto \mathbb{Z}$. The integer $r(u)$ associated with vertex $u \in V$ denotes the number of delay elements being removed from each of the fanouts of u and being added to each of the fanins of u. (See Figure. 2.5). The retiming transforms graph G into the graph $G_r = (V, E, D, w_r)$, where the new edge-weight $w_r(e)$ of an edge $u \xrightarrow{e} v$ is given by $w_r(e) = w(e) + r(v) - r(u)$. Therefore, the path weight $W_r(p)$ of path $p = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots \xrightarrow{e_{k-1}} v_{k-1} \xrightarrow{e_k} v_k$ in the retimed graph is given by Equation. 2.4.

$$\begin{aligned} W_r(p) &= \sum_{i=1}^{k} \left(w(e_i) + r(v_i) - r(v_{i-1})\right) \\ &= r(v_k) - r(v_0) + W(p) \end{aligned} \tag{2.4}$$

Note that in the special case of p being a cycle, $W_r(p)$ equals $W(p)$. Thus, the path weight (i.e. the path delay) of a cycle remains invariant during retiming.

A retiming is valid if for each $e \in E, w_r(e) \geq 0$. This is a very important property for checking the retiming validity.

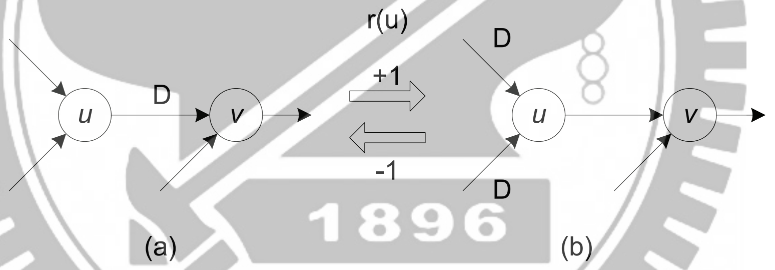


Figure 2.5: (a) An example of dataflow graph, (b) retimed dataflow graph

In practical application, retiming problem is usually modeled by using ILP (Integer Linear Programming) [57]. Given a dataflow design, its FSFG graph may not be rate-optimal; in other words, the FSFG graph does not reach the optimal IPB bound. Retiming is a process that may help making sample rate equal to IPB. With delay transfer or nodal transfer, it is possible to make a loop achieve IPB. ILP for retiming problem has been proposed by [58]. We redescribe in Algorithm. 1.

Algorithm 1 ILP modeling for retiming problem

```
1: Minimize: $IPB$
2: Subject: Precedence constraints: Node $v$ precedes node $w$ (i.e. arc of $v \rightarrow w$)
3:
4: $t_w - t_v > d_w - n_{vw} \cdot IPB$, $\forall e\,(v, w) \in E$.
5: Non-negativity constraints: $IPB, t_j \geq 0$, for $i = 1, \cdots, N$
```

2.2.3 FSFG Unfolding

In previous subsection, we know that retiming technique can be used to convert a FSFG graph to an optimal FSFG design reaching its IPB bound. However, not all of the FSFG graphs can be made to rate-optimal only using retiming. To solve this problem, Parhi and Messerschmitt proposed the graph-domain unfolding transformation for FSFG graph in their works [49, 59, 60]. For a given FSFG graph, if there is N nodes, the f-unfolded FSFG has $f \times N$ nodes, and the IPB is also f times larger than the original FSFG. The total number of delay elements remains unchanged. The Parhi-Messerschmitt unfolding (by f) algorithm that operates directly on the FSFG is redescribed by Algorithm. 2.

An example of FSFG unfolding is given in Figure. 2.6. In the original FSFG Figure. 2.6(a), its iteration period boubd IPB is 2, however, not all of two loops in (a) have only one delay in it. In other words, the original FSFG is not a *perfect rate* FSFG. After unfolding by 2, the unfolded graph in Figure. 2.6(b) is a *perfect rate* FSFG with $IPB = 4$, however, two iterations are completed in that, resulting in an equivalent IPB = 2 in the original FSFG.

Algorithm 2 Unfolding a FSFG by f

```
 1: for all $v_j \in V$ do
 2:     draw $f$ nodes labeled $v_j^i$, $i = 1, \cdots, f$
 3: end for
 4:
 5: for all edge $e \in E(u, v)$ with zero dealys in original FSFG do
 6:     draw arcs $u_k$ to $v_k$ with zero delay, $k = 1, \cdots, f$
 7: end for
 8: for edge $e \in E(u, v)$ with $i$ dealys in original FSFG do
 9:     if $i < f$ then
10:         draw arcs $u_{q-i}$ to $v_q$ with zero delay, $q = i + 1, \cdots, f$
11:         draw arcs $u_{f-i+q}$ to $v_q$ with one delay, $q = 1, \cdots, i$
12:     else if $i \geq f$ then
13:         draw arcs $u_m$ to $v_q$ with $z$ delays, $m = (z \times f) - i + f$
14:         and $z = \lceil (i - q + 1)/f \rceil$, $q = 1, \cdots, f$
15:     end if
16: end for
```

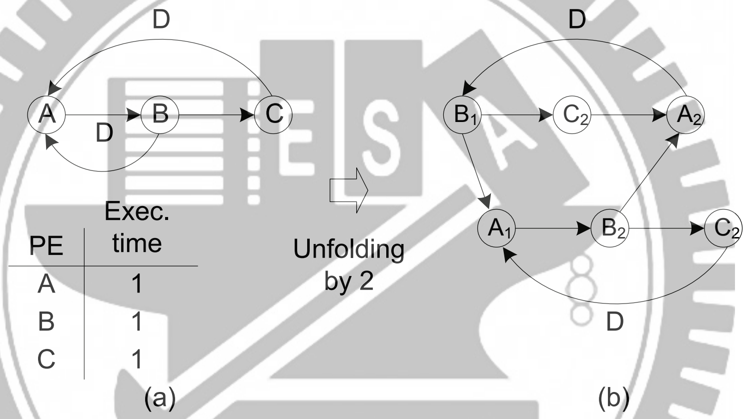


Figure 2.6: (a) Original FSFG with IPB=2, (b) Unfolding by 2 with IPB=2 (since two iterations are completed resulting in an equivalent IPB of 2 in the original FSFG)

2.2.4 FSFG Scheduling

Before mapping an FSFG design into a hardware, the execution start time of each task must be determined. A *static* schedule of a cycle FSFG is a repeated pattern of an execution of the corresponding loop. And a static schedule must obey the precedence relations of the directed acyclic graph (DAG) portion of a FSFG design that is obtained by removing all edges with delays from that FSFG. A *sequencing graph* is a DAG $G_s(V, E)$, where vertex set $V = \{v_i \mid i = 1, 2, \ldots, n\}$ is in one-to-one correspondence with the set of the FSFG design, and edge set $E = \{(v_i, v_j) \mid i, j = 1, 2, \ldots, n; i \neq j\}$ is representing their dependencies. Different scheduling algorithms have been proposed in [58, 59, 61] addressing different constrained problems to find the desired schedule. The desired schedule have to satisfy the precedence constraints specified by the sequencing graph. A schedule S to the FSFG design is represented in space-time ($P \times T$) domain. The *abscissa* denotes time axis, $[1, le(S)]$, where $le(S)$ is the length of the schedule. The *ordinate* denotes the processor space, $[1, n_{res}]$, where n_{res} is the total number of processors that implement each task. During the period of the i-th iteration, schedule determines the start times of all nodes in FSFG. Let op_j^i be a task, which is corresponded to each vertex $v_j \in V$ of a given FSFG in the ith iteration. A schedule of the given FSFG, $G_{FSFG} = (V, E, D)$, is a function $\varphi : V \to \mathbb{Z}^+$, which arranges each task node op_j^i to begin its execution at the time step $\varphi\left(op_j^i\right)$, where $\mathbb{Z}^+ = \{1, 2, \ldots\}$ is the positive integer.

Assuming d_j is the execution delay for each task node op_j^i, the length $le\,(S)$ of a schedule S is the latest finish time of all the operations scheduled, that is $le\,(S) = \max\left\{\varphi\left(op_j^i\right) + d_j - 1 | \forall op_j^i \in V\right\}$. For each task node $op_j^i \in V$, a schedule of the given FSFG is as following:

- Start time: $t_j^i = \varphi\left(op_j^i\right),\ \varphi : V \to \mathbb{Z}^+ = \{1, 2, \ldots\}$
- Execution delay: $d_j \in \mathbb{Z} = \{0, 1, 2, \ldots\}$
- Finish time: $\varepsilon_j = \varphi\left(op_j^i\right) + d_j - 1$
- Task assignment:

$pe_j^i = \tau(op_j^i), \tau : V \to \{1, 2, \ldots, n_{res}\}$

- Length of the schedule:

  $le(S) = \max\left\{\varphi\left(op_j^i\right) + d_j - 1 | \forall op_j^i \in V\right\}$

- The earliest task-finished step:

  $t_{etf} = \min\left\{\varphi\left(op_j^i\right) + d_j - 1 | \forall op_j^i \in V\right\}$

An example schedule of the second order IIR filter, for instance, is shown in Figure. 2.7.

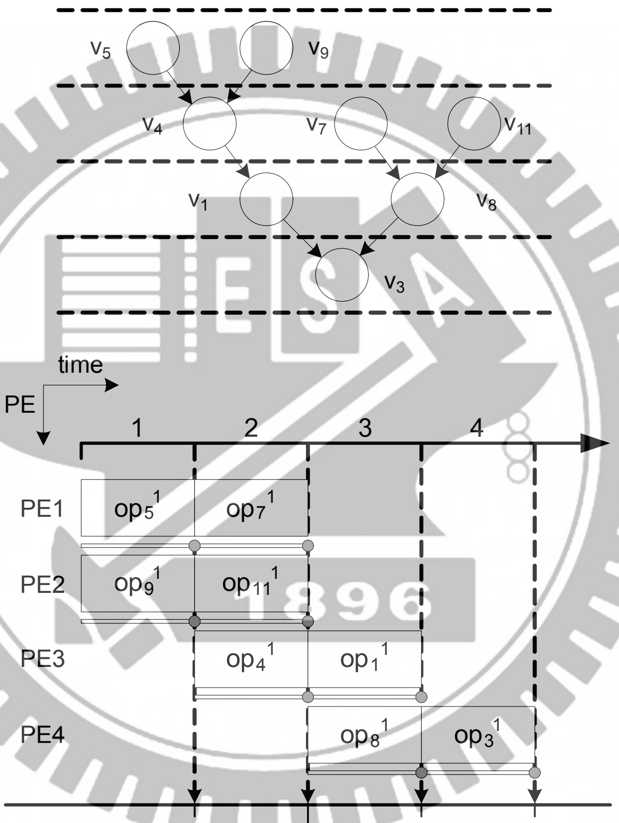


Figure 2.7: An example schedule of the second order IIR filter of Figure. 2.3(a)

In practical application, four scheduling algorithms are commonly used in HLS, and they are ASAP (as soon as possible) / ALAP (as late as possible) scheduling [48], list scheduling [62], force-directed scheduling [63] and ILP (Integer Linear Programming) scheduling [57, 64]. The first four algorithms are also called the heuristic method. The heuristic method initially utilizes ASAP / ALAP scheduling [48], then the algorithm iteratively selects and schedules one operation at a time into an appropriate control step. Since the greedy strategies make a series of local decisions, it may miss the globally optimal solution. However, heuristic method do produce results quickly, and the results may be sufficient for practical application. The last scheduling algorithm is based on solving formulas. The ILP method is guaranteed to find globally optimal schedule, although at the cost of more processing time. In contrast to the first one, ILP based scheduling produces a schedule for all operations simultaneously.

2.3 PN: Petri Net

Petri Net is a graphical and mathematical modeling tool applicable to many applications, especially the parallel or concurrency systems. The primary concept was original from Carl Adam Petri in his Ph.D thesis since 1962 [65]. Readers may refer more literatures about history and survey works from [66–73].

Formally, a Petri Net $G_{PN}(P, T, W, M_0)$ can be represented by a four-tuple graph [70], where $P = \{p_1, \ldots, p_n\}$ and $T = \{t_1, \ldots, t_m\}$ are finite sets of *place* and *transition*, W is the weighted flow relation, and M_0 is the initial *marking*. Visually in graph, places are drown as circles and transitions as boxes or bars. A marking (state) is a mapping function $M : P \rightarrow \{0, 1, 2, \cdots\}$. If $M(p_i) = k$ for place p_i, we say that place p_i is *marked* with k tokens. In graphic, we draw k block dots (tokens) inside place p_i if $M(p_i) = k$. The initial marking M_0 denotes the number of tokens that all places are marked initially. A marking M is said to be a *valid state* if and only if $M(p_i) \geqslant 0$, $\forall p_i \in P$. Let u and

v be two arbitrary adjacent nodes of PN. If $W(u,v) > 0$, then there is an arc from u to v with weight $W(u,v)$. In this research, we assume that $W(u,v) = 1$ for each node pair. For a node u in $P \cup T$, ${}^{\bullet}u$ (the pre-set of u) is specified by: ${}^{\bullet}u = \{v \in P \cup T | W(v,u) > 0\}$ and $u^{\bullet}$ (the post-set of u) is specified by: $u^{\bullet} = \{v \in P \cup T | W(u,v) > 0\}$. In our work, for each place $p_i \in P$, we only alow p_i has only one output transition, that is $\forall p_i \in P, |p_i{}^{\bullet}| = 1$.

When considering a condition-event system, places with marking (state) represent the conditions, while transitions represent the events. A transition (event) has certain number of input and output places representing the pre-conditions and post-conditions of the event. The presence of a token in a place is interpreted as holding the truth of the condition associated with the place. In [67], the author gave some typical interpretations of places and transitions. We redescribe these interpretations in Table. 2.2.

Table 2.2: Typical interpretations of places and transitions

Input places	Transition	Output places
Preconditions	Event	Postconditions
Input data	Computation step	Output data
Input signals	Signal processor	Output signals
Resources needed	Task or job	Resources released
Conditions	Clause in logic	Conclusions
Buffers	Processor	Buffers

The *execution* of a Petri Net is controlled by the number and distribution of tokens. Tokens reside in the places and control the execution of the transitions of the net. A Petri Net executes by *firing* transitions. A transition fires by removing tokens from its input places and creating new tokens which are distributed to its output places. A transition may fire if it is *enabled*. A transition tr is *enabled* at marking M (denoted by $M[tr\rangle$) if $\forall p \in {}^{\bullet}tr : M(p) \geqslant W(p,tr)$. Once a transition tr is enabled at M, it may fire and then reach a new marking M' (denoted by

$M[tr\rangle\, M')$. The occurrence of tr lead to a new marking M', defined for each place p by

$$M'(p) = M(p) - W(p, tr) + W(tr, p)\,. \tag{2.5}$$

A sequence of transitions $\sigma = tr_1 \cdots tr_{k-1} \in T^*$ is a *firing sequence* from a marking M_1 to a marking M_k if and only if there exist markings $M_2, \ldots, M_{k-1}$ such that

$$M_i[tr_i\rangle\, M_{i+1}, \text{ for } 1 \leqslant i \leqslant k-1. \tag{2.6}$$

Marking M_k is said to be *reachable* from M_0 if and only if there exists a firing sequence $\sigma : M_0[\sigma\rangle\, M_k$. $[M\rangle$ is the set of markings reachable from M by firing any sequence of transitions, i.e., $M' \in [M\rangle \Leftrightarrow \exists \sigma \in T^* : M[\sigma\rangle\, M'$. $[M_0\rangle$ is the set of all markings reachable from M_0.

Matrix representation of PN is defined by *incidence matrix* A (also called the characteristic matrix), which is a $|P| \times |T|$-matrix with entries

$$A_{ij} = W(tr_j, p_i) - W(p_i, tr_j)\,. \tag{2.7}$$

The matrix representation usually gives a complete characterization of PN. Let $x_j = \{tr_j\} = (\ldots, 0, 1, 0, \ldots)$ be the unit $|T| \times 1$ column vector which is zero everywhere except in the j-th element. Also, let μ is the $|P| \times 1$ column vector respected to a marking M_0 with entries $\mu_i = M_0(p_i)$. The transition tr_j is represented by the column vector x_j. A transition tr_j is enabled at a marking M_0(denoted by $M_0[tr_j\rangle$) if $\mu \geqslant A \cdot x_j$. And the result marking, μ', of firing enabled transition tr_j in a marking μ_0 is represented by

$$\mu' = \delta\left(\mu_0, tr_j\right) = \mu_0 + A \cdot x_j. \tag{2.8}$$

Where $\delta : M \times T \to M$ is a transition function mapping current marking state μ_0 and current firing transition into the next next marking state μ'. For a sequence of transition firing $\sigma : M_0[\sigma\rangle\, M_k$ and $M_i[tr_i\rangle\, M_{i+1}$, $1 \leqslant i \leqslant k-1$, we have:

$$\begin{aligned}\delta\left(\mu_0;\sigma\right) &= \delta\left(\mu_0; tr_1 tr_2 tr_3 \ldots tr_{k-1}\right) \\ &= \mu_0 + \textstyle\sum_1^{k-1} A \cdot x_j \\ &= \mu_0 + A \cdot f\left(\sigma\right).\end{aligned} \tag{2.9}$$

The vector $f\left(\sigma\right)$ is *firing vector* of the sequence $\sigma = tr_1 \ldots tr_{k-1}$. The i-th element of $f\left(\sigma\right)$, $f\left(\sigma\right)_i \in \mathbb{Z}$, is the number of times that transition tr_i fires.

The *reachability tree* is the important technique analyzing the reachability marking set of a PN model. The tree structure shows which states of the PN can be reached with arbitrary transition firing sequences starting from the initial marking state M_0. A reachability tree is a hierarchical graph with directed arcs. The nodes of the tree represent PN marking state and the arcs represent transitions firing. The *root* node represents the initial net state and the leaf nodes are reachable net states. *Duplicate* nodes reflect multiple net states in the tree. The *terminal* nodes represent the end of valid paths through the tree and can also represent net states. To prevent infinite number of markings which result from loops in a PN model, a special symbol ω is introduced in reachability analysis [66, 67]. For any constant a, ω is defined by:

$$
\begin{aligned}
&w + a = w \\
&w - a = w \\
&a < w \\
&w \leqslant w
\end{aligned}
\tag{2.10}
$$

By using ω notation, an PN tree constructing algorithm is given in Algorithm. 3. An example PN is given in Figure. 2.8(a), and its reachability using ω notation is given in Figure. 2.8(b).

Algorithm 3 Algorithm of constructing a reachability tree from a PN

```
1:  Initialize the root node of the tree with $x = \mu_0$
2:  for all node $\mu$ to be inserted, evaluate $\delta(x, tr_j), \forall tr_j \in T$ do
3:      if $\delta(x, tr_j)$ is undefined, $\forall t_j \in T$ then
4:          the $x$ is a terminal node
5:      else if $\delta(x, tr_j)$ is defined for at least one $tr_j \in T$ then
6:          create a new node $x' = \delta(x, t_j)$
7:          if $x'(p_i) = \omega, \forall pi \in P$ then
8:              defien $x'(p_i) \leftarrow \omega$
9:          else if a node $y$ exists and $x'(p_i) > y(p_i), \forall p_i \in P$ then
10:             set $x'(p_i) \leftarrow \omega, \forall p_i \in P$
11:         else
12:             $x'(p_i) \leftarrow x'(p_i)$
13:         end if
14:     end if
15: end for
16: if all nodes are either terminal nodes or duplicate nodes then
17:     stop tree construction
18: end if
```

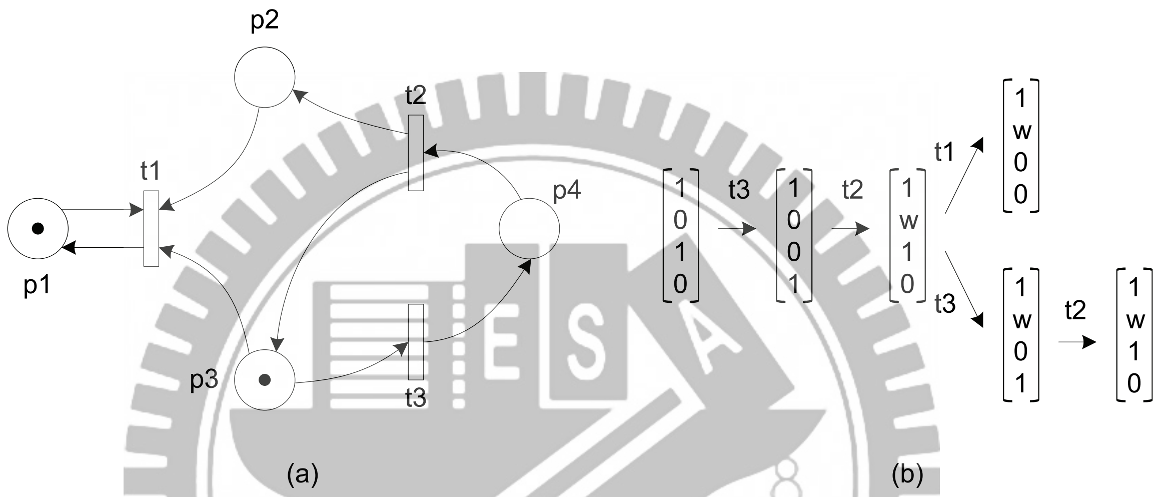


Figure 2.8: (a) A Petri Net to illustrate the construction of a reachability tree, (b) the reachability tree of the Petri Net

CHAPTER 3

Modeling and Formal Verification of Dataflow Graph in System-Level Design Using Petri Net

3.1 Introduction

In this chapter, we address a PN (Petri Net) based formal verification method for dataflow or DSP-driven algorithms of HLS (High-Level Synthesis) result. Traditionally, given a DSP-driven algorithm, the arithmetic transformations [60], such as retiming and unfolding techniques [59, 61], and the scheduling techniques are used to achieve optimal system specification goals in HLS. These system-level design techniques are usually applied iteratively and error-prone. In order to detect design faults of system-level design, system-level verification is introduced in design flow. In this work, two-phases verification including static and dynamic phases are proposed, and Figure. 3.1 illustrates the verification flowchart.

As showing in flowchart, the dataflow algorithm is usually represented by a Fully-Specified Signal Flow Graph (FSFG). To check a DSP-driven algorithm in PN domain, the design is first converted into a PN (Petri Net) model by using proposed conversion method and then apply verification techniques in PN domain. In static phase, the target is to verify the arithmetic transformations of HLS. Since, the procedures of retiming or unfolding in FSFG domain can be seen a serial token movement in PN domain, the verifier can check the correctness of the arithmetic transformations by checking the validation of PN token movements in PN domain.

In dynamic phase, PN model is used to verify the schedule schemes of given FSFG. In order to map DSP algorithm on a hardware circuit, scheduling techniques are applied to find suitable schedules of FSFG. For high performance issues, such as smaller silicon area, higher throughput rate, and lower power consumption, these algorithms start from as-late-as-possible (ALAP) to as-soon-as-possible (ASAP) schedule time and assign the starting time of each task of FSFG [74]. The execution of all tasks is referred to as an iteration. Within an iteration, the execution sequences are different from various scheduling algorithms, but the original behavior is not changed. Once a schedule scheme of FSFG is produced from scheduling algorithms, the permanent behavior of system and schedule scheme are transformed and verified in PN domain as well in static phase.

The PN-based verification can be considered as a model checker with capability of checking a number of properties that should be hold under system specification. In PN domain, the inputs to the model checker include a characteristic matrix describing the system and the system properties described in matrix equations. By applying mathematics theory and matrix manipulation, the model checker reports whether the design is valid. When comparing with FSFG, designing DSP algorithm at FSFG domain and verifying it at PN domain provides double-checking verification solution. Additionally, PN theory also provides primary property analysis techniques of system, such as reachability, liveness, safeness, boundedness, and reversibility [75] that FSFG can not be offered.

This chapter is organized as follows. First, we describe the methodology of

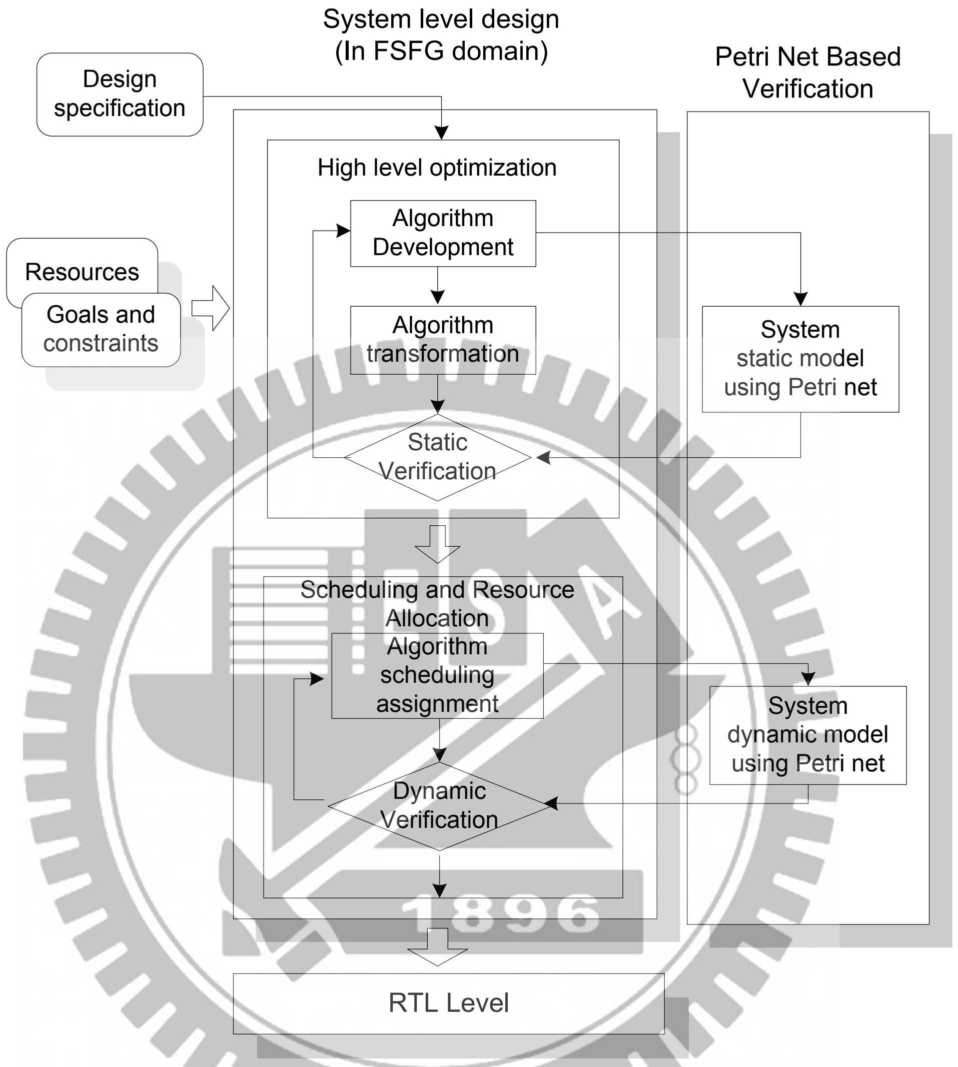


Figure 3.1: System-level design flow and the PN-based verification

conversion from FSFG to PN model. Then, PN based high-level verification flow is presented in Section. 3.3. In Section. 3.4, we introduce our verification software application tool called HiVED (High-Level Verification Engine for DSP) and show some experimental results. Finally, we make some concluding remarks in the latest section.

3.2 Conversion from FSFG to PN model

The FSFG is attractive to algorithm developers because it directly models the equations of DSP algorithm. Yet, it does not sufficiently unveil the dynamic behavior and the implementation limits in terms of the degree of parallelism and the memory requirement. Thus, we use Petri net to model DSP algorithms. It also allows us to discover the characteristic of the target architecture and to observe the dynamic behavior of the algorithm.

The FSFG $G_{FSFG}(V, E, D)$ of a DSP-driven algorithm can be modeled as PN $G_{PN}(P, T, W, M_0)$ by applying following rules:

1. Vertex set V, whose elements have the computational power, in FSFG domain is transformed into transition set T in PN domain.

2. Edge set E in FSFG domain is transformed into place set P in PN domain.

3. Since each place in PN has only one output transition, the pseudo transition is added as the signal duplicator which is corresponded to the fork node in FSFG.

4. The delay element set D in FSFG domain is corresponded to the number of tokens in places in PN domain. In static analysis, tokens in a PN model can represent delay elements of a FSFG; thus, retiming delay elements between

edges in a FSFG graph can be seen as moving tokens between places in a PN model. In dynamic analysis, executions of vertices in a FSFG graph can represent moving tokens between places in a PN model.

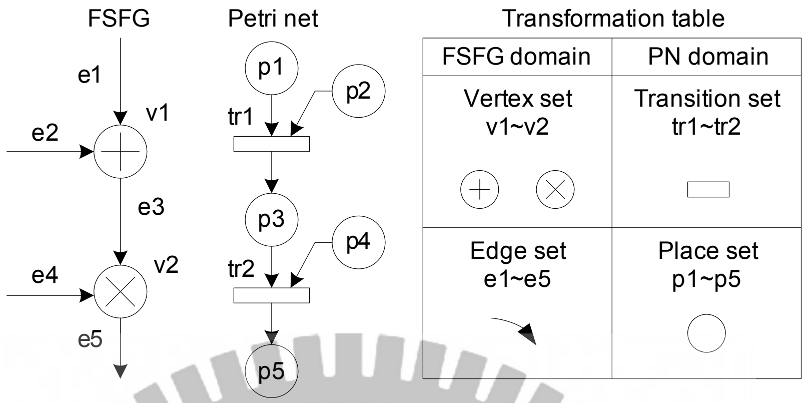


Figure 3.2: Transformation from FSFG to Petri net

Figure. 3.2, for instance, illustrates the transformation from vertex set V and edge set E to transition set T and place set P. The vertex set $V = \{v_1, v_2\}$ and the edge set $E = \{e_1, e_2, e_3, e_4, e_5\}$ in FSFG domain are transformed into the transition set $T = \{tr_1, tr_2\}$ and the place set $P = \{p_1, p_2, p_3, p_4, p_5\}$ in PN domain with respect. Another example is given in Figure. 3.3, a FSFG graph with delay elements is transformed into a Petri net. The vertex set $V = \{a, b, c\}$ and the edge set $E = \{e_1, e_2, e_3\}$ of FSFG are transformed into the transition set $T = \{tr_1, tr_2, tr_3\}$ and the place set $P = \{p_1, p_2, p_3\}$ of PN model. The delay elements in FSFG domain is donated by the number of tokens in places, such that $M_0(p_1) = 0, M_0(p_2) = 1$ and $M_0(p_3) = 2$.

In FSFG domain, a computing result of a functional element is one or more other functional elements' inputs. Data source in the prior functional element causes a data fork point. A fork point in FSFG can be modeled as a pseudo-transition in PN model. The pseudo-transition duplicates copies of data source

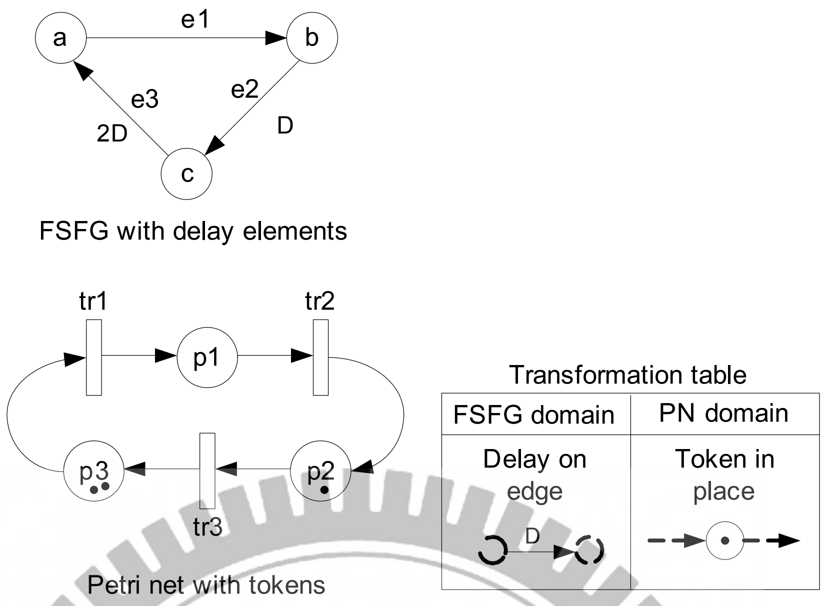


Figure 3.3: FSFG with delay elements and Petri net with tokens

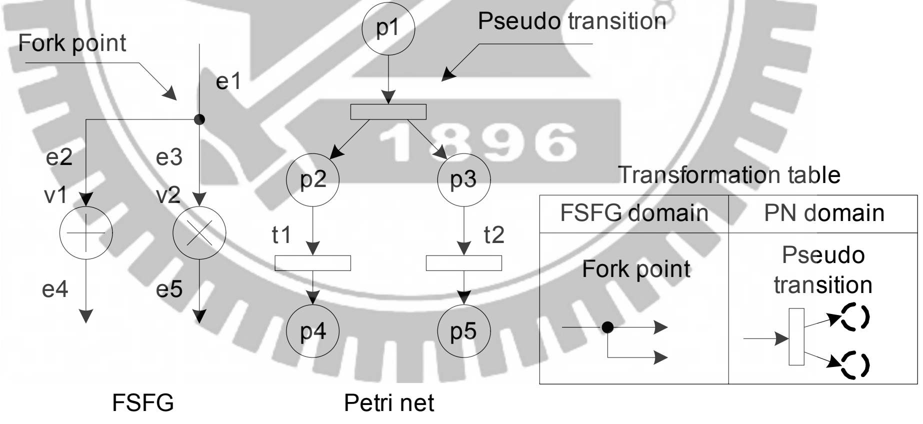


Figure 3.4: Fork point in FSFG domain and pseudo-transition in PN domain

as many as the output nodes in FSFG graph. The equivalence graph of fork point in FSFG domain and pseudo-transition in PN domain is shown in Figure. 3.4. In Figure. 3.5, a PN model example of the second-order IIR filter of Figure. 2.3(a) is shown. Anther example illustrating the PN model of the third-order IIR filter of Figure. 2.3(b) by applying above transformation rules shows in Figure. 3.6 while the characteristic matrix A with the initial marking m shows the matrix representation of the PN model.

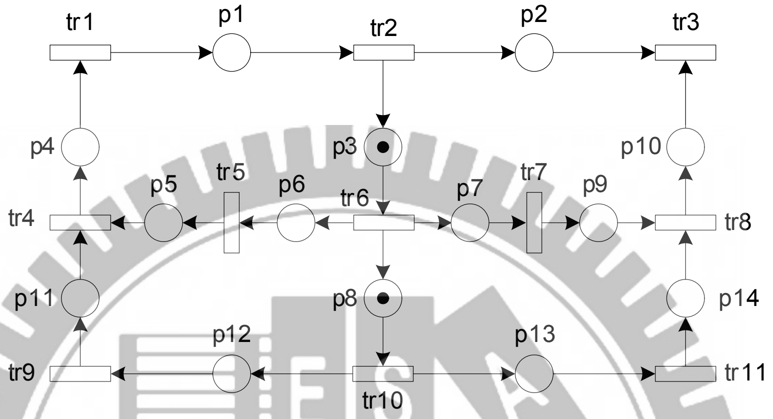
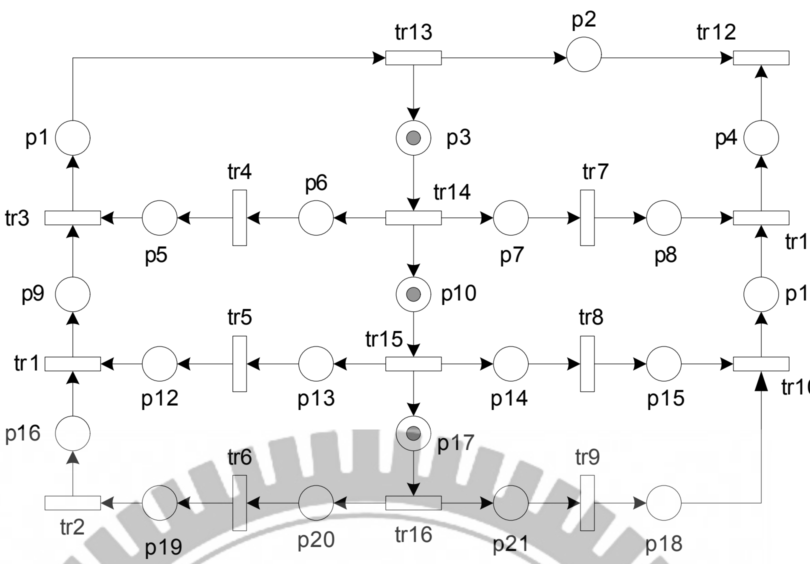


Figure 3.5: PN model of the second order IIR filter



$$[A] = \begin{array}{c|cccccccccccccccc}
\text{P/T} & \text{tr1} & \text{tr2} & \text{tr3} & \text{tr4} & \text{tr5} & \text{tr6} & \text{tr7} & \text{tr8} & \text{tr9} & \text{tr10} & \text{tr11} & \text{tr12} & \text{tr13} & \text{tr14} & \text{tr15} & \text{tr16} \\
\hline
\text{p1} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
\text{p2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\
\text{p3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
\text{p4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
\text{p5} & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p6} & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{p7} & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{p8} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
\text{p9} & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p10} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
\text{p11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
\text{p12} & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p13} & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{p14} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\text{p15} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p16} & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p17} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
\text{p18} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p19} & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{p20} & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\text{p21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\qquad
m = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Figure 3.6: A PN graph and the matrix representation to the third-order IIR filter of Figure. 2.3(b).

3.3 High-Level Verification

In last section, we have addressed the conversion method to convert a FSFG graph into PN domain and use PN characteristic matrix to represent a design system. Since, retiming process or tasks executing for the HLS of the original FSFG design can be seen token movement of PN model in PN domain. Therefore, we can use PN theory to check the correctness of arithmetic transformations or scheduling of HLS. In the following, two-phases verification method including static and dynamic phases are presented.

3.3.1 Static Verification

The target of the static phase verification is to verify the arithmetic transformations including the unfolding and retiming techniques. In FSFG domain, the arithmetic transformations are applied to the original FSFG design attempting to obtain optimal FSFG [59–61]. These optimization techniques are typically used to reduce the critical path delay, silicon area, number of latches and power consumption. However, these algorithms are error-prone and have a lot cost that limits their use. High-level verification for the arithmetic transformations can check the correctness of HLS in the early stage of the design flow. In designers' point of view, if we can quickly tell whether the HLS result of the design is correct, then the redesigning time or costs can be reduced.

The flowchart to the static phase verification flow is shown in Figure. 3.7. In HLS, designs may apply arithmetic transformations on DUV design iteratively in their design flow. We have stated in previous section that the designing procedures of arithmetic transformations can be seen a serious token movements in PN domain. The designing parameters, which include delay R_i, unfolding factor

f, and final delay R_f of DUV, are transformed into PN model as initial token $\mu_{0(|P|\times 1)}$, unfolding factor f, and final token $\mu'_{(|P|\times 1)}$ with respect. While comparing to the unfolding technique in FSFG domain, the converted PN model is also unfolded in PN domain. Before applying retiming checking, we first address unfolding algorithm of PN model in next section.

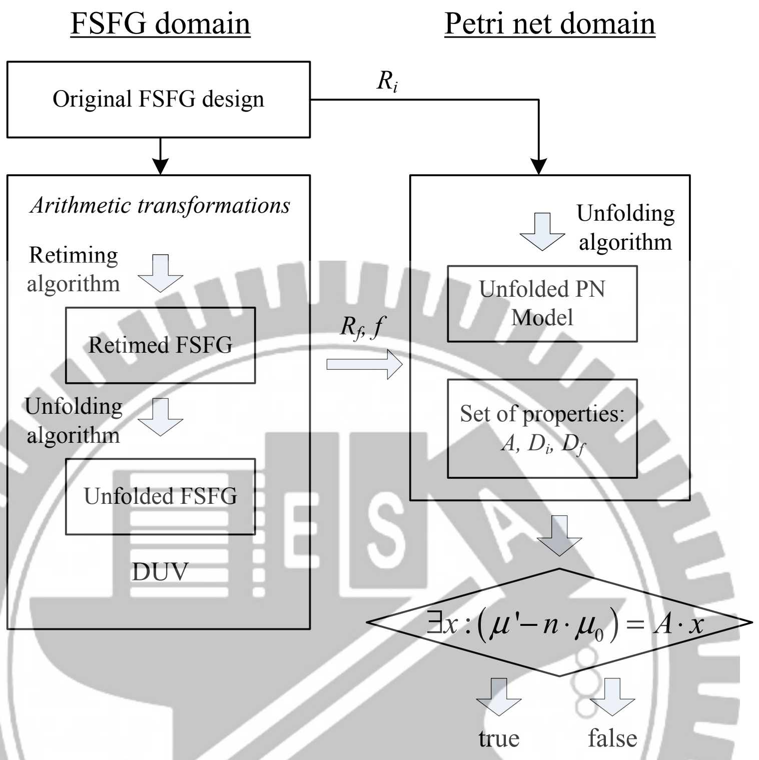


Figure 3.7: Flowchart of static phase verification.

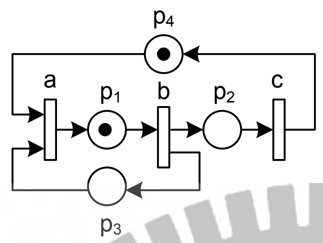
3.3.2 PN Unfolding

Let $A_{(|P|\times|T|)}$ and $d_{(|P|\times 1)}$ be the characteristic matrix and the initial delay vector of a PN model. Assuming its unfolding PN model has PN matrix $A'_{f\cdot|P|\times f\cdot|T|}$ and initial delay vector $bd_{(f\cdot|P|\times 1)}$. For each element $a'(i,j)$ in A, there is a submatrix $b_{i,j}$ in dimension $(f \times f)$ with it. The value of all element $b_{i,j}(s,t)$ and $bd_i(s)$ depend on delay $d(i)$ and following PN unfolding Algorithm. 4.

Algorithm 4 Unfolding a Petri Net by f

```
1: for all $s = 1, 2, \cdots, f$ and $t = 1, 2, \cdots, f$ do
2:     if $d(i) = 0$ then
3:         $b_{i,j}(s, s) = 1$
4:         $bd_i(s) = 0, s = \{1, 2, \cdots, f\}$
5:     else if $d(i) < f$ then
6:         if $a(i, j) = 1$ then
7:             $b_{i,j}(s, s) = 1$
8:         else if $a(i, j) = -1$ then
9:             $b_{i,j}(s, s + d(i)) = -1, s = \{1, 2, \cdots, f - d(i)\}$
10:            $bd_i(p - s + 1) = 1, s = \{1, 2, \cdots, d(i)\}$
11:        end if
12:    else if $d(i) = f$ then
13:        if $a(i, j) = -1$ then
14:            $b_{ij}(s, s) = -1$
15:        else if $a(i, j) = 1$ then
16:            $b_{ij}(s, s) = 1$
17:            $bd_i(s) = 1, s = \{1, 2, \cdots, f\}$
18:        end if
19:    else if $d(i) > f$ then
20:        if $a(i, j) = -1$ then
21:            $b_{i,j}(s, s) = -1$
22:        else if $a(i, j) = 1$ then
23:            $b_{i,j}(s, f - mod(d(i), f) + 1) = 1,$
24:            $bd_i(s) = \lceil d(i)/p \rceil, s = \{1, 2, \cdots, mod(d(i), f)\};$
25:            $b_{i,j}(s + mod(d(i), f), s) = 1,$
26:            $bd_i(s) = \lfloor d(i)/f \rfloor, s = \{1, 2, \cdots, f - mod(d(i), f)\}.$
27:        end if
28:    end if
29: end for
```

Petri Net domain



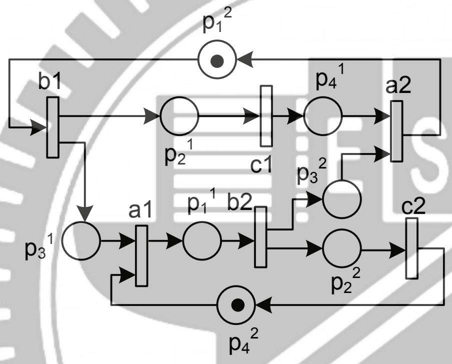
(a)

Characteristic matrix

$$\begin{array}{c|ccc} & a & b & c \\ \hline p_1 & 1 & -1 & 0 \\ p_2 & 0 & 1 & -1 \\ p_3 & -1 & 1 & 0 \\ p_4 & -1 & 0 & 1 \end{array}$$

$D_i=[1\ 0\ 0\ 1]^{-1}$

(c)



Unfolded PN

(b)

$$\begin{array}{c|cc|cc|cc} & a1 & a2 & b1 & b2 & c1 & c2 \\ \hline p_1^1 & 1 & 0 & 0 & -1 & 0 & 0 \\ p_1^2 & 0 & 1 & -1 & 0 & 0 & 0 \\ p_2^1 & 0 & 0 & 1 & 0 & -1 & 0 \\ p_2^2 & 0 & 0 & 0 & 1 & 0 & -1 \\ p_3^1 & -1 & 0 & 1 & 0 & 0 & 0 \\ p_3^2 & 0 & -1 & 0 & 1 & 0 & 0 \\ p_4^1 & 0 & -1 & 0 & 0 & 1 & 0 \\ p_4^2 & -1 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$D_i=[\ 0\ 1\ |\ 0\ 0\ |\ 0\ 0\ |\ 0\ 1\ ]^{-1}$

(d)

Figure 3.8: (a) A example PN model, (b) the 2-folded PN model , (c) PN characteristic matrix of (a) and (d) PN characteristic matrix of (b)

Figure. 3.8(a) is an example PN model, and Figure. 3.8(b) is its unfolded PN model by using the above rules. Figure. 3.8(c) and (d) are the PN characteristic matrices of (a) and (b) respectively.

3.3.3 Retiming

After applying PN unfolding rule, the unfolded PN model with PN characteristic matrix, initial and final marking states are used to check the correctness of retiming process. Let μ_0 be the initial marking state, A be the characteristic matrix, x be the firing transition vector and $\mu\prime$ be the marking state after x firing. The firing process can be expressed by a matrix equation by Equation. (2.8) and is rewritten by Equation. (3.1):

$$\mu\prime = \mu_0 + A \cdot x_j. \tag{3.1}$$

The delays of FSFG can also be scaled by a positive integer number n without affecting the functional behavior of the FSFG, thus it becomes:

$$\mu' = n \cdot \mu_0 + A \cdot x, \ \ or \ A \cdot x = (\mu' - n \cdot \mu_0)\,. \tag{3.2}$$

By using linear system theory [76], equation $y = A \cdot x$ has a unique solution x if and only if matrix $[A]$ and matrix $[A|y]$ have the same *rank*. In our case, $y = (\mu' - n \cdot \mu_0)$ is known and A is the characteristic matrix of PN. Therefore, we can check the correctness of HLS result by checking the equivalence of the matrix rank. Finally, as showing in Figure. 3.7, the verifier reports true if x has solution and false else.

3.3.4 Dynamic Verification

In dynamic phase, the target is to verify the tasks executing schedule of FSFG design. The flowchart to the dynamic verification flow is shown in Figure. 3.9. In FSFG domain, scheduling algorithms are applied to the FSFG and produce desired schedule schemes. A schedule of a given FSFG includes the start time $s(j) = t_j$ and execution delay $c(j) = d_j$ of each PE (process element) v_j of FSFG, that is:

$$s(v_j) = \{t_j \in \mathbb{N} | 0 \leq t_j \leq IP\}, \tag{3.3}$$

$$c(v_j) = \{d_j \in \mathbb{N} | v_j \in V = \{v_1, \cdots, v_n\}\}. \tag{3.4}$$

Within one iteration period (IP), a schedule scheme gives the executing sequence of all PE of V. Generally, various schedule schemes that need to be verified are either solved by Integer Linear Programming (ILP) [74, 77] or found by heuristic method. The solution of schedule scheme is in *binary decision variable matrix X* or Gantt chart form. When a schedule of FSFG design is produced, the DUV schedule and the original FSFG are transformed into PN model. As describing in previous section, moving tokens between places is meaning firing the executable transitions. Thus, schedule verification in PN domain can be seen as PN reachability problem.

Considering timing interval within one IP, the marking μ_l, where $l \in \mathbb{N}$ and $1 \leq l \leq IP$ is the token state of the corresponded time step. For each time step l, the executable PE set x_l is defined as:

$$x_l = \{v_j \in V | s(v_j) + c(v_j) - 1 = l\}. \tag{3.5}$$

Therefore, there exists firing sequence set σ in PN domain with respected to x_l in FSFG domain. The verifier check whether there is only one initial marking M_0 for each time step that satisfied the following equation:

$$\exists M_0 : M_0[\sigma\rangle M_l. \tag{3.6}$$

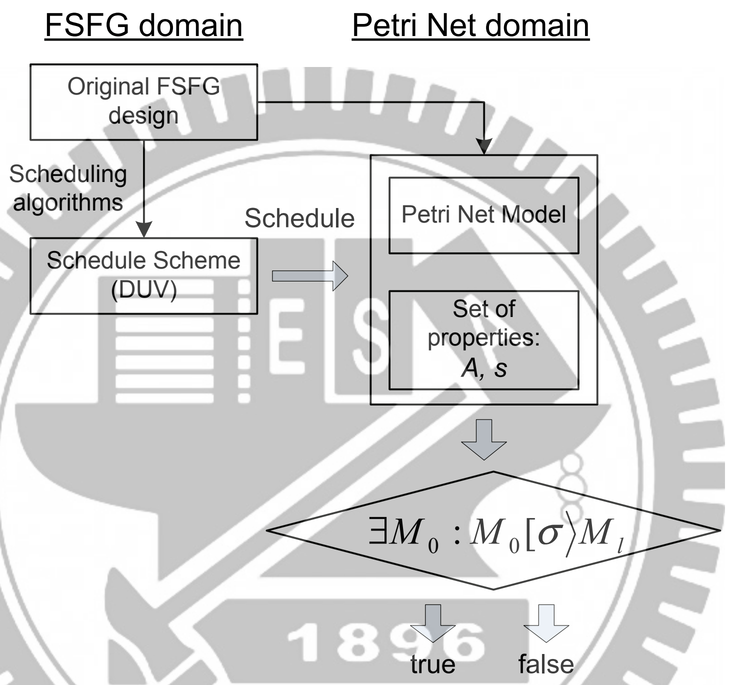


Figure 3.9: Flowchart of dynamic phase verification.

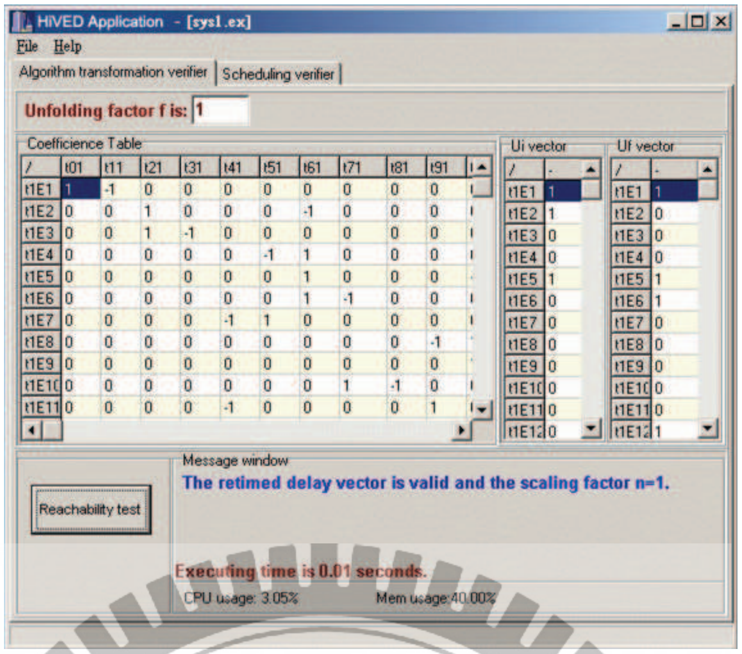
3.4 Implementation and Result

In order to prove our methodology, we develop verification software HiVED (High Level Verification Engine for DSP) in this study. The software application is implemented by using Borland C++ Builder on Microsoft Windows XP. In static phase, as shown in Figure. 3.10(a), the inputs to HiVED are PN characteristic matrix of two-order IIR filter in Figure. 2.3(a), the initial and retimed vectors. In a few time, HiVED will report the retimed vector is correct or not. In dynamic phase, Figure. 3.10(b), the inputs to HiVED are PN characteristic matrix and scheduling schemes from various schedule algorithms. In the center of HiVED workplace, it shows the schedule from input file. After push the aided buttons, HiVED will show the result step by step animatedly.

3.5 Summary

In this chapter, we explored the modeling and formal verification of dataflow graph in system level design using PN model. The verification flow includes two phases, static and dynamical. In static phase, proposed methodology checks system architecture design using PN characteristic matrix equations. In dynamical phase, a FSFG schedule also can be verified by checking reachability of PN firing sequence.

The main contribution of our work is to put formal verification beyond RTL level by using PN model. PN theory also provides analysis methods that FSFG can not offer. The implementation software HiVED also proves our verification methodology. We believe to have contributed fostering the evidence that high-level verification to be widely used in DSP system design beyond RTL level.


HiVED Application - [sys1.ex]
File
Help
Algorithm transformation verifier
Scheduling verifier
Reachability test

Unfolding factor f is: 1

Coefficience Table

/	t01	t11	t21	t31	t41	t51	t61	t71	t81	t91
t1E1	1	-1	0	0	0	0	0	0	0	0
t1E2	0	0	1	0	0	0	-1	0	0	0
t1E3	0	0	1	-1	0	0	0	0	0	0
t1E4	0	0	0	0	0	-1	1	0	0	0
t1E5	0	0	0	0	0	0	1	0	0	0
t1E6	0	0	0	0	0	0	1	-1	0	0
t1E7	0	0	0	0	-1	1	0	0	0	0
t1E8	0	0	0	0	0	0	0	0	0	-1
t1E9	0	0	0	0	0	0	0	0	0	0
t1E1(	0	0	0	0	0	0	0	1	-1	0
t1E11	0	0	0	0	-1	0	0	0	0	1

Ui vector

/	-
t1E1	1
t1E2	1
t1E3	0
t1E4	0
t1E5	1
t1E6	0
t1E7	0
t1E8	0
t1E9	0
t1E1(	0
t1E11	0
t1E12	0

Uf vector

/	-
t1E1	1
t1E2	0
t1E3	0
t1E4	0
t1E5	1
t1E6	1
t1E7	0
t1E8	0
t1E9	0
t1E1(	0
t1E11	0
t1E12	1

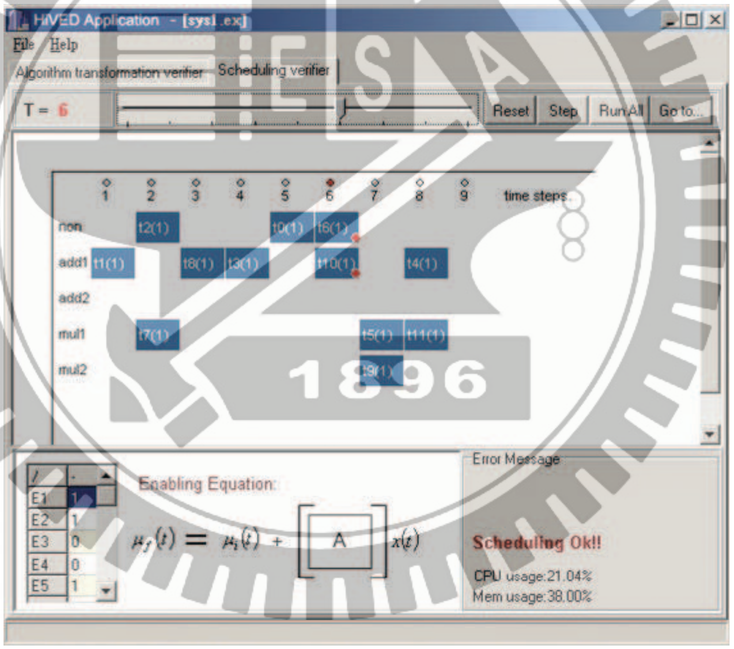
Message window

The retimed delay vector is valid and the scaling factor n=1.

Executing time is 0.01 seconds.

CPU usage: 3.05% Mem usage: 40.00%

(a)



T = 6

/	-
E1	1
E2	1
E3	0
E4	0
E5	1

Enabling Equation:

$$\mu_f(t) = \mu_i(t) + \left[\boxed{A}\right] x(t)$$

Error Message

Scheduling Ok!!

CPU usage:21.04%
Mem usage:38.00%

(b)

Figure 3.10: (a) Static verification, and (b) dynamic verification of second-order IIR filter by using proposed HiVED verification software.

CHAPTER 4

On Verification on Dataflow Scheduling

4.1 Introduction

This work presents a hybrid verification algorithm to verify high-level synthesis (HLS) results of dataflow algorithms. Typically, given a dataflow graph (DFG) or a DSP (Digital Signal Processing) design and a set of design constraints, the HLS aims to generate tasks schedules with processor resources assignment. The HLS performs high-level algorithmic transformations including retiming, scaling, and unfolding techniques on the DFG to meet the architectural constraints and then allocates processor resources accordingly [61, 74, 77–79]. In general, most solutions to the scheduling problem can be found by heuristic and Integer Linear Programming (ILP) [80,81]. Heuristic method finds good solutions for large problems quickly but suffers with tightly constrained problems where early pruning decisions exclude candidates leading to superior solutions. On the other hand, the theoretical framework of ILP based method commonly uses several ILP mapping techniques with cost functions as model of constrained-based schedule. The cost functions may combine several performance measurements such as Iteration Period Bound (IPB), Periodic Delay Bound (PDB) and Processor Bound (PB), which reflect the absolute limits on computation rate, latency and area of hardware implementation [59, 60, 77]. ILP method exactly solves scheduling but has difficulties with time complexity and constraint formulation. Heuristic and ILP scheduling methods produce a single schedule at a time. In order to find optimal one, scheduling algorithms may be applied iteratively. The overall error-prone refining process and the complexity increasing with more constraints added to problem formulations make scheduling algorithms difficult to solve. Herein, any mistake or incomplete description made in the scheduling procedures may lead an illegal solution and defeat following synthesis results. In our opinion, introducing high-level verification in system design flow may take benefit to speed up the scheduling procedure by filtering out invalid scheduling and prevent from scheduling faults. Therefore, this work intends to present a formal verification method to unveil the faults produced in HLS.

The proposed verification method is two-folded, and the verifier utilizes both techniques including Petri Net theory and SMV model checker in it. In the first folded, a high-level dataflow design is converted into a Petri Net model. Petri Net model can hold data dependence of dataflow design, therefore, any legal high-level algorithmic transformation has to conform to the firing rules of the Petri Net model. In the second folded, SMV model checker is used to check the correctness of a Finite State Machine (FSM) for the data-path scheduling. An admissible FSM schedule must satisfy system specification of the dataflow design. In the proposed verifier, given a DUV (design-under-verification), two inputs including *system description* and *tasks schedule* are required. The system description is basically a Fully-Specified Signal Flow Graph (FSFG) [50]. The FSFG represents the behavioral of the dataflow algorithm which is also a design entry of HLS. In order to meet architectural constraints, high-level algorithmic transformations normally reconstruct the original FSFG design and find the optimal tasks schedule. To verify the correctness of the algorithmic transformations, the reconstructed FSFG and its original FSFG design are converted into Petri Net domain. In PN domain, each Petri Net graph can be represented by a PN characteristic matrix. Two PN characteristic matrices including the reconstructed FSFG design and its original FSFG graph must satisfy PN characteristic matrix equation. By using two proposed traverse algorithms, the verifier tries to find the candidate reconstructed FSFGs from PN reachability tree. Each candidate reconstructed FSFG can be seen as a high-level algorithmic transformed design which is corresponded to its original FSFG design. All the relationships of the data dependence between each operation-pair of the reconstructed FSFG graph can be seen as the system specifications of all the composed operators of the FSFG graph. The system specifications are classified into three classes including the non-preemption, job completion and precedence properties. Each legal executing sequence of the FSFG graph must satisfy those system specifications. Another input to the verifier is the tasks schedule expressed in the format of processor-time chart (or $P \times T$ chart). The $P \times T$ chart equally shows the executing sequence of all tasks of dataflow algorithms. In order to verify the tasks schedule, $P \times T$ chart is re-represented by a FSM description. Then, the SMV model checker is used to verify the FSM machine which must satisfy the system specifications.

The remainder of this work is organized as follows. Section 4.2 introduces some backgrounds of the state-of-art formal verification techniques. Section 4.4 describes the admissible conditions for FSFG schedule in HLS. The proposed high-level verification flow is presented in Section 4.5. The proposed two-stage verification algorithm is discussed in Section 4.6. In Section 4.7, we discuss the complexity analysis of the verification algorithms. In section 4.8 some experimental results are given. Section 4.9 gives the summary of this work.

4.2 Background

Formal verification gains the confidence from mathematical reasoning in the correctness of various aspects of the complex system design. The trends of formal method can be roughly divided into two mainstreams: *equivalence checking* [7] and *model checking* [10]. This section introduces the background of both techniques for the formal verification.

4.2.1 Equivalence Checking

The purpose of the equivalence checking is to prove functional equivalence of two design representations modeled at the same or different levels of abstraction. Generally, two Boolean reasoning techniques including BDD (Binary Decision Diagrams) [19–21] and Boolean SAT (Satisfiability) [25, 26] are used in formal equivalence checking. Both extensional techniques are applied in RTL or gate level.

4.2.2 BDD: Binary Decision Diagrams

A BDD is a data structure for representing a Boolean function. The basic idea from which the data structure was created is the Shannon expansion. And the full potential for efficient algorithms based on the data structure is investigated by Bryant [19–21]. Conceptually, a Boolean function can be constructed and represented by a BDD tree as follows. First, a Boolean function is split into two sub-functions (cofactors) or if-then-else normal form by assigning one variable. Each sub-functions is split iteratively along any path from root to leaf, no variable appears more than once, and the variables always appear in the same order. Next, apply the following two reduction rules: merge any duplicate nodes, and delete one of the child pointers of a node point to the same descendant. A BDD example of Boolean function ($x \oplus y \oplus z$) is given by Figure. 4.1(a). After applying reduction rules of Figure. 4.1(b), the resulting direct, acyclic graph is the ordered redundant BDD in Figure. 4.1(c). In practical, BDDs are generated and manipulated in the full reduced form without ever building the decision tree. Bryant [20] also provides a detailed exposition on BDDs and surveys some applications and variables.

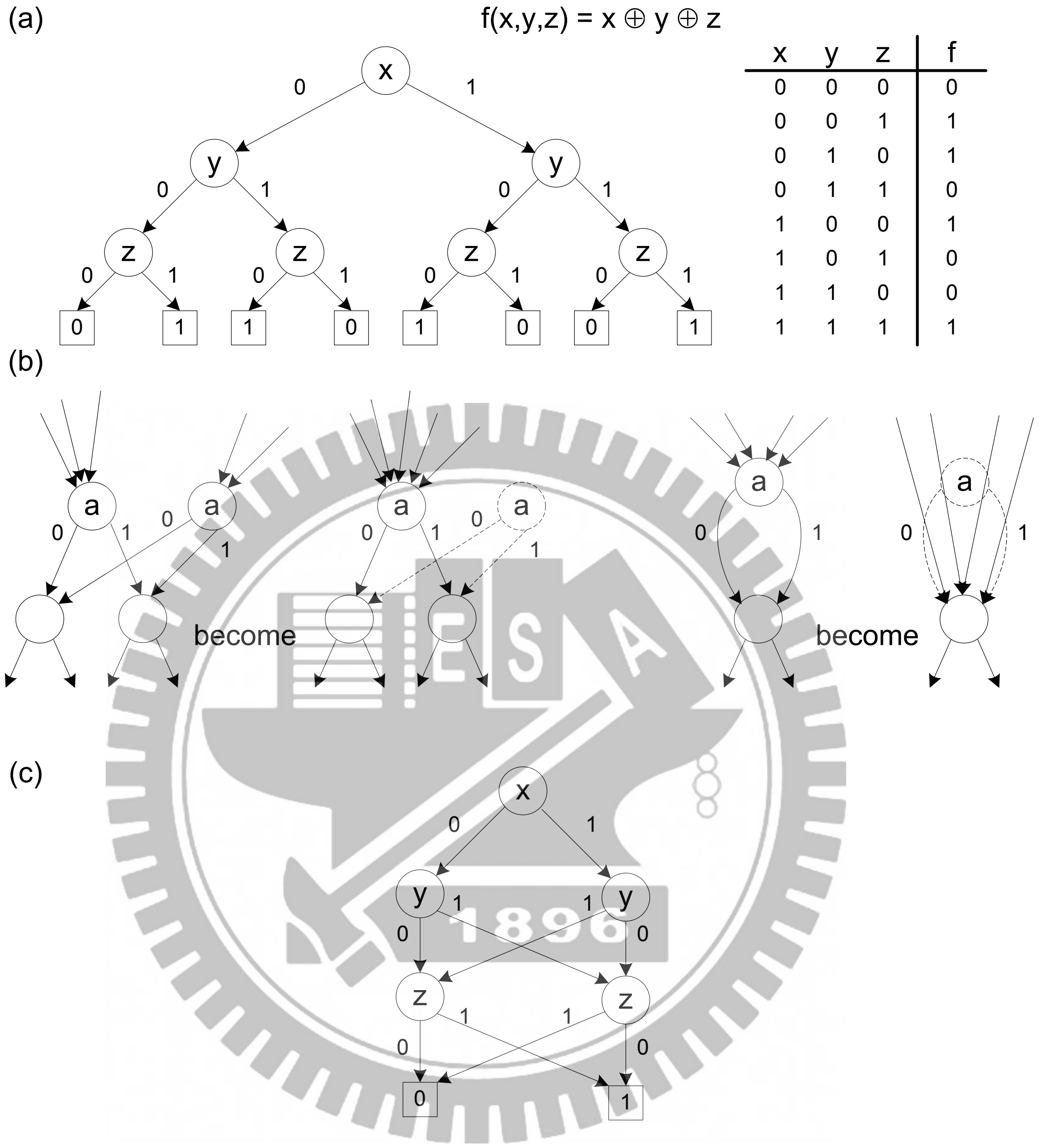
4.2.3 SAT: Boolean Satisfiability

Boolean Satisfiability (SAT) [25–27] is the decision problem of determining whether the variables of a given Boolean function can be assigned in which to make the function evaluate to Boolean *true*. The SAT problem is known to be NP-complete [82]. However, in practical, there has been tremendous progress in SAT solvers technique over the years [83].

Most SAT solvers use a Conjunctive Normal Form (CNF) representation of the Boolean formula. In CNF, the formula is represented as a conjunctive of clauses, each clause is a disjunction of literals, and a literal is a variable or its negation. Note that the CNF formula is satisfied if and only if all clauses are satisfied, i.e. all clauses must evaluate to Boolean *true*. In practical application, a Boolean circuit may be encoded as a satisfiability equivalent CNF formula using the method of [84]. And then, the existential SAT solver, zChaff [85] solver for instance, may work directly on the Boolean circuit representation.

4.3 Model Checking

Over the last decades, model checking has been widely used for verification of FSM (Finite-State Machine) system, such as hardware design and communication



x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 4.1: (a) A example BDD of Boolean function ($x \oplus y \oplus z$), (b) BDD reduction rules, and (c) final ordered redundant BDD graph.

protocols. The verification process is to check whether a given implementation system satisfies a given specification. The specification of the system is formulated as a temporal logic formula, while the implementation is described as a FSM system. When surveying the kernel algorithms of model checking, some papers utilize Symbolic Model Checking [10, 11, 29, 86] that represents state space symbolically using Binary Decision Diagrams (BDD). And most literatures use SAT based model checking and perform the graph search to improve model checking performance [25, 27, 30, 87].

Formally, given a desired property, expressed as a temporal logic formula φ, and a model M with initial state s, the model checking problem can be stated that decide if $M, s \models \varphi$. In generally, φ is usually described by using Computation Tree Logic (CTL) [88], which is a branching temporal logic for describing properties of the transition system. In Section 4.3.1, some definitions and properties of CTL formula are introduced as following.

4.3.1 CTL: Computation Tree Logic

The Computation Tree Logic (CTL) [88] is a branching temporal logic for describing properties of the transition systems. CTL formula is defined over an alphabet set AP of atomic propositions. CTL formulas are constructed from two path quantifiers and several temporal operators. The path quantifiers include: A, meaning "for all transition paths", and E, meaning "there exists a path". The temporal operators include: X meaning "next time", F meaning "eventually in the future", G meaning "globally in the future", and U meaning "until". Let AP be a set of atomic propositions, a CTL formula over AP is defined as one of the following:

- **true**, **false**, φ, or $\neg\varphi$, for $\varphi \in AP$.

- $\varphi \wedge \psi$, $\varphi \vee \psi$, or $\varphi \rightarrow \psi$, where φ and ψ are CTL formulas.
- $EX\varphi$, $EF\varphi$, $AX\varphi$, $AF\varphi$, or $AG\varphi$, where φ is a CTL formula.
- $E\varphi U\psi$, or $A\varphi U\psi$, where φ and ψ are CTL formulas.
- (φ), where φ is a CTL formula.

4.4 Admissible Conditions for Scheduling

In HLS, designers may apply high-level transformation techniques on their original FSFG design and obtain the desired optimal or suboptimal schedule from its restructured FSFG. For all operation nodes in a FSFG design, schedule determines the start time of each task. The executing order of all tasks in the schedule must satisfy the system specification, such as job completion, precedence and non-preemptive conditions. In the following sections, we will address these conditions. The false cause and the false detection for the schedule using CTL (computational tree logic) formulas are also discussed.

4.4.0.1 Job Completion Condition

Let f be the unfolding factor of the design. An admissible schedule must ensure that each operation node in vertex set V of the FSFG is scheduled at exact once. Thus, during the period of the ith-iteration of S, it must ensure that each operation node in vertex set V of the FSFG must be scheduled at exact f times during the length of S, that is:

$$t_j^i > 0, t_j^i = \varphi(op_j^i), 1 \leq i \leq f, \forall op_j^i \in V. \tag{4.1}$$

Job completion fault occurs while one or more operation nodes are not scheduled in S. For example, the start time of operation node op_j^i is $t_j^i = 0$. Let p be the atomic proposition that op_j^i executes at the c-th step of schedule S. The CTL formula of job completion property is represented by

$$EF(p) \,\&\, (p \rightarrow EF(\neg p)) \tag{4.2}$$

4.4.0.2 Precedence Condition

Let DAG graph $G_s(V, E)$ be the scheduled sequencing graph of the given schedule S, where node set V represents operation nodes, and edge set E describes dependencies between the nodes. For each edge $e(op_j^i, op_k^i) \in E$, the precedence property ensures that operation op_j^i should be completed before operation op_k^i can start, that is:

$$t_k^i \geq t_j^i + d_j^i. \tag{4.3}$$

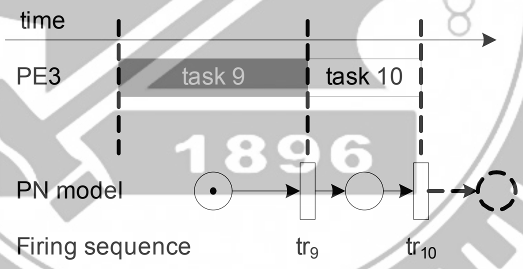


Figure 4.2: Precedence property of operation nodes

In Figure. 4.2, for instance, operator v_1 is a successor of operator v_4, thus, the execution order of these two operation nodes must ensure that $op_4 \rightarrow op_1$. Schedule S violates precedence property if these two operation nodes execute in reverse order. Let p and q be the atomic propositions that two operators op_j

and op_k execute at some steps of schedule S respectively. And operator op_j is precedent to op_k. The CTL logic of precedence property is represented by

$$AG((p)\&(\neg q) \rightarrow AF((\neg p)\&(q))) \tag{4.4}$$

4.4.0.3 Non-Preemption Condition

An admissible schedule must ensure that a computation is not preempted by another that is scheduled on the same processor at the same time. On the other word, if the deterministic busy time of a single task op_j^i in the i-th iteration is d_j^i, then for each time unit during its busy period, the same processor pe_j^i must execute that task, such that:

$$PE_r(u) = pe_j^i,\ pe_j^i = \tau(op_j^i),\ t_j^i \leq u < t_j^i + d_j^i. \tag{4.5}$$

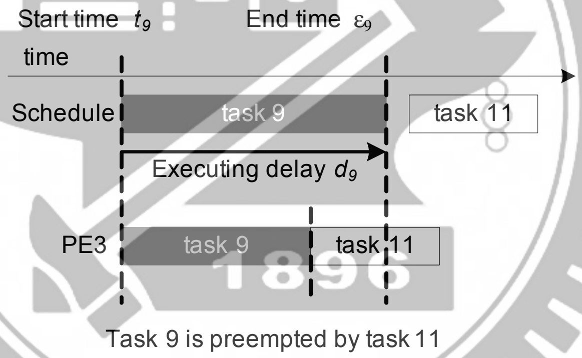


Figure 4.3: Task 9 is preempted by operation task 11

Where $PE_r(u)$ is the assignment function for resource r, $0 < u \leq le(S)$. In Figure. 4.3, for instance, the executing delay of task 9 is d_9. During executing interval d_9, task 9 is preempted by operation task 11. An admissible schedule must avoid such preemptive execution. Let p be the atomic proposition that task

op_j^i executes during the period $t_j^i \leq u < t_j^i + d_j^i$ using PE_r. And q be the atomic proposition that each of the other task op_k^i uses the same resource PE_r during the period u. The CTL logic of non-preemptive property is represented by

$$AG((p) \,\&\, (\neg q)) \tag{4.6}$$

4.5 Proposed verification flow for HLS

4.5.1 Verification Flow

Figure. 4.4 shows the flowchart to illustrate our verification method. The inputs to the flow include a given schedule and the original FSFG graph. Before performing two-stage verification method, the preprocessing on each input is applied separately. First, given schedule is the DUV (design under verification) that needs to be verified. In system-level design flow, designers may use unfolding algorithm to pursue perfect FSFG achieving MASP on their original FSFG design. Usually, the FSFG of the DSP algorithm describes one iteration of the computation. By applying unfolding algorithm on the FSFG is to unfold the original FSFG by a factor f which implies f consecutive iterations of the design. In contrast, we perform *unfolding checking* in our verification flow to detect the *unfolding factor* f from given schedule. Another input to the flow is the original FSFG graph. It is transformed into a PN model by using proposed transformation.

The delay elements of the original FSFG design can be seen as the initial marking state of its transformed PN model. In PN domain, each marking state reached from the initial marking is a retimed version of its original design. Some of these markings, called the candidate markings, may be the correct restructured FSFGs for the given DUV. They can be found by using *initial tasks* obtained from the given schedule. After preprocessing, two-stage verification method is applied continuously.

At the first stage of the flow, we build the reachability tree rooted by initial marking. Let m be one of the marking in the tree. We use vector κ, which includes all *initial tasks* of the given schedule, to be the firing vector of PN. Marking m is said to be a *candidate marking* if and only if its result marking m', which is obtained by taking m and κ into PN matrix equation 2.8, is valid. A valid marking, m', also means that each element in m' is a non-negative integer. Two Breadth-First traverse algorithms are applied to find all candidate markings from the reachability tree at this stage. If there is no candidate marking found, the erroneous message is reported, since there does not exist any candidate marking leading all *initial tasks* valid. On the other hand, if there exists candidate marking, the flow continues verifying the schedule by using model checker.

At the second stage, we verify the DUV schedule by using SMV model checker. The inputs to the model checker include the behavioral description of FSM and the set of CTL formulas. FSM is directly obtained from given schedule. CTL formulas, which contain job-completion, precedence and non-preemptive properties, are generated according to the retimed FSFG corresponded to each candidate marking. If the FSM model satisfies all the CTL formulas, we say that the candidate marking is satisfied. Given schedule is said to be correct if and only if all the candidate markings are satisfied. If one or more than one of the candidate markings violates its CTL formulas, the erroneous message is reported and the counterexample of the schedule is given by the model checker. On the other hand, if all candidate markings satisfy all the CTL formulas, we say that given schedule is valid.

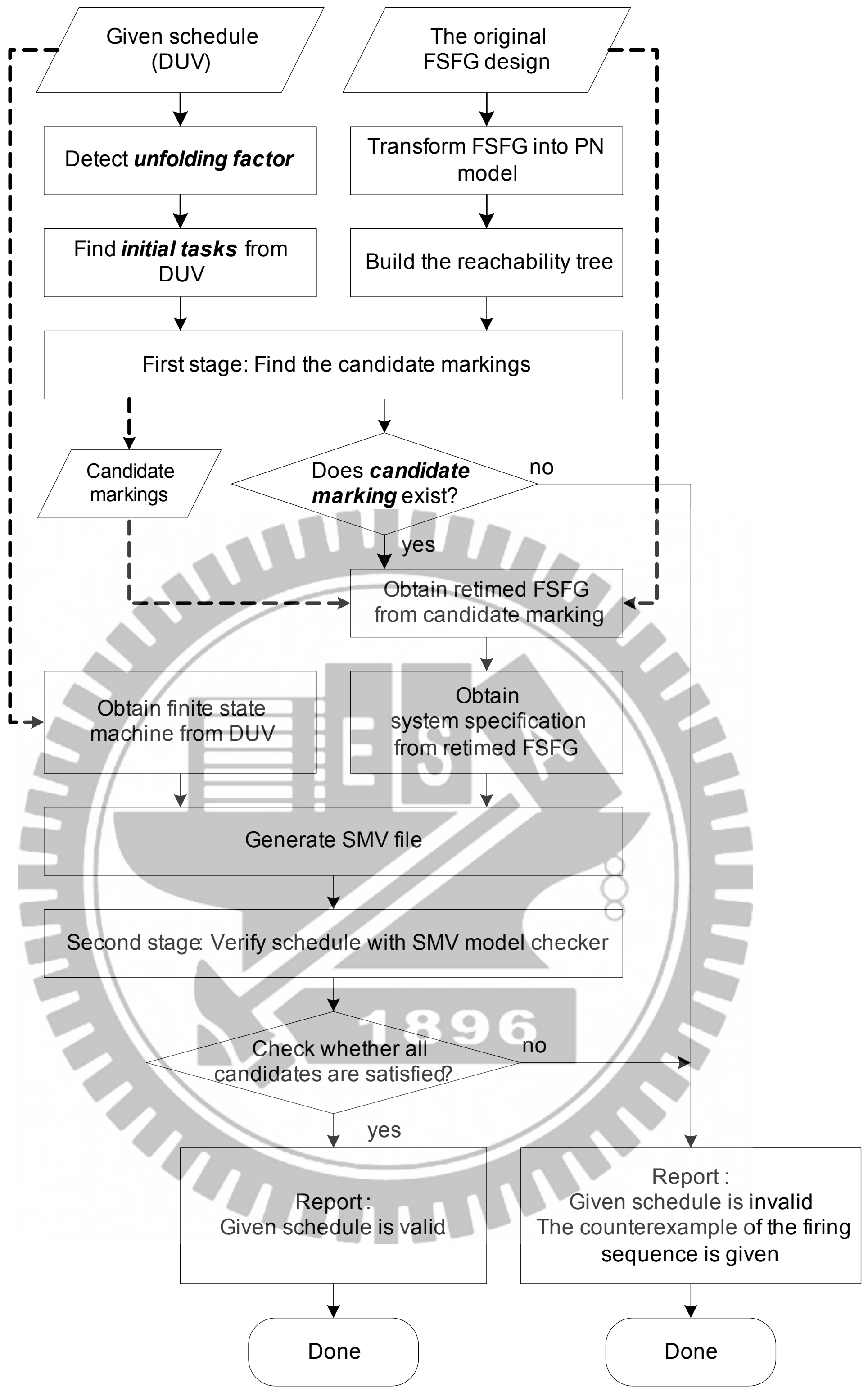

Given schedule (DUV)
The original FSFG design
Detect *unfolding factor*
Transform FSFG into PN model
Find *initial tasks* from DUV
Build the reachability tree
First stage: Find the candidate markings
Candidate markings
Does *candidate marking* exist?
no
yes
Obtain retimed FSFG from candidate marking
Obtain finite state machine from DUV
Obtain system specification from retimed FSFG
Generate SMV file
Second stage: Verify schedule with SMV model checker
Check whether all candidates are satisfied?
no
yes
Report: Given schedule is valid
Report: Given schedule is invalid The counterexample of the firing sequence is given
Done
Done

Figure 4.4: Flowchart for the proposed high-level verification method

4.5.2 Relation Between Marking Sets

Since the nodes of reachability tree are exponential growth with the height of the tree, the policy to shorten the searching space is to find candidate markings to reduce the searching space at the first stage . And then, it verifies the schedule by checking the correctness of the candidate retimed FSFG at the second stage.

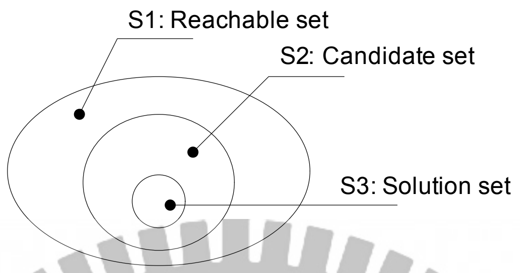


Figure 4.5: The relation between reachable, candidate and solution marking sets

Assuming there are n operations in a given FSFG and n transitions in the correspond PN model. Let f be the *unfolding factor* of a given schedule. At the first stage, the algorithm tries to find the candidate marking set from the reachable marking set in reachability tree and fires each transition once each time. The height of each node in the reachability tree is the distance from the root node to itself. Since, during one iteration period of the schedule S, $le(S)$, each scheduled task must be fired one time, the height can also be seen as the number of transitions that have been fired from the root node. Thus, for an n-tasks schedule, the *upper height-bound* of the reachability tree is bounded by $H_{up} = f \times n$. At the second stage, it continually finds the solution marking set from the candidate marking set. The set relation between three marking sets is shown in Figure. 4.5, that is $S3 \subseteq S2 \subseteq S1$. The purpose of the first stage is trying to reduce the searching space from reachable marking set $S1$ to candidate marking set $S2$, while the second stage is trying to find solution marking set $S3$

from candidate marking set $S2$.

4.6 Verification Algorithms

4.6.1 First stage: Two approaches for candidate search

At first stage, two approaches including the early-terminated and the optimal methods are proposed. We will discuss in the following sections.

4.6.1.1 The early-terminated approach

The first approach is the early-terminated traverse method. Before introducing early-terminated traverse algorithm, we first consider Lemma 4.1.

Lemma 4.1 *Let T_{tree} be a reachability tree which is bounded by upper height-bound H_{up} and m_1 be any one of the candidate markings in T_{tree}. For any other candidate marking m_2 in the successor path of marking m_1, m_2 is in the solution marking set $S3$ if and only if m_1 is in $S3$.*

Proof: Let etf_set be the earliest task-finished set of a given schedule and transition sequence σ_1 be a firing sequence that leads $m_1 \in S2$ from root marking m_r of T_{tree} to be a candidate marking, that is $m_r \xrightarrow{\sigma_1} m_1$. Assuming there exists another candidate marking $m_2 \in S3$, $m_2 \neq m_1$, with firing sequence σ_2 that leads m_2 from root marking m_r of T_{tree} to be a candidate marking, that is $m_r \xrightarrow{\sigma_2} m_2$, and is in the successor path of marking m_1.

As defined in Definition 1, it must be satisfied that $etf\_set \subseteq \sigma_1$ and $etf\_set \subseteq \sigma_2$ where the elements of σ_1 and σ_2 are all in $\{nop\} \cup \{etf\_set\}$. As described in assumption, m_2 is in the successor path of marking m_1, it is still satisfied that

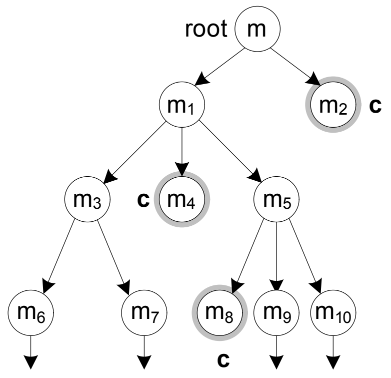


Figure 4.6: The traverse order of early-terminated approach

$\sigma_2 = \sigma_1 \cup \{nop\}$. This implies m_2 is in solution marking set $S3$ if and only if m_1 is in $S3$. ■

The early-terminated approach tries to minimize the size of candidate set $S2$ from reachable set $S1$. When an enqueued unvisited marking is candidate, the early-terminated algorithm ignores the candidate marking and marks it as a visited node. Then, it proceeds other unvisited nodes in queue Q until all the markings have been visited. In Figure. 4.6, as an example, the traverse order of the early-terminated approach is $m, m_1, m_2, m_3, \ldots, m_{10}$. The pseudo-code of the earliest-terminated traverse method is shown in Figure. 4.7. In lines 12 to 14, it ignores the candidate marking and proceeds other unvisited nodes.

4.6.1.2 The optimal approach

The second approach to verify a schedule is the optimal approach which is improved from the early-terminated approach. In order to reduce reachable marking set $S1$, it tries to merge the redundant nodes when it proceeds Breadth-First traverse.

Let m be an unvisited node to be processed. If m is a candidate marking, it ignores this node by using Lemma 4.1 and proceeds other unvisited nodes in

```
procedure BFS_BUILD_TREE_EARLY_TERMINATED($m_0$)
    Initialize Queue structure $Q$
    Allocate new node $nn$                                   ▷ initialize root node
    $nn.visited \leftarrow false$
    $nn.marking \leftarrow m_0$
    $nn.height \leftarrow 0$
    $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
    $Q$.ENQUEUE($nn$)

    for all unvisited node $n \in Q$ do
        $n.visit \leftarrow true$
        if $n.candidate = true$ then
            continue to the next node                        ▷ Lemma 4.1
        end if
        if $n.height \leqslant H$ then
            $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($n.marking$)
            for all transition $tr \in ebl\_set$ do
                $m \leftarrow n.marking$
                $\kappa \leftarrow$ MAKE_FIRING_VECTOR_FROM($tr$)
                $m\prime \leftarrow m + [A] \cdot \kappa$              ▷ Eq.2.8
                Allocate new node $nn$
                $nn.marking \leftarrow m\prime$
                $nn.visited \leftarrow false$
                $nn.height \leftarrow n.height + 1$
                $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
                CREATE_BRANCH($n, nn$)
                $Q$.ENQUEUE($nn$)
            end for
        end if
    end for
end procedure
```

Figure 4.7: The early-terminated approach

queue. If m is not a candidate marking, it finds enabled set of transitions and creates new node on each enabled transition. For each new produced node with marking $m\prime$, if there exists another node in the reachability tree, and has the same marking associated with it, then the node with marking $m\prime$ is a duplicate node. Since, the marking $m\prime$ has appeared in the tree, this new produced node is redundant. Then, it merges this redundant node to the existential node and creates transition link from marking m to the existential node. As an example in Figure. 4.8, when it proceeds marking m_5, it founds the new created node with marking m_7 is a duplicate node. It merges these nodes and creates transition from m_5 to m_7. Then, the algorithm continually proceeds other unvisited nodes in queue.

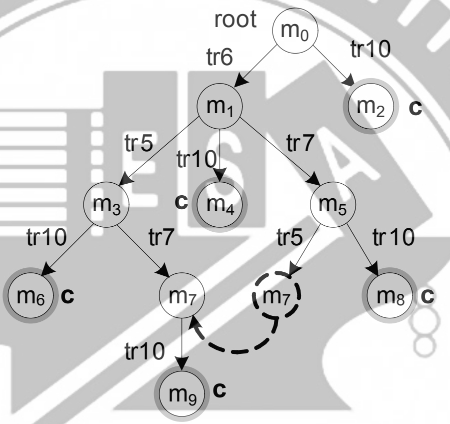


Figure 4.8: Merge the redundant nodes in optimal approach

The pseudo-code of the optimal approach is shown in Figure. 4.9. In lines 12 to 14, if the node n is a candidate marking, then it ignores this node by using Lemma 4.1 and proceeds the other nodes in queue Q. In lines 26 to 32, function $find\_dual\_node$ checks whether the new created node nn is duplicate. If there exists a duplicate node, it ether returns the dual node to $dual\_node$ or returns $null$. It continually proceeds other unvisited nodes until all nodes are visited.

```
1: procedure BFS_BUILD_TREE_OPTIMAL($m_0$)
2:     Initialize Queue structure $Q$
3:     Allocate new node $nn$                                ▷ initialize root node
4:     $nn.visited \leftarrow false$
5:     $nn.marking \leftarrow m_0$
6:     $nn.height \leftarrow 0$
7:     $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
8:     $Q$.ENQUEUE($nn$)
9:
10:    for all unvisited node $n \in Q$ do
11:        $n.visit \leftarrow true$
12:        if $n.candidate = true$ then
13:            continue to the next node                     ▷ Lemma 4.1
14:        end if
15:        if $n.height \leqslant H$ then                    ▷ max. height $H$
16:            $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($n.marking$)
17:            for all transition $tr \in ebl\_set$ do
18:                $m \leftarrow n.marking$
19:                $\kappa \leftarrow$ MAKE_FIRING_VECTOR_FROM($tr$)
20:                $m\prime \leftarrow m + [A] \cdot \kappa$    ▷ Eq.2.8
21:                Allocate new node $nn$
22:                $nn.marking \leftarrow m\prime$
23:                $nn.visited \leftarrow false$
24:                $nn.height \leftarrow n.height + 1$
25:                $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
26:                $dual\_node \leftarrow$ FIND_DUAL_NODE($nn$)
27:                if $dual\_node \neq null$ then
28:                    CREATE_BRANCH($n, dual\_node$)
29:                else
30:                    CREATE_BRANCH($n, nn$)
31:                    $Q$.ENQUEUE($nn$)
32:                end if
33:            end for
34:        end if
35:    end for
36: end procedure
```

Figure 4.9: The optimal traverse approach

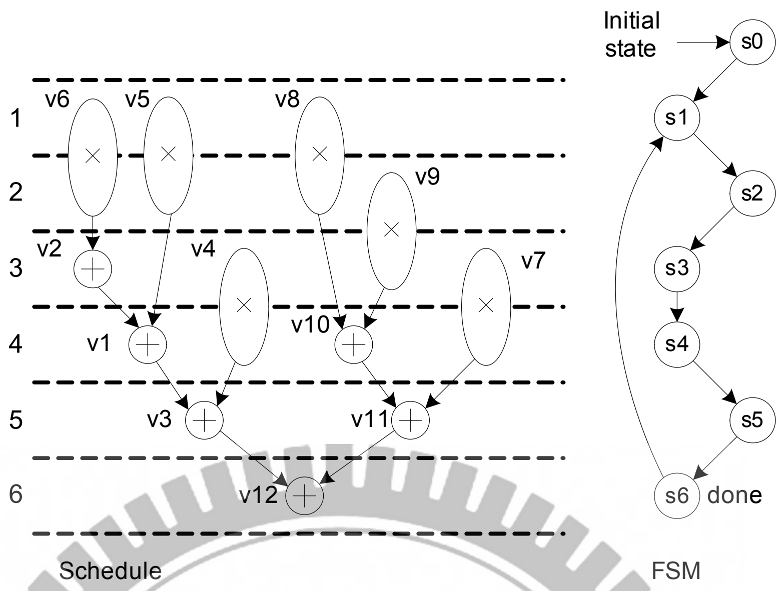
4.6.2 Second stage: The model-checking approach

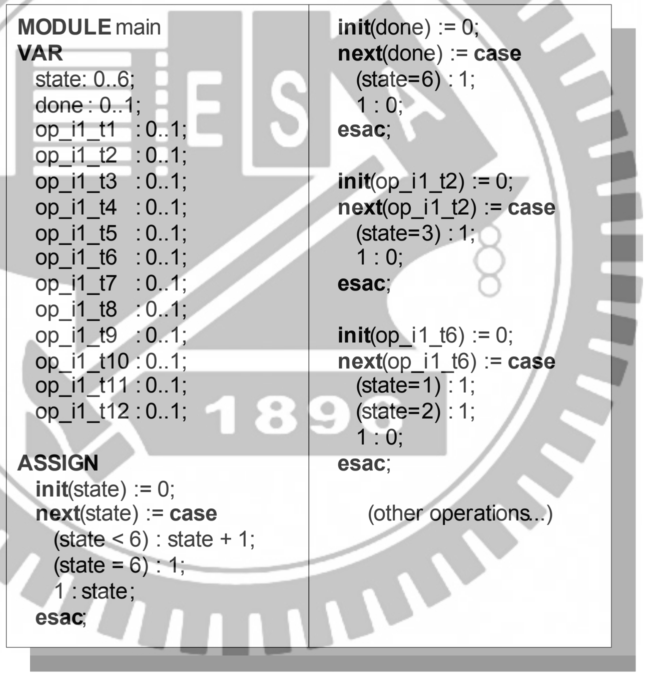
At the second stage, we verify the given DUV schedule on all candidate markings aided by using SMV model checker. The inputs to the model checker are a FSM and a set of system specification that needs to be verified. FSM of the DUV is automatically generated by the verification flow. Each candidate marking obtained from the first stage can be seen a restructured FSFG. System specification, CTL formulas, to the DUV is generated from each restructured FSFG.

Schedule S can be converted into a Moore machine. A Moore machine is defined as a 6-tuple $M(W, \Sigma, \Delta, \delta, \lambda, w_0)$, where W, Σ and Δ are the finite set of state, input signal and output signal with respect. Transition function $\delta : W \times \Sigma \rightarrow W$ is a mapping function from state and an input to the next state. Output function $\lambda : W \rightarrow \Delta$ is a mapping function from each state to the output signal. And, w_0 is an initial state. Figure. 4.10 is an DUV schedule of a third-order IIR filter. The length of schedule S is $le(S) = 6$, and it can be seen as a FSM with $le(S) + 1$ states, where s_0 is the initial state. Each state of the FSM is actually represented as a control step in S. For each task ${op_j}^i$ in S, there is an enabled signal *op_i_j* on it to indicate whether ${op_j}^i$ is enabled in each control step of S. For example, task op_6^1 is scheduled at step 1 to step 2, and let *op_i_j* be its enabled signal. The behavioral description of FSM for task ${op_6}^1$ is described at the bottom in Figure. 4.10.

4.7 The complexity analysis

Assuming there are n operations in a given FSFG, f be the *unfolding factor* of a given schedule. Thus, the upper-height of the reachability tree of the correspond PN model is bounded by $H_{up} = f \times n$.





```
MODULE main
VAR
   state: 0..6;
   done : 0..1;
   op_i1_t1   : 0..1;
   op_i1_t2   : 0..1;
   op_i1_t3   : 0..1;
   op_i1_t4   : 0..1;
   op_i1_t5   : 0..1;
   op_i1_t6   : 0..1;
   op_i1_t7   : 0..1;
   op_i1_t8   : 0..1;
   op_i1_t9   : 0..1;
   op_i1_t10 : 0..1;
   op_i1_t11 : 0..1;
   op_i1_t12 : 0..1;

ASSIGN
   init(state) := 0;
   next(state) := case
      (state < 6) : state + 1;
      (state = 6) : 1;
      1 : state;
   esac;
```

```
init(done) := 0;
next(done) := case
   (state=6) : 1;
   1 : 0;
esac;

init(op_i1_t2) := 0;
next(op_i1_t2) := case
   (state=3) : 1;
   1 : 0;
esac;

init(op_i1_t6) := 0;
next(op_i1_t6) := case
   (state=1) : 1;
   (state=2) : 1;
   1 : 0;
esac;

   (other operations...)
```

Figure 4.10: The behavioral description of FSM for the third-order IIR filter

At the first stage, there are two approaches to perform the *Breadth-First* traverse procedure. Considering exhaustive search, each node of the reachability tree has n enabled transitions in worse case. Then, the total number of nodes is:

$$1 + n + n^2 + \ldots + n^{f \cdot n} = \left(n^{f \cdot n+1} - 1\right) / (n - 1) \tag{4.7}$$

In the first approach, the early-terminated approach, the algorithm stops traversing a node while it is a candidate. Let p, $p \leq (f \cdot n)$, be the deepest level that *Breadth-First* traverse procedure can reach. Thus, the complexity of the first approach is $O(N^p), p \leq f \cdot n$.

In the second approach, the optimal approach, the algorithm merges duplicate markings in order to reduce the reachable marking set of the reachability tree. Let $x \in \mathbb{Z} = \{1, 2, \cdots\}$ be the merging radio in the reachability tree. The complexity of the two approach is $O((N/x)^p), p \leq f \cdot n$.

Thus, the relation of the complexity between two approaches is:

$$O(N^p) > O((N/x)^p). \tag{4.8}$$

At the second stage, the verification flow performs SMV model checking to verify a given schedule by checking the precedence, job completion and non-preemptive properties on given DUV. Let Ω be the size of the FSM converted from DUV, Ψ be the size of the CTL formulas, $le(S)$ be the steps of DUV schedule and c be the number of candidate markings. The complexity of the model checker in the second stage is about $O(\Omega \times \Psi \times le(S) \times c)$, in worse case.

4.8 Experimental Results

We have implemented these two approaches in our study. Each of these approaches is applied to several dataflow algorithms. Figure 4.1 shows the statistics of these designs.

Table 4.1: The statistics of test designs

Design name	num. vertices	num. edges	num. delays	Size of PN (Place x Trans.)	Schedule length	Unfolding factor
iir2d-sch1	8	14	2	(14 x 11)	6	1
iir2d-sch2	8	14	2	(14 x 11)	4	1
iir2d-sch3	8	14	2	(14 x 11)	4	1
iir2d-sch4	8	14	2	(14 x 11)	4	1
iir3d-sch1	12	21	3	(21 x 16)	6	1
iir3d-sch2	12	21	3	(21 x 16)	6	1
p243-sch1	5	7	5	(7 x 5)	96	6
p243-sch2	5	7	5	(7 x 5)	96	6
ewf-sch1	34	47	0	(47 x 34)	40	1
ewf-sch2	34	47	0	(47 x 34)	40	1

In Table. 4.1, design *iir2d-sch1* to *iir3d-sch2* [50] are the second and third Infinite Impulse Response filters. Design *p243* [50] is a design with *unfolding factor* 6, the lengths of schedule *p243-sch1* and *p243-sch2* are both 96 steps. Design *ewf-sch1* and *ewf-sch2* are low power schedules for the Elliptic Wave Filter in [74].

Table 4.2 shows the experimental results of using two approaches. There are two columns on each approach including execution time in seconds and the resource usage in unit number of allocated nodes of the reachability tree. According to experimental results, early-terminated approach suffers the state explosion problem while it traverses the reachability tree in *iir3d-sch1* and *iir3d-sch2*. In average, optimal approach takes more benefit than early-terminated approach. According to experimental results, the optimal approach outperforms the others in terms of time and resource usage in average.

Table 4.2: The experimental results

	Early-terminated			Optimal		
Test schedule	Time (sec)	Res. usage	num. of candidates	Time (sec)	Res. usage	num. of candidates
iir2d-sch1	93.81	44972	1005	1.01	376	9
iir2d-sch2	94.27	44972	1005	1.02	376	9
iir2d-sch3	148.82	114867	635	0.84	319	8
iir2d-sch4	1610.29	49502	15486	1.85	237	21
iir3d-sch1	n/a	n/a	n/a	28.58	24676	166
iir3d-sch2	n/a	n/a	n/a	77.22	22613	335
p243-sch1	0.84	2	1	0.76	2	1
p243-sch2	0.81	2	1	0.8	2	1
ewf-sch1	0.66	2	1	0.64	2	1
ewf-sch2	0.73	2	1	0.62	2	1

4.9 Summary

This work aims to exploit formal verification techniques for high-level synthesis. In the top-down design flow, design errors should be removed as early as possible; otherwise, errors detected at the later stages will result a costly, time-consuming redesign cycles. Although formal verification for logic synthesis has been studied very extensively, little work has been done for high-level synthesis. The paper presents a verification flow that can efficiently detect the design errors from the results of high-level synthesis. As shown in the experimental results, we can apply the optimal approach for the first phase to efficiently verify complex design cases.

CHAPTER 5

Verification Method of Dataflow Algorithms in High-Level Synthesis

5.1 Introduction

The System-On-Chip (SOC) design encompasses a large design space. Typically, the designer explores the possible architectures, selecting algorithms, choosing architectural elements, and constructing candidate architectures. Designing such a complex system is hard; designing such a system that will work correctly is even harder. Design errors should be removed as early as possible; otherwise, errors detected at the later stages will result a costly and time-consuming redesign cycles. Thus, the designer should face two distinct tasks in SOC design; carrying out design process itself and establishing the correctness of a design. Design correctness is the main theme of this work. Since most of SOC design efforts are on digital signal processing or dataflow computing, this work aims on the formal verification for HLS (High-Level Synthesis) of dataflow computing.

This chapter presents a verification algorithm to verify HLS of dataflow algorithms. Given a dataflow graph (DFG) and architectural constraints, the HLS aims to generate the task schedule with processor assignment. Typically, the HLS performs algorithmic transformation, such as retiming, scaling, and unfolding, on the DFG to meet the architectural constraints, and allocates resources accordingly [61, 74, 77–79]. Both algorithmic transformation and resource allocation require complex procedures. These procedures are rather heuristic and error-prone. The integer linear programming (ILP), for instance, is one of the popular techniques applied for HLS. The success of ILP is relied on the completeness of clauses. Any mistake or incomplete description made in the ILP may result in an illegal solution and affect the correctness of following synthesis results. This work intends to present a formal verification algorithm to unveil the faults produced in HLS.

In the proposed algorithm, we employed the Petri Net model as the formal description to check the correctness of dataflow behavior. Since the Petri Net model executes the data-driven behavior, it has the nature of dataflow computing and hence a good tool for the verification of algorithmic transformations and datapath scheduling. The use of the Petri Net is two-folded. First, the Petri Net model of dataflow algorithm can hold the data dependence and hence any legal transformation has to conform to the firing rules of the Petri Net model. Secondly, the scheduling candidates are correct if and only if the initiation of each task is allowed in the Petri Net model. Comparing with the traditional model checking [10, 11, 25] techniques, the first use can provide simple but thorough model for restructured algorithms while the second use can avoid false negative problems.

The inputs to the proposed formal verification are the system description and task schedule. The system description is basically a fully-specified Signal Flow Graph (FSFG) [50]. The FSFG represents the behavioral specification of the dataflow algorithm which is also a design entry of HLS. In HLS, to meet the architectural constraints, the algorithmic transformation normally reconstructs the initial FSFG to find out optimal scheduling results. The reconstructed FSFG is admissible if and only if it is equivalent to the initial FSFG. To verify the correctness of the task schedule, the proposed algorithm first converts the initial FSFG to a Petri Net model which is expressed by Petri Net characteristic matrix, because any admissible reconstructed FSFG has to have the same characteristic matrix.

Another input is the schedule, the DUV (design under verification), generated by HLS. The schedule is expressed in the format of processor-time chart (or $P \times T$ chart). The $P \times T$ chart equally shows the firing sequence. The proposed verification uses the firing sequence to unveil the legal algorithmic transformations applied for the original FSFG. The legal algorithmic transformations will then be the candidates to trace the firing sequence of the given schedule.

Based on the inputs, the proposed verification first extracts the initial firing pattern to determine the candidate reconstructed FSFGs. The candidates will then be verified with the Petri Net model. If at least one candidate exists who can allow the firing sequence to execute legally (without against the firing rules), the HLS result is claimed as a correct solution; otherwise, the verification will show the counter example in proof of the incorrectness. In this work, we propose three approaches to realize the PN-based formal verification and conclude the best one that outperforms the others in terms of processing speed and resource usage.

The remainder of this chapter is organized as follows. Section 5.2 presents the system specifications to the FSFG design. The proposed high-level verification technique and verification algorithms are presented in section 5.3. In section 5.4, we discuss the complexity analysis of three verification algorithms and some experimental results. Section 5.6 gives the summary of this work.

5.2 System Specification

For a given DUV schedule of the FSFG design, PEs must execute their tasks under system specifications. In this section, we first describe the execution of a task in FSFG corresponded to the transitions firing in PN domain. The system specification for the given schedule is then presented later.

5.2.1 Execution of a task

For a given FSFG design, the nodes represent computational tasks, while the arcs represent data dependencies, buffering, and direction of data transfer between task nodes. A task node ***cannot*** execute until sufficient data is ready on its input arcs. When the data-amount has been reached on each of its input arcs, a task node can execute. When a task node executes, it consumes data from its input arcs, executes the task through the duration of the executing delay of that task, and it produces data onto its output arcs. The same executing process can cross-refer from FSFG to PN model. In PN domain, transitions represent task nodes and places represent arcs, while token movements between places represent the firing sequence of transitions. A transition ***cannot*** fire until sufficient tokens are ready on its input places. When each of its input places has at least one token, a transition can fire. If there exists more tokens on its input places, a transition may fire several times. When a transition fires, it consumes tokens from each of its input arcs, and it produces tokens onto its output places.

As showing in Figure. 5.1, task 9 is a task node of the schedule. Where t_9 is the start time, d_9 is the executing delay, and the finish time of this task is ε_9. The executing process of this task node can be divided into three durations: before task executing, during task executing, and after task executing. In PN domain, a task of FSFG can be transformed into a transition tr_9 of PN model. Before task executing, tr_9 may or may not enable. Then, tr_9 is enabled during task executing. At last, transition tr_9 fires instantaneously at the moment after finishing the execution of task 9. An example of FSFG schedule of the third-order IIR filter of Figure. 2.3(b) is shown in Figure. 5.2(a), and the corresponded firing transitions to the task schedule is illustrated in Figure. 5.2(b). At each step of the schedule in Figure. 5.2(c), enabled transitions may fire at the end of each step. The tasks schedule and its firing transitions are illustrated.

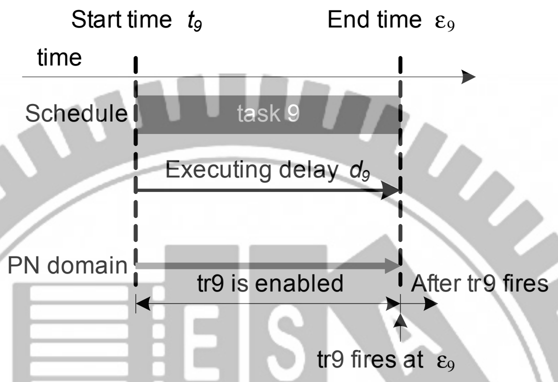
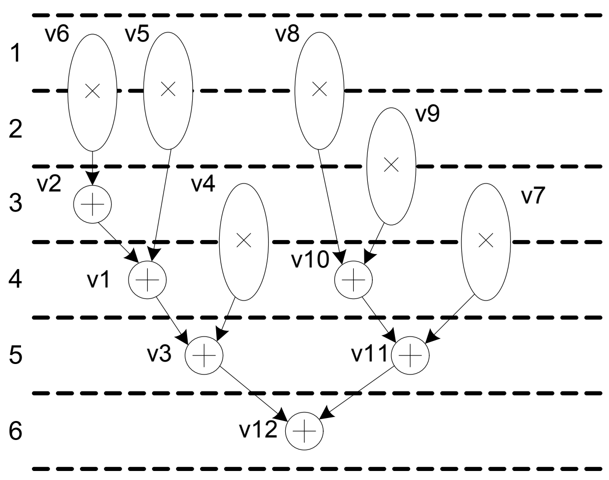


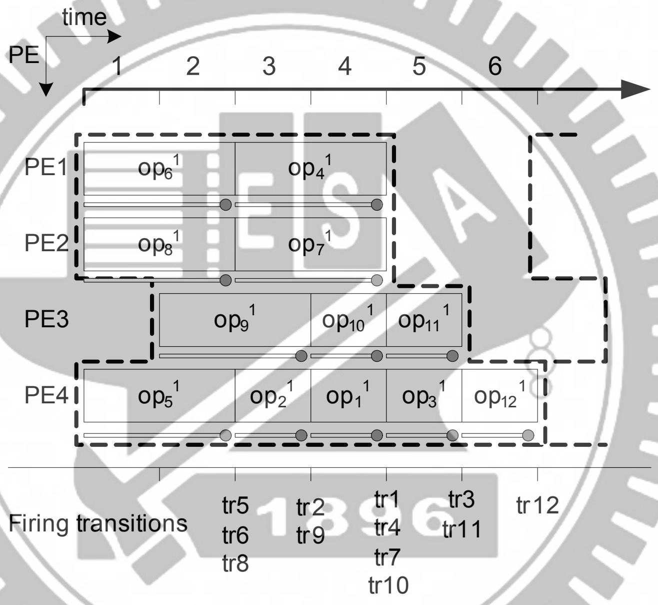
Figure 5.1: Executing duration of a task

5.2.2 System Specification

The mentioned system is the given schedule, or the DUV. In HLS, designers may apply high-level transformation techniques on their original FSFG design and obtain the desired optimal or suboptimal schedule from its restructured FSFG. For all operation nodes in a FSFG design, schedule determines the start time of each task. The executing order of all tasks in the schedule must satisfy the system specification, which can be extracted from the restructured FSFG. Let op_j^i and op_k^i be two distinguish tasks in i-th iteration of S. The properties to the system are defined as following. We also describe the false-cause and its detection



(a) A sample schedule



(b) The schedule in space-time domain

Step	Tasks schedule
1	op_5^1, op_6^1, op_8^1
2	op_9^1
3	op_2^1, op_4^1, op_7^1
4	op_1^1, op_{10}^1
5	op_3^1, op_{11}^1
6	op_{12}^1

Step	Firing transitions
1	{ non }
2	{tr5, tr6, tr8}
3	{tr2, tr9}
4	{tr1, tr4, tr7, tr10}
5	{tr3, tr11}
6	{tr12}

(c) Tasks schedule and firing transitions

Figure 5.2: A sample schedule of the third-order IIR filter in Figure. 2.3(b)

in PN domain.

1. Non-Preemption Property

   - Definition: Let op_j^i be one of the task in schedule S. An admissible schedule must ensure that a computation is not preempted by another that is scheduled on the same processor at the same time. On the other word, if the deterministic executing delay of a single task op_j^i in the i-th iteration is d_j^i, then for each time unit during its executing delay, the same processor pe_j^i must execute that task, such that:

$$PE_k(u) = pe_j^i,\ pe_j^i = \tau(op_j^i),\ t_j^i \leq u < t_j^i + d_j^i. \qquad (5.1)$$

     Where $PE_k(u)$ is the assignment function for resource k, $1 \leq u \leq le(S)$.

   - Fault-cause: In Figure. 5.3, for instance, the executing delay of task 9 is d_9. During executing delay d_9, task 9 is preempted by operation node op'. An admissible schedule must avoid such preemptive execution.

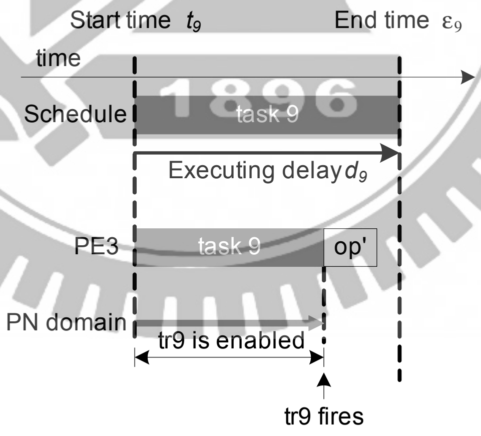


Figure 5.3: Task 9 is preempted by operation node op'

- Detection: At the s-th step of schedule length $le(S)$, the marking state μ_s to the s-th step is a $(|P| \times 1)$-column vector, which indicates the number of tokens on each places. During executing delay d_j^i of task op_j^i, the PN transition tr_j with respect to its operation node op_j^i must be enabled, that is:

$$\mu_s(p) > 0,\ \forall p \in {}^{\bullet}tr_j, 1 \leq s \leq le(S). \tag{5.2}$$

2. Job Completion Property

- Definition: Let f be the unfolding factor, which implies f consecutive iterations of the design. During one period of the ith-iteration of S, an admissible schedule must ensure that each operation node in vertex set V of the FSFG is scheduled at exact once. Thus, job completion property must ensure that each operation node in vertex set V of the FSFG must be scheduled at exact f times during the length of S, that is:

$$t_j^i > 0, t_j^i = \varphi(op_j^i), 1 \leq i \leq f, \forall op_j^i \in V. \tag{5.3}$$

- Fault-cause: Schedule S violates job completion property if one or more operation nodes are not scheduled in S. For example, the start time of operation node op_j^i is $t_j^i = 0$.

- Detection: Let f be the unfolding factor obtained from given schedule S, and s be the index of the s-th step of S. Let firing vector κ_s be a $(|T| \times 1)$-column vector. If the j-th element of the firing vector is $\kappa_s(j) = 1$, it indicates the transition tr_j, corresponded to the operator node op_j^i at ith-iteration, fires at the s-th step. An acceptable schedule must ensure that each operation node op_j^i at the ith-iteration is scheduled exact once during the schedule length $le(S)$. On the other word, transition tr_j must fire f times totally during the length $le(S)$.

3. Precedence Property

   - Definition: Let DAG graph $G_s(V, E)$ be the scheduled sequencing graph to the given schedule S, where the set of nodes V represents operation nodes, and the set of edges E describes dependencies between the nodes. For each edge $e(op_j^i, op_k^i) \in E$, the precedence property must ensure that operation op_j^i should be completed before operation op_k^i can start, that is:

$$t_k^i \geq t_j^i + d_j^i. \tag{5.4}$$

     Where $t_j^i = \varphi(op_j^i)$, $t_k^i = \varphi(op_k^i)$ are the start times of operation nodes, and d_j^i is the executing delay of op_j^i.

   - Fault-cause: In Figure. 5.4, transition $tr10$ is a successor of transition $tr9$, thus the execution order of these two operation nodes must ensure $tr9 \rightarrow tr10$. Schedule S violates precedence property if these two operation nodes execute in reverse order.

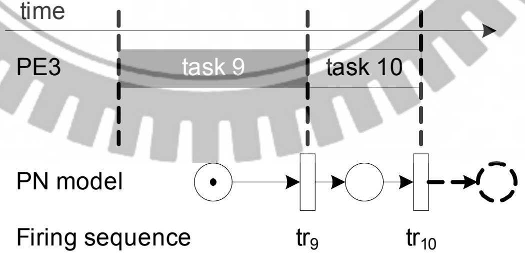


Figure 5.4: Precedence property of two operation tasks

- Detection: At the s-th step of S, the firing vector κ_s is a $(|T| \times 1)$-column vector. If the j-th element of the firing vector is $\kappa_s(j) = 1$, it indicates the operation node op_j^i fires at the s-th step. Let μ_s be the marking state at the s-th step of S, next marking μ_{s+1} can be obtained by using Equation. 5.5, that is:

$$\mu_{s+1} = \mu_s + A \cdot \kappa_s, 1 \leq s \leq le(S). \tag{5.5}$$

  Vector κ_s is said to be a valid firing if resulting marking μ_{s+1} from μ_s is valid. If all the resulting markings are valid, schedule S is satisfied under the precedence property.

5.3 High-Level Verification

In this section, proposed two-stages verification technique is introduced. The algorithms to both stages are also presented separately as the implementations.

5.3.1 Verification Flow

A flowchart illustrating our verification flow is shown in Figure. 5.5. There are two inputs to the flow: a given schedule and the original FSFG. The given schedule is the DUV (design under verification) that needs to be verified. The original FSFG reserves the characteristics of the system that the DUV must be satisfied. The proposed verification method tries to find the correct restructured FSFG, which is candidate to the DUV at the first stage, and then, it checks whether the executing sequence, the DUV, of the PN model corresponded to the candidate is satisfied at the second stage. Before introducing two-stages verification method, we address the preprocessing on both inputs separately.

One of the inputs is the given schedule. In system-level design flow, designers may use unfolding algorithm to pursue perfect FSFG achieving iteration period bound on their original FSFG design. Usually, the FSFG of the DSP algorithm describes one iteration of the computation. By applying unfolding algorithm on the FSFG is to unfold the original FSFG by a factor f which implies f consecutive iterations of the design. In contrast, we perform *unfolding checking* in our verification flow to detect the *unfolding factor* f from given schedule. Another input to the verification flow is the original FSFG graph. It is transformed into a PN model by proposed transformation rules.

In PN domain, the markings, which can be reached from the initial marking, can be seen as the retimed FSFGs of the original design. Some reachable markings are the correct restructured FSFGs for the given schedule. These markings, which dominate the correctness of the given schedule, are said to be the *candidate markings.* In order to find the candidate markings, *Breadth-First algorithm* is used to traverse all the markings of PN reachability tree at the first stage. If the candidate marking does not exist, it means the correct retimed FSFG does not exist, it reports the given schedule is not valid due to absent of the candidate marking. If the candidate marking exists, we continuously apply *Depth-First algorithm* on each candidate marking.

The given schedule is valid if there exists an initial marking, the candidate marking, of the PN model leading a firing sequence of the schedule valid. At the second state, we apply Depth-First traverse procedure on each candidate marking to check whether the given schedule is valid by checking the firing sequence of the schedule. At last, if there exists such candidate marking, the flow is done and reports given schedule is valid, or a counterexample of invalid firing sequence is reported if given schedule is invalid.

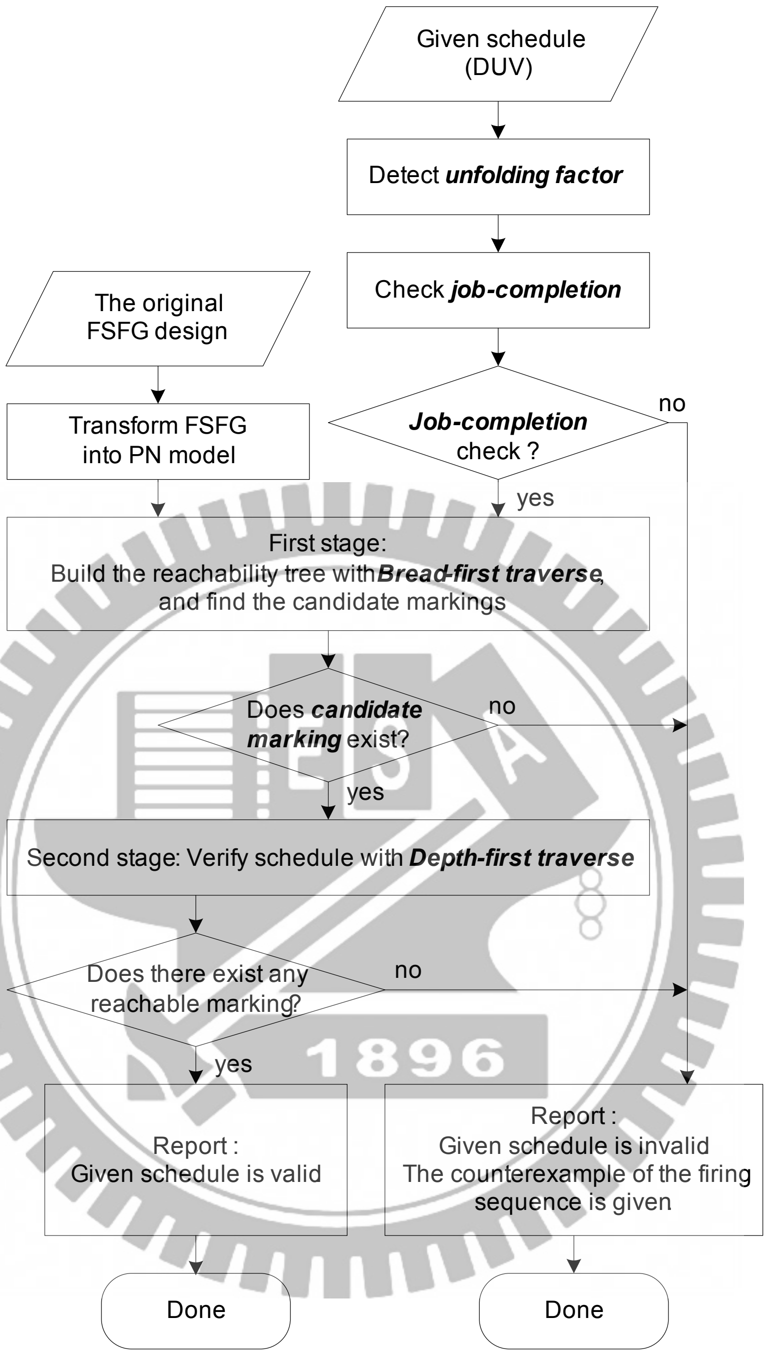

Given schedule (DUV)
Detect unfolding factor
Check job-completion
The original FSFG design
Transform FSFG into PN model
Job-completion check ?
no
yes
First stage: Build the reachability tree with Bread-first traverse, and find the candidate markings
Does candidate marking exist?
no
yes
Second stage: Verify schedule with Depth-first traverse
Does there exist any reachable marking?
no
yes
Report : Given schedule is valid
Report : Given schedule is invalid The counterexample of the firing sequence is given
Done
Done

Figure 5.5: Flowchart for the proposed high-level verification method

5.3.2 The candidate marking

The candidate marking set is a subset of the reachable marking set of a Petri net. A candidate marking is probably the correct initial marking, it also means correct retimed FSFG, which leads the firing sequence of a given schedule being valid.

Let S be a schedule of a FSFG. The earliest task-finished set etf_set of S are the tasks which are finished at the earliest task-finished step t_{etf} in S, such that

$$etf\_set = \left\{ op_j^i | \varepsilon_j^i = t_{etf},\ \forall op_j^i \in V \right\}. \tag{5.6}$$

Assuming m is a marking of the PN model. A firing sequence $\sigma : tr_1 \ldots tr_k$ of transitions is denoted by $m \xrightarrow{\sigma} m_k$, where $m_1 \ldots m_k$ are valid states, such that

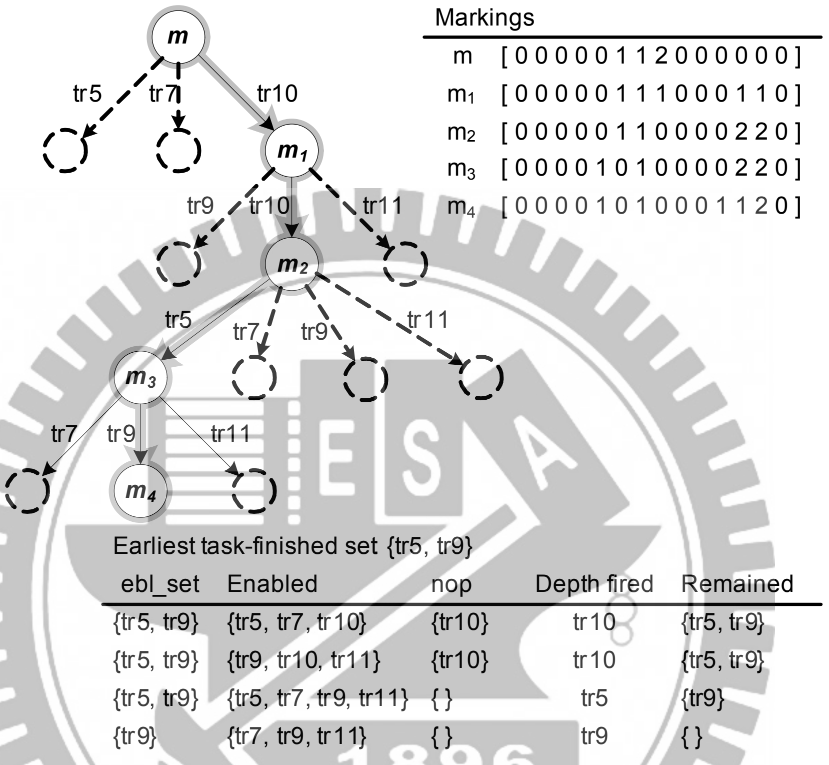
$$m \xrightarrow{tr_1} m_1 \xrightarrow{tr_2} \cdots \xrightarrow{tr_k} m_k. \tag{5.7}$$

Marking m is defined as a ***candidate marking***, if marking m and firing sequence σ satisfy Definition 1.

Definition 1 *Marking m is said to be a* ***candidate marking*** *if and only if there exists a firing sequence $\sigma : tr_1 \ldots tr_k$, such that for all tasks $op \in etf\_set$ are covered by all the transitions in σ, i.e. $etf\_set \subseteq \sigma$. And it is also satisfied that each firing transition $tr_j \in \sigma$ is either a pseudo transition, $d_j = 0$, or an earliest task-finished transition, $\varepsilon_j = t_{etf}$.*

As an example, Figure. 5.6 shows the procedure to check whether a marking is candidate. For a given schedule in Figure. 5.7, the earliest task-finished step is $t_{etf} = 1$, and the earliest task-finished set is $etf\_set = \left\{ op_j^i | \varepsilon_j^i = t_{etf} = 1 \right\} =$

$\{v_5,\ v_9\}$. Marking $m = [00000112000000]$ is said to be a ***candidate marking*** of the corresponded PN model. Since, there exists a firing sequence $\sigma : tr_{10}\ tr_{10}\ tr_5\ tr_9,\ m \xrightarrow{\sigma} m_4$, such that $etf\_set \subseteq \sigma$. The markings, $m_1 \ldots m_k$, of the firing sequence, $m \xrightarrow{tr_{10}} m_1 \xrightarrow{tr_{10}} m_2 \xrightarrow{tr_5} m_3 \xrightarrow{tr_9} m_4$, are valid states, where $m_1 = [00000111000110]$, $m_2 = [00000110000220]$, $m_3 = [00001010000220]$, and $m_4 = [00001010001120]$.



Markings	
m	[0 0 0 0 0 1 1 2 0 0 0 0 0 0]
m_1	[0 0 0 0 0 1 1 1 0 0 0 1 1 0]
m_2	[0 0 0 0 0 1 1 0 0 0 0 2 2 0]
m_3	[0 0 0 0 1 0 1 0 0 0 0 2 2 0]
m_4	[0 0 0 0 1 0 1 0 0 0 1 1 2 0]

Earliest task-finished set {tr5, tr9}

ebl_set	Enabled	nop	Depth fired	Remained
{tr5, tr9}	{tr5, tr7, tr10}	{tr10}	tr10	{tr5, tr9}
{tr5, tr9}	{tr9, tr10, tr11}	{tr10}	tr10	{tr5, tr9}
{tr5, tr9}	{tr5, tr7, tr9, tr11}	{ }	tr5	{tr9}
{tr9}	{tr7, tr9, tr11}	{ }	tr9	{ }

Figure 5.6: Check whether a marking is a candidate marking

5.3.3 Proposed Verification Method

The proposed high-level verification method includes two stages: the Breadth-First and the Depth-First traverse procedures. At the first stage, the Breadth-First traverse procedure tries to find candidate markings, the correct retimed

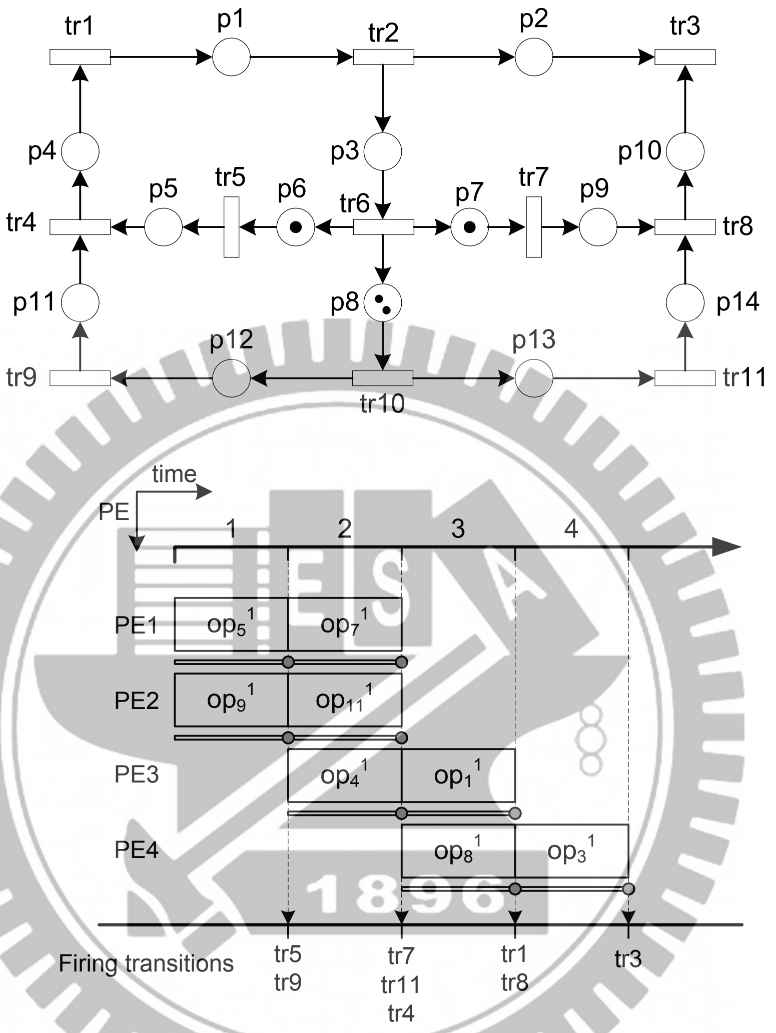


Figure 5.7: An example schedule and 2nd order IIR filter

FSFGs, from reachability tree. At the second stage, the Depth-First traverse procedure verifies given schedule by checking the candidate markings. Since, the nodes of reachability tree are exponential growth with the height of the tree, two-stages method is the better policy. The verification method shortens the searching space by finding candidate markings at the first stage. At the second stage, it verifies given schedule by checking candidate markings rather than all the reachable markings of reachability tree.

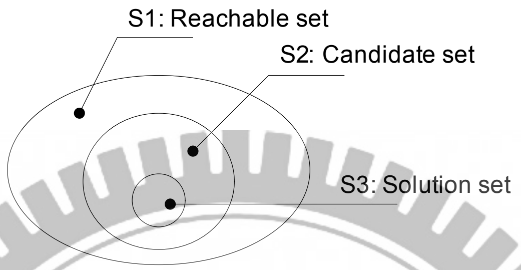


Figure 5.8: The relation between reachable, candidate and solution marking sets

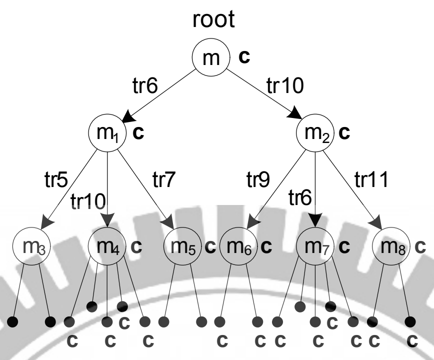
Assuming there are n operations in a given FSFG, and hence there are n transitions in the corresponded PN model. Let f be the *unfolding factor* of a given schedule while designers performing *unfolding* technique on their FSFG design. At the first stage, the procedure tries to find the candidate marking set from the reachable marking set from the reachability tree and fires each transition once each time. The height of each marking in reachability tree is the distance from the root node to itself. Since, during one iteration period of the schedule S, $le(S)$, each scheduled task must be fired once, the height can also be seen as the number of transitions that have been fired since the root node. Thus, for an n-tasks schedule, the *upper height-bound* of the reachability tree is bounded by $H_{up} = f \cdot n$. At the second stage, it continually finds the solution marking set from the candidate marking set. The set relation between three marking sets is shown in Figure. 5.8, that is $S3 \subseteq S2 \subseteq S1$. The purpose of the first stage is trying to reduce the searching space from reachable marking set $S1$ to candidate marking set $S2$, while the second stage is trying to find solution marking set $S3$ from candidate marking set $S2$.

5.3.4 First stage: Breadth-First traverse procedure

At the first stage of the verification method, we apply Breadth-First traverse procedure to find the candidate markings from the reachability tree. Three approaches, which include the exhaustive, the early-terminated and the optimal approaches, are proposed in this work and discussed in the following sections.

5.3.4.1 The exhaustive approach

The first approach to verify a given schedule of a FSFG is the exhaustive approach. It tries to build reachability tree with Breadth-First traverse procedure. The reachability tree contains all the reachable markings of a Petri net. As an example, Figure. 5.9 shows a reachability tree of the Petri net in Figure. 5.7. In Figure. 5.9, each node in the tree associated with a reachable marking of the Petri net. The root node of the reachability tree is the initial marking m of a given PN model. In this marking, two transitions are enabled: tr_6 and tr_{10}. For each enabled transition, the procedure creates new nodes in reachability tree for the reachable markings which result from firing both transitions. An arc, labeled by the transition fired, leads from the initial marking to each new node. Then it applies candidate checking for each new node to check whether a marking is candidate. The procedure continually builds reachability tree for each new produced node until reach the *upper height-bound* H_{up}, which is bounded by $H_{up} = f \cdot n$ for a finite n-tasks schedule. The pseudo-code of the exhaustive approach is shown in Figure. 5.10.



#.	States	Candidate	Branches
m	0 0 1 0 0 0 0 1 0 0 0 0 0 0	C	{tr6, tr10}
m_1	0 0 0 0 0 1 1 2 0 0 0 0 0 0	C	{tr5, tr10, tr7}
m_2	0 0 1 0 0 0 0 0 0 0 0 1 1 0	C	{tr9, tr6, tr11}
m_3	0 0 0 0 1 0 1 2 0 0 0 0 0 0	-	{tr10, tr7}
m_4	0 0 0 0 0 1 1 1 0 0 0 1 1 0	C	{tr9,tr5,tr10,tr11,tr7}
m_5	0 0 0 0 0 1 0 2 1 0 0 0 0 0	C	{tr5, tr10}
m_6	0 0 1 0 0 0 0 0 0 0 1 0 1 0	C	{tr6, tr11}
m_7	0 0 0 0 0 1 1 1 0 0 0 1 1 0	C	{tr9,tr5,tr10,tr11,tr7}
m_8	0 0 1 0 0 0 0 0 0 0 0 1 0 1	C	{tr9, tr6}

• • • • • • • •

Figure 5.9: Build reachability tree with Breadth-First search algorithm

```
 1: procedure BFS_BUILD_TREE_EXHAUSTIVE($\mu_0$)                ▷ root marking $\mu_0$
 2:     Initialize Queue structure $Q$
 3:     Allocate new node $nn$                                   ▷ initialize root node
 4:     $nn.visited \leftarrow false$
 5:     $nn.marking \leftarrow \mu_0$
 6:     $nn.height \leftarrow 0$
 7:     $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)               ▷ candidate check
 8:     $Q$.ENQUEUE($nn$)
 9:
10:     for all unvisited node $n \in Q$ do
11:         $n.visit \leftarrow true$
12:         if $n.height \leqslant H$ then                        ▷ max. height $H$
13:             $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($n.marking$)     ▷ negative test
14:             for all transition $tr \in ebl\_set$ do
15:                 $\mu \leftarrow n.marking$
16:                 $\kappa \leftarrow$ MAKE_FIRING_VECTOR_FROM($tr$)
17:                 $\mu' \leftarrow \mu + [A] \cdot \kappa$                          ▷ Eq.5.5
18:                 Allocate new node $nn$
19:                 $nn.marking \leftarrow \mu'$
20:                 $nn.visited \leftarrow false$
21:                 $nn.height \leftarrow n.height + 1$
22:                 $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
23:                 CREATE_BRANCH($n, nn$)              ▷ create branch from $n$ to $nn$
24:                 $Q$.ENQUEUE($nn$)
25:             end for
26:         end if
27:     end for
28: end procedure
```

Figure 5.10: The exhaustive approach

In the beginning of Figure. 5.10, function *is_candidate* checks whether the marking μ is candidate. Then, marking μ is marked unvisited and enqueued in a queue structure Q. For each unvisited node in Q, the algorithm finds enabled set of transitions, creates breaching nodes and applies candidate checking on each new produced node. At last, new produced nodes are enqueued in Q and wait for next iteration, in lines 10 to 27.

The queue structure Q is a first-in-first-out queue. The markings in Q need to be processed are in ascending order with respected to their height. As an example in Figure. 5.9, the traverse order of the reachability tree which is rooted by marking m is $m, m_1, m_2, \ldots, m_8, \ldots$.

5.3.4.2 The early-terminated approach

The second approach to verify a schedule of a given FSFG is called the early-terminated approach which improves the exhaustive approach. Before introducing the improved approach, we first consider Lemma 5.1.

Lemma 5.1 *Let T_{tree} be a reachability tree which is bounded by upper height-bound H_{up} and m_1 be any one of the candidate markings in T_{tree}. For any other candidate marking m_2 in the successor path of marking m_1, m_2 is in the solution marking set $S3$ if and only if m_1 is in $S3$.*

Proof: Let etf_set be the earliest task-finished set of a given schedule and transition sequence σ_1 be a firing sequence that leads $m_1 \in S2$ from root marking m_r of T_{tree} to be a candidate marking, that is $m_r \xrightarrow{\sigma_1} m_1$. Assuming there exists another candidate marking $m_2 \in S3$, $m_2 \neq m_1$, with firing sequence σ_2 that leads m_2 from root marking m_r of T_{tree} to be a candidate marking, that is $m_r \xrightarrow{\sigma_2} m_2$, and is in the successor path of marking m_1.

As defined in Definition 1, it must be satisfied that $etf\_set \subseteq \sigma_1$ and $etf\_set \subseteq \sigma_2$ where the elements of σ_1 and σ_2 are all in $\{nop\} \cup \{etf\_set\}$. As described in assumption, m_2 is in the successor path of marking m_1, it is still satisfied that $\sigma_2 = \sigma_1 \cup \{nop\}$. This implies m_2 is in solution marking set $S3$ if and only if m_1 is in $S3$. ■

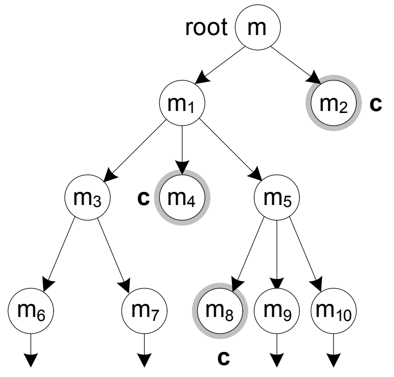


Figure 5.11: The traverse order of early-terminated approach

The early-terminated approach is based on the exhaustive approach and uses Lemma 5.1. It tries to minimize the size of candidate set $S2$ from reachable set $S1$. The difference between the exhaustive and the early-terminated approaches is that when an enqueued unvisited marking is candidate, the early-terminated approach ignores the candidate marking and marks as a visited node. Then, it proceeds other unvisited nodes in queue Q until all the markings have been visited. In Figure. 5.11, as an example, the traverse order of the early-terminated approach is $m, m_1, m_2, m_3, \ldots, m_{10}$. The pseudo-code of the earliest-terminated traverse method is shown in Figure. 5.12. In lines 12 to 14, it ignores the candidate marking and proceeds other unvisited nodes.

5.3.4.3 The optimal approach

The third approach to verify a schedule of a given FSFG is the optimal approach which is improved from the early-terminated approach. In order to reduce reachable marking set $S1$ of the reachability tree, it tries to merge the redundant nodes when it proceeds Breadth-First traverse.

Let m be an unvisited node to be processed. If m is a candidate marking, it ignores this node by using Lemma 5.1 and proceeds other unvisited nodes in

```
procedure BFS_BUILD_TREE_EARLY_TERMINATED($\mu_0$)    ▷ root marking $\mu_0$
    Initialize Queue structure $Q$
    Allocate new node $nn$    ▷ initialize root node
    $nn.visited \leftarrow false$
    $nn.marking \leftarrow \mu_0$
    $nn.height \leftarrow 0$
    $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
    $Q$.ENQUEUE($nn$)

    for all unvisited node $n \in Q$ do
        $n.visit \leftarrow true$
        if $n.candidate = true$ then
            continue to the next node    ▷ Early-terminate Lemma 5.1
        end if
        if $n.height \leqslant H$ then
            $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($n.marking$)    ▷ negative test
            for all transition $tr \in ebl\_set$ do
                $\mu \leftarrow n.marking$
                $\kappa \leftarrow$ MAKE_FIRING_VECTOR_FROM($tr$)
                $\mu' \leftarrow \mu + [A] \cdot \kappa$    ▷ Eq.5.5
                Allocate new node $nn$
                $nn.marking \leftarrow \mu'$
                $nn.visited \leftarrow false$
                $nn.height \leftarrow n.height + 1$
                $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
                CREATE_BRANCH($n, nn$)
                $Q$.ENQUEUE($nn$)
            end for
        end if
    end for
end procedure
```

Figure 5.12: The early-terminated approach

queue. If m is not a candidate marking, it finds enabled set of transitions and creates new node on each enabled transition. For each new produced node with marking m', if there exists another node in the reachability tree, and has the same marking associated with it, then the node with marking m' is a duplicate node. Since, the marking m' has appeared in the tree, this new produced node is redundant. Then, it merges this redundant node to the existential node and creates transition link from marking m to the existential node. As an example in Figure. 5.13, when it proceeds marking m_5, it founds the new created node with marking m_7 is a duplicate node. It merges these nodes and creates transition from m_5 to m_7. Then, it continually proceeds other unvisited nodes in queue.

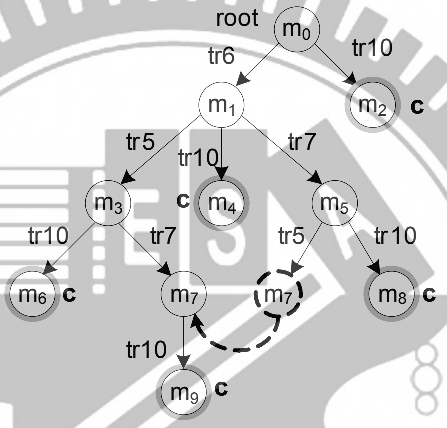


Figure 5.13: Merge the redundant node in optimal traverse approach

The pseudo-code of the optimal approach is shown in Figure. 5.14. In lines 12 to 14, if the node n is a candidate marking, then it ignores this node by using Lemma 5.1 and proceeds the other nodes in queue Q. In lines 26 to 32, function $find\_duplicate\_node$ checks whether the new created node nn is duplicate. If there exists a duplicate node, it ether returns the dual node to *dual_node* or returns *null*. It continually proceeds other unvisited nodes until all nodes are visited.

```
procedure BFS_BUILD_TREE_OPTIMAL($\mu_0$)                    ▷ root marking $\mu_0$
    Initialize Queue structure $Q$
    Allocate new node $nn$                                   ▷ initialize root node
    $nn.visited \leftarrow false$
    $nn.marking \leftarrow \mu_0$
    $nn.height \leftarrow 0$
    $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
    $Q$.ENQUEUE($nn$)

    for all unvisited node $n \in Q$ do
        $n.visit \leftarrow true$
        if $n.candidate = true$ then
            continue to the next node                        ▷ Early-terminate Lemma 5.1
        end if
        if $n.height \leqslant H$ then                       ▷ max. height $H$
            $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($n.marking$)    ▷ negative test
            for all transition $tr \in ebl\_set$ do
                $\mu \leftarrow n.marking$
                $\kappa \leftarrow$ MAKE_FIRING_VECTOR_FROM($tr$)
                $\mu' \leftarrow \mu + [A] \cdot \kappa$                    ▷ Eq.5.5
                Allocate new node $nn$
                $nn.marking \leftarrow \mu'$
                $nn.visited \leftarrow false$
                $nn.height \leftarrow n.height + 1$
                $nn.candidate \leftarrow$ IS_CANDIDATE($nn$)
                $dual\_node \leftarrow$ FIND_DUPLICATE_NODE($nn$)
                if $dual\_node \neq null$ then
                    CREATE_BRANCH($n, dual\_node$)
                else
                    CREATE_BRANCH($n, nn$)
                    $Q$.ENQUEUE($nn$)
                end if
            end for
        end if
    end for
end procedure
```

Figure 5.14: The optimal traverse approach

5.3.5 Second stage: Depth-First traverse method

At the second stage, we apply Depth-First traverse procedure to verify a schedule on candidate markings rather than all reachable markings in PN model. As showing in Figure. 5.15, a candidate marking $\boldsymbol{m}$ which is found in the first stage is probably the correct marking, the correct retimed FSFG, that leads a given schedule being valid. For a given schedule in Figure. 5.7, task tr_5 and task tr_9 are scheduled and finished at the first step of the schedule. The procedure tries to fire one transition of these scheduled tasks or enabled *nop* operations once each time during the first scheduled step. At the end of the first step, marking m_1 is obtained from candidate marking m by firing transition sequence $\sigma : tr_6\, tr_5\, tr_{10}\, tr_9$, that is $m \xrightarrow{\sigma} m_1$, where transition tr_6 and tr_{10} are *nop* operations. The procedure continually traverses entire length of the schedule step-by-step until all the scheduled tasks are fired. A given schedule is said to be valid if and only if all the markings in the traverse path are valid.

The pseudo-code of the second stage is shown in Figure. 5.16. At the beginning, the procedure marks all nodes in queue Q unvisited in line 3. For all unvisited candidate markings, it traverses all entire length of the schedule, applies procedure $go\_further\_depth\_firing$ at each scheduled step in line 12. Procedure $go\_further\_depth\_firing$ has three parameters: the absolute step s of a given schedule, the current marking μ and the output marking μ'. It returns *true* if and only if the firing markings during step s are valid. A valid schedule exists if and only if all the firing markings of the entire length of the schedule are valid. At last in lines 19 to 21, all solution markings are added to sol_set.

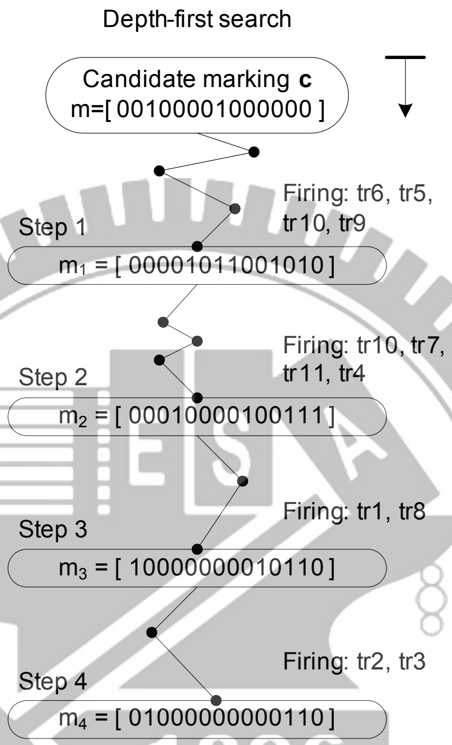


Figure 5.15: Verify schedule with Depth-First search algorithm

```
procedure DFS_TRAVERSE_PATH($Q$)                              ▷ queue structure $Q$
    Initialize schedule structure $sch$
    Mark all node $n \in Q$ unvisited

    for all unvisited candidate node $n \in Q$ do
        $n.visit \leftarrow true$
        $\mu \leftarrow n.marking$
        for all control step $s \in [1, 2, \ldots, sch.max\_step]$ do
            if there is no task end at step $s$ then
                continue to next $s$
            end if
            $valid \leftarrow$ GO_FURTHER_DEPTH_FIRING($s, \mu, \mu'$)
            if $valid = true$ then              ▷ depth firing is valid at step $s$
                $\mu \leftarrow \mu'$
            else                                ▷ depth firing is not valid at step $s$
                break
            end if
        end for
        if $valid = true$ then
            $sol\_set \leftarrow \{sol\_set \cup n.marking\}$                ▷ solution set
        end if
    end for
end procedure
```

Figure 5.16: Path traverse with Depth-First search algorithm

```
procedure GO_FURTHER_DEPTH_FIRING($s, \mu, out\_mark$)
    $tran\_set \leftarrow$ FIND_ALL_TASKS_FINISHED_AT_STEP($s$)

    $cm \leftarrow \mu$                                          ▷ current marking $cm$
    $violate \leftarrow false$
    while ({$tran\_set$} is not empty) and ($violate = false$) do
        $fired \leftarrow false$
        $ebl\_set \leftarrow$ FIND_ENABLED_TRANS($cm$)                ▷ negative test
        for $i \leftarrow 1, |T|$ do                    ▷ total number of transitions $|T|$
            $tr \leftarrow$ the $i$th transition
            if $tr \in \{ebl\_set\} \cap \{nop\}$ then                ▷ fire $nop$ transition
                $\mu' \leftarrow$ FIRE_TRANSITION($cm, tr$)
                $fired \leftarrow true$
            else if $tr \in \{ebl\_set\} \cap \{tran\_set\}$ then  ▷ fire $tran\_set$ transition
                $tran\_set \leftarrow \{tran\_set\} \setminus tr$          ▷ delete $tr$ from $tran\_set$
                $\mu' \leftarrow$ FIRE_TRANSITION($cm, tr$)
                $fired \leftarrow true$
            end if
            if $fired = true$ then                             ▷ one firing each time
                break
            end if
        end for                                                   ▷ next transition
        if $fired = false$ then                         ▷ there is no firing occurred
            $violate \leftarrow true$
        else if IS_MARKING_VALID($\mu'$) = $false$ then      ▷ marking is not valid
            $violate \leftarrow true$
        else
            $cm \leftarrow \mu'$                                          ▷ next iteration
        end if
    end while
    if $violate \leftarrow false$ then
        $out\_mark \leftarrow cm$                        ▷ $out\_mark$ is the output marking
        return true
    else
        return false
    end if
end procedure
```

Figure 5.17: One more deep step firing

5.4 The Complexity Analysis

Assuming there are n non-nop operations in a given FSFG. Let f be the *unfolding factor* of a given schedule. As described in previous section, the upper-height of the reachability tree of the corresponded PN model is bounded by $H_{up} = f \times n$. The complexity analysis of the proposed two-stages verification method is discussed as following.

At the first stage, three approaches are proposed including the exhaustive, the early-terminated and the optimal traverse methods. In the first approach, each node in the reachability tree has n enabled transitions in worse case, the level 0 (the root node) has one node.
Level 1 has n nodes
Level 2 has $(n)(n) = n^2$ nodes
Level 3 has $(n^2)(n) = n^3$ nodes
$\ldots\ \ldots$
Level $f \cdot n$ has $(n^{f \cdot n-1})(n) = n^{f \cdot n}$ nodes

The total number of nodes is:

$$1 + n + n^2 + \ldots + n^{f \cdot n} = \left(n^{f \cdot n+1} - 1\right) / (n - 1) \tag{5.8}$$

Thus, the space complexity of the heuristic approach is $O(N^{f \cdot N})$, in worse case.

In the second approach, the early-terminated approach, the algorithm stops traversing a node while it is candidate. Let p, $p \leq (f \cdot n)$, be the deepest level that Breadth-First traverse procedure can reach. The complexity of the second approach is $O(N^p), p \leq f \cdot n$.

In the third approach, the optimal approach, the algorithm merges duplicate markings in order to reduce the reachable marking set of the reachability tree. Let $x \in \mathbb{Z} = \{1, 2, \cdots\}$ be the merging radio in the reachability tree. The complexity of the three approach is $O((N/x)^p), p \leq f \cdot n$. Thus, the relation of the complexity between three approaches is:

$$O\left(N^{f \cdot n}\right) > O\left(N^p\right) > O\left((N/x)^p\right). \tag{5.9}$$

At the second stage, the algorithm performs Depth-First traverse to verify a given schedule by checking the firing sequence, which contains $f \cdot n$ transitions , of the PN model. Thus, the complexity is $O(f \cdot n)$, in worse case.

5.5 Experimental Results

We have implemented these three approaches as the proposed formal verification algorithms. Each of these approaches is applied to several dataflow algorithms. Table. 5.1 shows the statistics of these designs.

Table 5.1: The statistics of test designs

Design name	num. vertices	num. edges	num. delays	Size of PN (Place x Trans.)	Schedule length	Unfolding factor
iir2d-sch1	8	14	2	(14 x 11)	6	1
iir2d-sch2	8	14	2	(14 x 11)	4	1
iir2d-sch3	8	14	2	(14 x 11)	4	1
iir2d-sch4	8	14	2	(14 x 11)	4	1
iir3d-sch1	12	21	3	(21 x 16)	6	1
iir3d-sch2	12	21	3	(21 x 16)	6	1
p243-sch1	5	7	5	(7 x 5)	96	6
p243-sch2	5	7	5	(7 x 5)	96	6
ewf-sch1	34	47	0	(47 x 34)	40	1
ewf-sch2	34	47	0	(47 x 34)	40	1

Table 5.2: The experimental results

Test schedule	Exhaustive Time (sec)	Exhaustive Res. usage	Early-terminated Time (sec)	Early-terminated Res. usage	Optimal Time (sec)	Optimal Res. usage
iir2d-sch1	171.01	168648	0.17	16	0.19	16
iir2d-sch2	180.38	168648	0.2	14	0.2	14
iir2d-sch3	206.81	168701	24.80	34084	0.33	244
iir2d-sch4	179.47	171341	19.845	35720	0.32	293
iir3d-sch1	n/a	n/a	0.19	19	0.21	19
iir3d-sch2	n/a	n/a	0.18	19	0.21	19
p243-sch1	n/a	n/a	0.2	32	0.21	32
p243-sch2	n/a	n/a	0.22	32	0.21	32
ewf-sch1	n/a	n/a	0.26	36	0.28	36
ewf-sch2	n/a	n/a	0.3	36	0.28	36

Design *iir2d-sch1* to *iir2d-sch4* and design *iir3d-sch1* to *iir3d-sch1* [50] are the second-order and the third-order Infinite Impulse Response filters. Design *p243* [50] is a design with *unfolding factor* 6, the lengths of schedule *p243-sch1* and *p243-sch2* are both 96 steps. Design *ewf-sch1* and *ewf-sch2* are low power schedules for the Elliptic Wave Filter in [74].

Table. 5.2 shows the experimental results of using three approaches. The optimal approach outperforms the others in terms of time and resource usage.

5.6 Summary

This work aims to exploit formal verification techniques for high-level synthesis. In the top-down design flow, design errors should be removed as early as possible; otherwise, errors detected at the later stages will result a costly, time-consuming redesign cycles. Although formal verification for logic synthesis has been studied very extensively, little work has been done for high-level synthesis. The work presents a novel verification flow that can efficiently detect the design errors from the results of high-level synthesis. As shown in the experimental results, we can apply the optimal approach for the first phase to efficiently verify complex design cases.

CHAPTER 6

Conclusion and Future Works

The existential formal verification techniques including equivalence checking and model checking have proven to be successful verification methods that be applied to real industrial design. However, since it requires the design to be synthesized at RTL or gate level, high-level faults may not be revealed until the RTL generated.

This thesis investigates system-level verification method using Petri Net model to detect high-level faults of HLS result for DSP-driven or dataflow systems. While comparing to the conventional RTL verification, early faults detection may reduce redesigning time and costs, and hence it speeds up time-to-market cycle time. We studied Petri Net theory in verifying the HLS of dataflow system design. The verification methodology starts from dataflow graph modeling as mentioned in Chapter. 2. The proposed PN conversion method converts the dataflow FSFG graph into PN model. In PN domain, HLS optimal manipulations, such as arithmetic transformation and scheduling, can be seen a serial token-state movements of PN model. Valid optimization techniques must satisfy the validity of PN token movement rules. Based on the PN theory, two verification frameworks are proposed in Chapter. 4 and Chapter. 5. Both verification flows are two-folded corresponded to the static and dynamic phase verification. In the first phase, PN theory is utilized to verify the arithmetic transformations of HLS, such as the retiming and unfolding techniques. Several tree traversal based methods are proposed in Chapter. 4 and Chapter. 5 to find the candidate FSFG design from

PN reachability tree. In the second phase, the verifier in Chapter. 4 attempts to verify schedule scheme by using PN theory, and Chapter. 5 addresses a SMV-based model checker to check the correctness of schedule. We also conclude the best traverse algorithm from the experimental results.

Although, this thesis provides the whole new verification concepts, which mention the PN conversion, high-level fault modeling, verifying technique developing and the implementation of the frameworks, there still exists some possible subjects for further studies. We will address some limitation and further work in the following sections.

6.1 Petri Net Modeling Limitation

Essentially, the introducing of PN modeling for the dataflow graph can handle all kinds HLS optimization techniques as we mention in Chapter 2. However, PN based modeling still suffers from a couple known limitations. We state these limitations in the next subsections.

6.1.1 Loop Shrunk

Loop shrunk [50] is an optimization technique that it reduce the iteration period bound (IPB) would be to reduce the computational latency of the critical loop, effectively reducing the IPB bound.

Considering the loop segment shown in Figure. 6.1(a) where the chain of two additions within the loop is reduced to one addition within the loop in Figure. 6.1(b). This is an application of associativity, such that $x = (a + b) + c = a + (b + c)$. Distributivity (followed by associativity) can be used to shrink the computational latency in the loop as shown in Figure. 6.2(a)-(b). The distributivity interchanges the multiplication and addition operations, which when followed by an associativity operation can shrink the loop. As a consequence, the IPB can be reduced, if the other loops do not increase their computational latencies at the same time.

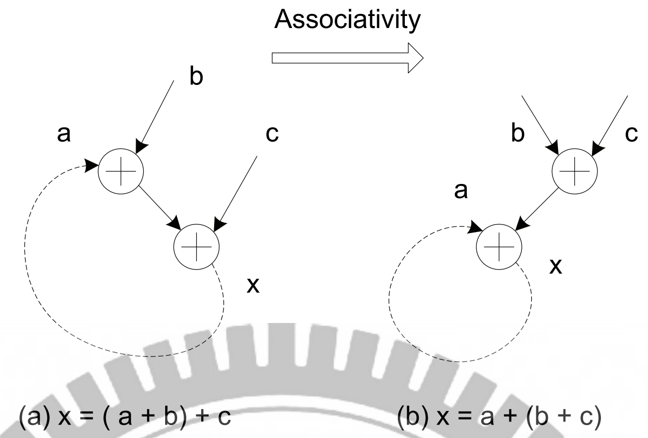


Figure 6.1: (a) Original segment of a loop, (b) loop shrunk by associativity of addition

As an example, consider the second order IIR filter shown in Figure. 6.3(a). By simple associativity at the input, the IPB can be reduced from 3 to 2 as shown in Figure. 6.3(b).

The limitation of the proposed verification is that PN based verification can only detect the high-level faults of HLS result under the architecture of the design unchanged. The loop shrunk optimization is based on arithmetic associativity and distributivity rules. Such optimization does change the positions of the process elements from the original design. Even the functionality remains unchange, loop shrunk optimization does change the architecture of the original design.

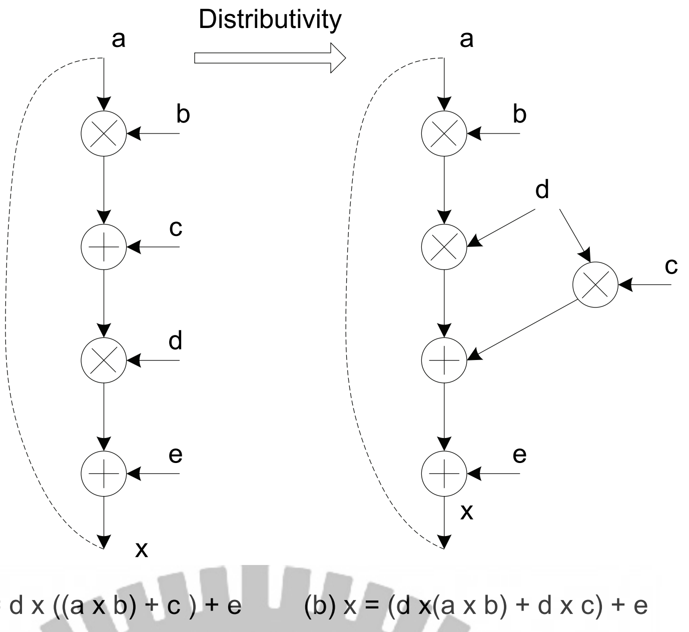

Distributivity
a
b
c
d
e
x
(a) x = d x ((a x b) + c) + e
(b) x = (d x(a x b) + d x c) + e

Figure 6.2: (a) Original segment of a loop, (b) distributivity interchanges the multiplication and addition, allowing application of associativity to shrink the loop

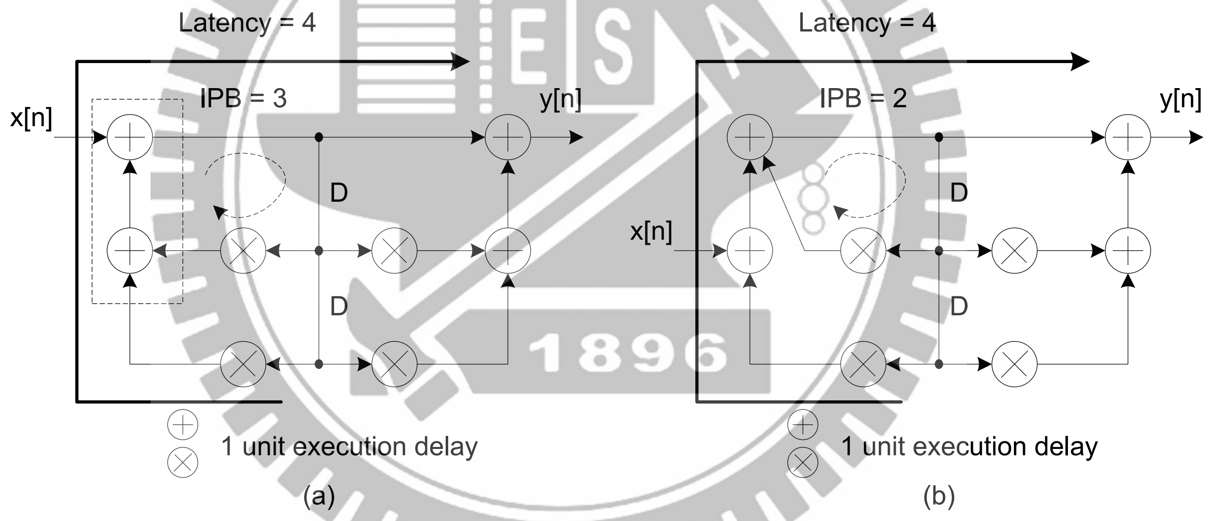


Figure 6.3: (a) Original FSFG with IPB = 3 and Latency = 4, (b) IPB = 2 after associativity

6.1.2 Look-Ahead

Lookahead is a HLS optimization technique introduced by Prihi and Messerschmitt in [89, 90]. The basic idea of lookahead requires either pipelining or unfolding to achieve optimal efficiency. A case of the IIR filter design is given in Figure. 6.4(a), the original FSFG has a $IPB = 2(T_m + T_a)/2$. After using arithmetic lookahead or distributivity optimization technique, the result design in Figure. 6.4(b) has a less iteration period bound with $IPB = (T_m + T_a)/2$. In general case, design may apply $M - 1$ steps IIR filter of the lookahead design of Figure. 6.4(b). The result $M - 1$ steps IIR filter with $IPB = (T_m + T_a)/M$ is shown in Figure. 6.5.

As showing in previous cases, lookahead optimization technique first utilizes the arithmetic distributivity, then the unfolding and retiming techniques are applied to achieve optimal iterator period bound. We can see that even the functionality remains the same, the architecture of the original design has been changed after lookahead. The proposed verification technique may not be used to detect such kind of architecture changes yet.

6.2 SystemC using Petri Net Modeling

The FSFG representation is commonly used in high-level design, however, the language based descriptions, such as SystemC [91,92], have more numerous practical applications. SystemC is a modeling platform consisting of C++ class libraries and a simulation kernel for design at the system-behavioral and RTL level. One of the major advantages of SystemC is that it can be used to describe a system at several levels of function description and down to synthesizable RTL. In [93], the authors gave the DSP recommended design flow. In the recommended flow, designs still need refinement or optimization techniques in their design flow. As showing in Figure. 6.6, the verification to check HLS design faults also becomes a challenge work.

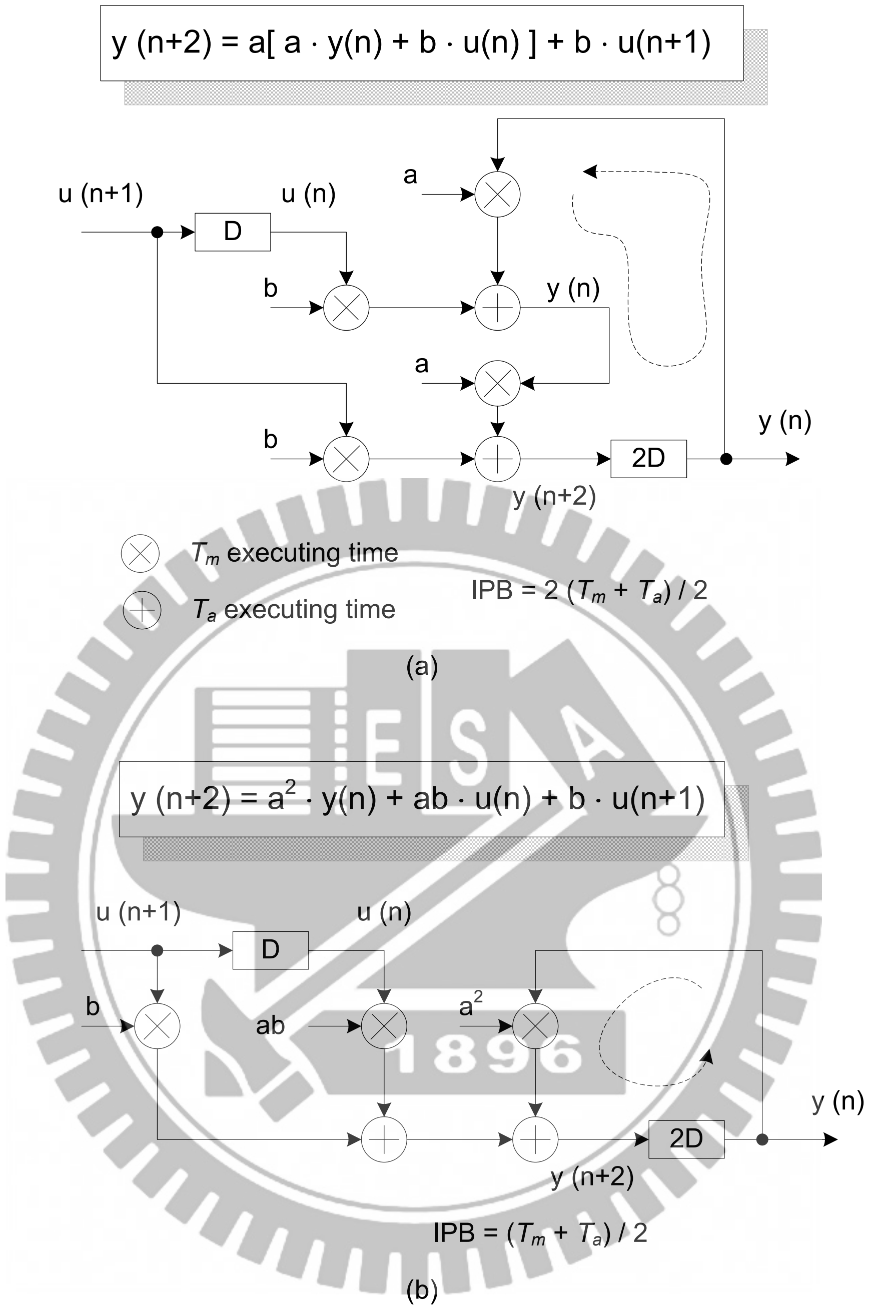

y (n+2) = a[a · y(n) + b · u(n)] + b · u(n+1)
u (n+1)
D
u (n)
a
b
y (n)
a
b
2D
y (n)
y (n+2)
T_m executing time
T_a executing time
IPB = 2 (T_m + T_a) / 2
(a)
y (n+2) = a² · y(n) + ab · u(n) + b · u(n+1)
u (n+1)
D
u (n)
b
ab
a²
2D
y (n)
y (n+2)
IPB = (T_m + T_a) / 2
(b)

Figure 6.4: (a) Original IIR filter with $IPB = 2(T_m + T_a)/2$, (b) $IPB = (T_m + T_a)/2$ after lookahead

$$\boxed{y(n+M) = a^M \cdot y(n) + \sum_{i=0}^{M-1} a^i \cdot b \cdot u(n+M-1-i)}$$

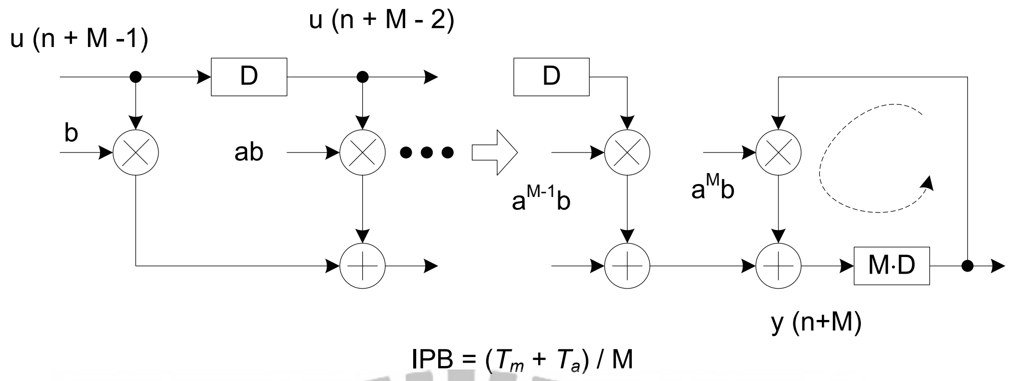


Figure 6.5: Applying $M-1$ steps IIR filter of the lookahead design in previous case, with $IPB = (T_m + T_a)/M$

HLS is crucial for the success of the design, the verification for HLS is even a challenge work. Except low level technique, HLS still dominates the performance of a system. PN based verification open the way to the development of the HLS formal verification techniques. We hope PN based verification method can handle more complex design in the future.

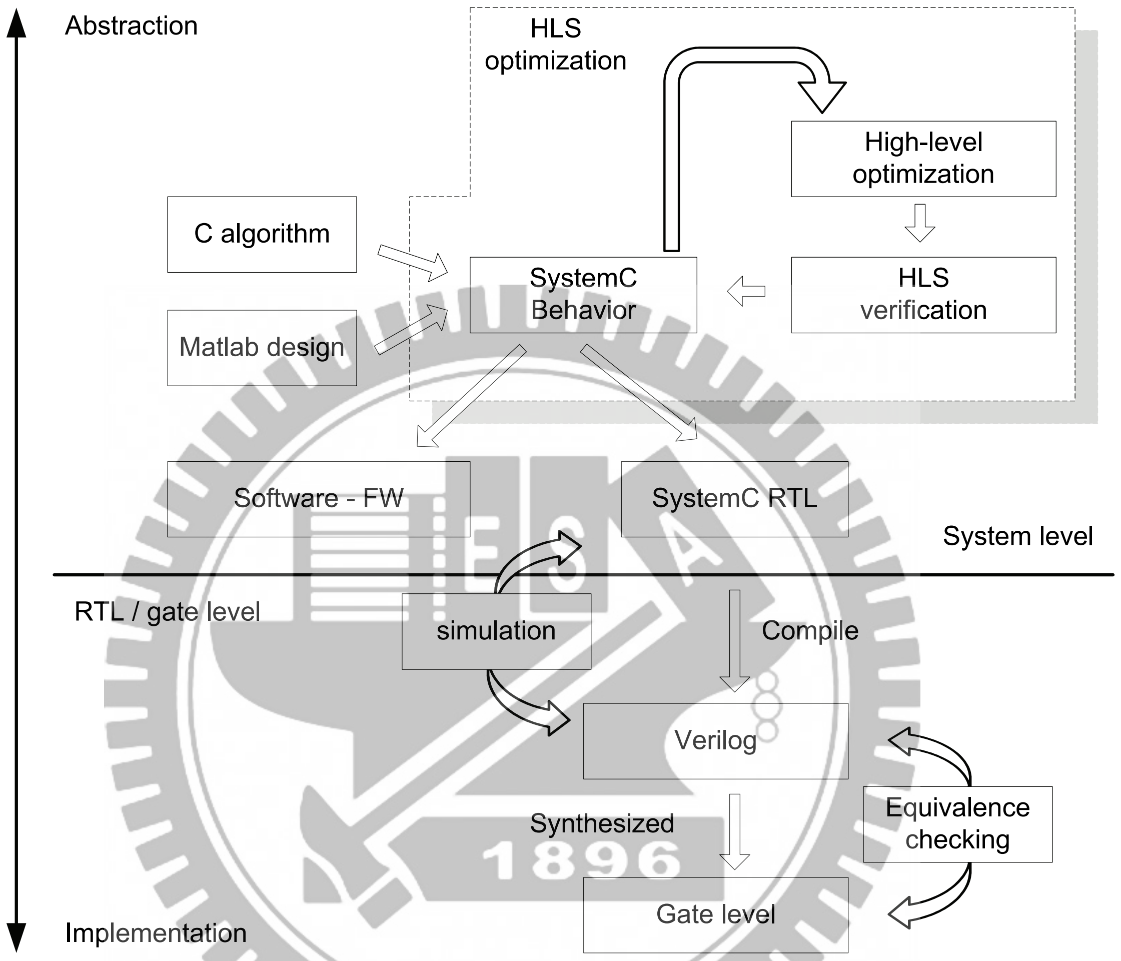


Figure 6.6: System-level design flow and the PN-based verification

References

[1] Rolf Drechsler, "Towards formal verification on the system level", *Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*, vol. 28, no. 30, pp. 2–5, jun 2004.

[2] Alan John Hu and Andrew K. Martin, *Formal Methods In Computer-aided Design*, Springer, 2004.

[3] William K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches*, Prentice Hall, 2005.

[4] Carl Pixley, "Guest editor's introduction: Formal verification of commercial integrated circuits", *IEEE Design and Test*, vol. 18, no. 4, pp. 4–5, jul 2001.

[5] Vigyan Singhal, Carl Pixley, Adnan Aziz, and Robert K. Brayton, "Theory of safe replacements for sequential circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 2, pp. 249–265, feb 2001.

[6] Aarti Gupta, "Formal hardware verification methods: a survey", *Formal Methods in System Design*, vol. 1, no. 2-3, pp. 151–238, oct 1992.

[7] Christoph Kern and Mark R. Greenstreet, "Formal verification in hardware design: a survey", *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 4, no. 2, pp. 123–193, apr 1999.

[8] Randal E. Bryant and James H. Kukula, "Formal methods for functional verification", *International Conference on Computer-Aided Design (ICCAD)*, 2002.

[9] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin, "A framework for constrained functional verification", *International Conference on Computer Aided Design*, pp. 142–145, nov 2003.

[10] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled, *Model Checking*, MIT Press, 1999.

[11] Kedar S. Namjoshi and E. Allen Emerson, *Verification, Model Checking, And Abstract Interpretation*, Springer, 2006.

[12] Tsung-Hsi Chiang and Lan-Rong Dung, "System level verification on high-level synthesis of dataflow algorithms using Petri net", *WSEAS Transactions on Circuits and Systems*, vol. 5, no. 6, pp. 790–796, 2006.

[13] Tsung-Hsi Chiang and Lan-Rong Dung, "Verification method of dataflow algorithms in high-level synthesis", *Journal of System and Software*, vol. 80, no. 8, pp. 1256–1270, 2007.

[14] Tsung-Hsi Chiang and Lan-Rong Dung, "On verification on dataflow scheduling", *to be appeared in International Journal of Software Engineering and Knowledge Engineering*, 2007.

[15] Tsung-Hsi Chiang and Lan-Rong Dung, "Hybrid verification technique for high-level synthesis of dataflow algorithms", *WSEAS Transactions on Circuits and Systems*, vol. 6, no. 3, pp. 348–354, 2007.

[16] Tsung-Hsi Chiang and Lan-Rong Dung, "Modeling and formal verification of dataflow graph in system-level design using Petri net", *IEEE International Symposium on Circuits and Systems (ISCAS 2005)*, 2005.

[17] Tsung-Hsi Chiang and Lan-Rong Dung, "System-level verification of dataflow graph using Petri net model", *16th VLSI Design/CAD Taiwan*, 2005.

[18] Tsung-Hsi Chiang and Lan-Rong Dung, "System-level verification on high-level synthesis of dataflow graph", *IEEE International Symposium on Circuits and Systems (ISCAS 2006)*, 2006.

[19] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant, "Efficient implementation of a BDD package", *Proceedings of the 27th ACM/IEEE conference on Design automation*, pp. 40–45, 1991.

[20] Randal E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams", *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, sep 1992.

[21] Randal E. Bryant, "Graph-based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, aug 1986.

[22] Rolf. Drechsler, Rudiger Ebendt, and Gorschwin Fey, *Advanced BDD Optimization*, Springer, 2005.

[23] Ruy J. G. B. de Queiroz, *Logic for Concurrency and Synchronisation*, Springer, 2003.

[24] Jun Yuan, Kurt Shultz, Carl Pixley, Hillel Miller, and Adnan Aziz, "Modeling design constraints and biasing in simulation using BDDs", *IEEE/ACM International Conference on Computer-Aided Design*, pp. 584–589, 1999.

[25] Kenneth L. McMillan, "Applying SAT methods in unbounded symbolic model checking", *14th International Conference on Computer Aided Verification*, pp. 250–264, jul 2002.

[26] G. Parthasarathy, K-T. Cheng, and C-Y Huang, "An analysis of ATPG and SAT algorithms for formal verification", *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop (HLDVT '01)*, pp. 177–182, 2001.

[27] Mukul R. Prasad, Armin Biere, and Aarti Gupta, "A survey of recent advances in SAT-based formal verification", *Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, 2005.

[28] J. R. Burch, E. M. Clarke, and D. E. Long, "Symbolic model checking with partitioned transition relations", *Proceeding of International Conference on Very Large Scale Integration*, , no. 49–58, 1991.

[29] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill, "Symbolic model checking for sequential circuit verification", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 4, pp. 401–424, apr 1994.

[30] Hyeong-Ju Kang and In-Cheol Park, "SAT-based unbounded symbolic model checking", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 2, pp. 129–140, feb 2005.

[31] Ganapathy Parthasarathy, Madhu K. Iyer, Kwang-Ting Cheng, and Li-C. Wang, "Safety property verification using sequential sat and bounded model checking", *IEEE Design and Test of Computers*, vol. 21, no. 2, pp. 132–143, mar 2004.

[32] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential circuit verification using symbolic model checking", *In 27th ACM/IEEE Design Automation Conference*, 1990.

[33] Edmund M. Clarke and David E. Long, "Model checking and abstraction", *ACM Transactions on Programming Languages and Systems*, 1992.

[34] Matt Kaufmann, Andrew Martin, and Carl Pixley, "Design constraints in symbolic model checking", *Proceedings of the 10th International Conference on Computer Aided Verification*, vol. 1427, pp. 477–487, 1998.

[35] Joseph Kljaich, Brian T. Smith, and Anthony S. Wojcik, "Formal verification of fault tolerance using theorem-proving techniques", *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 366–376, mar 1989.

[36] Mark D Ryan and Michael Huth, *Logic in Computer Science: Modelling and Reasoning about Systems*, Cambridge University Press, 2004.

[37] Pranav Ashar, Subhrajit Bhattacharya, Anand Raghunathan, and Akira Mukaiyama, "Verification of RTL generated from scheduled behavior in a high-level synthesis flow", *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pp. 517–524, 1998.

[38] D. Sarkar, "Register transfer operation analysis during data path verification", *Proceedings of the 2002 conference on Asia South Pacific design automation / VLSI Design*, pp. 172–177, jan 2002.

[39] C. Bolchini, R. Montandon, F. Salice, and D. Sciuto, "Design of VHDL-based totally self-checking finite-state machine anddata-path descriptions", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 1, pp. 98–103, feb 2000.

[40] Dominique Borrione, Julia Dushina, and Laurence Pierre, "A compositional model for the functional verification of high-levelsynthesis results", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 5, pp. 526–530, oct 2000.

[41] C Karfa, C Mandal, D Sarkar, S R. Pentakota, and Chris Reade, "A formal verification method of scheduling in high-level synthesis", *Proceedings of the 7th International Symposium on Quality Electronic Design*, pp. 71–78, 2006.

[42] Nazanin Mansouri and Ranga Vemuri, "Automated correctness condition generation for formal verification of synthesized RTL designs", *Journal of Formal Methods in System Design*, vol. 16, no. 1, pp. 59–91, jan 2000.

[43] Daniel D. Gajski and Loganath Ramachandran, "Introduction to high-level synthesis", *IEEE Design and Test*, vol. 11, no. 4, pp. 44–54, oct 1994.

[44] Luciano Lavagno, Grant Martin, and Louis Scheffer, *Electronic Design Automation for Integrated Circuits Handbook*, CRC, 2006.

[45] Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin, *High-level synthesis : introduction to chip and system design*, Kluwer academic publishers, 1992.

[46] Daniel Gajski and Robert H. Kuhn, "New VLSI tools - guest editors' introduction", *IEEE Computer*, vol. 16, no. 12, pp. 11–14, 1983.

[47] Donald E. Thomas, Elizabeth D. Lagnese, Robert A. Walker, Jayanth V. Rajan, Robert L. Blackburn, and John A. Nestor, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Springer, 1989.

[48] Giovanni De Micheli, *Synthesis and optimization of digital circuits*, McGraw-Hill, 1994.

[49] Keshab K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, Wiley, 1999.

[50] Vijay K. Madisetti, *VLSI Digital Signal Processors*, IEEE Press, 1995.

[51] A. Fettweis, "Realizability of digital filter networks", *Archiv fur Elektronik und Ubertragungstechnik*, vol. 30, no. 2, pp. 90–96, 1976.

[52] Markku Renfors and Yrjo Neuvo, "Fast multiprocessor realizations of digital filters", *IEEE International Conference on ICASSP'80 Acoustics, Speech, and Signal Processing*, vol. 5, pp. 916–919, apr 1980.

[53] Thomas P. Barnwell, C.J.M. Hodges, and Mark Randolph, "Optimum implementation of single time index signal flow graphs on synceronous multiprocessor machines", *IEEE International Conference on ICASSP'82 Acoustics, Speech, and Signal Processing*, vol. 7, pp. 679–682, aug 1982.

[54] T. Barnwell, C. Hodges, and M. Randolph, "Optimum implementation of single time index signal flow graphs on synceronous multiprocessor machines", *IEEE International Conference on ICASSP'82 Acoustics, Speech, and Signal Processing*, vol. 7, pp. 679–682, 1982.

[55] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems", *Journal of VLSI and Computer Systems*, vol. 1, no. 1, pp. 41–67, 1983.

[56] Charles E. Leiserson and James B. Saxe, "Retiming synchronous circuitry", *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[57] Alexander Schrijver, *Theory of Linear and Integer Programming*, John Wiley and Sons, 1998.

[58] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu, "A formal approach to the scheduling problem in high level synthesis", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, apr 1991.

[59] Keshab K. Parhi and David G. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding", *IEEE Transactions on Computers*, vol. 40, no. 2, pp. 178–195, feb 1991.

[60] Keshab K. Parhi, "Algorithm transformations for concurrent processors", *In Proceedings of the IEEE*, vol. 77, no. 12, pp. 1879–1895, dec 1989.

[61] Liang-Fang Chao and Edwin Hsing-Mean Sha, "Scheduling data-flow graphs via retiming and unfolding", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1259–1267, dec 1997.

[62] Nikil D. Dutt and Daniel D. Gajski, "Design synthesis and silicon compilation", *IEEE Design and Test*, vol. 7, no. 6, pp. 8–23, nov 1990.

[63] Pierre G. Paulin and John P. Knight, "Algorithms for high-level synthesis", *IEEE Design and Test*, vol. 6, no. 6, pp. 18–31, nov 1989.

[64] Robert A. Walker and Samit Chaudhuri, "Introduction to the scheduling problem", *IEEE Design and Test*, vol. 12, no. 2, pp. 60–69, jun 1995.

[65] Carl Adam Petri, *Communication with automata*, PhD thesis, University of Bonn, 1966.

[66] James L. Peterson, *Petri net theory and the modeling of system*, Prentice-Hall, 1981.

[67] Tadao Murata, "Petri nets: properties, analysis and applications", *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, apr 1989.

[68] Christos G. Cassandras and Stephane Lafortune, *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 1999.

[69] Rene David and Hassane Alla, *Petri Nets and Grafcet: tools for modelling discrete event systems*, Prentice Hall International, 1992.

[70] Wolfgang Reisig and Grzegorz Rozenberg, *Lectures on Petri nets I: Basic Models*, Springer, 1998.

[71] Wolfgang Reisig and Grzegorz Rozenberg, *Advances in Petri Nets - Lectures on Petri nets II: Applications*, Springer, dec 1998.

[72] Jorg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, *Lectures on Concurrency and Petri Nets*, Springer, 2004.

[73] Grzegorz Rozenberg, *Advances in Petri Nets 1984, 1986, 1988, 1990 - Lecture Notes in Computer Science*, Springer, 1984.

[74] Lan-Rong Dung and Hsueh-Chih Yang, "On multiple-voltage high-level synthesis using algorithmic transformations", dec 2004, number 12, pp. 3100–3108.

[75] Claude Girault and Rudiger Valk, *Petri nets for systems engineering*, Springer, 2003.

[76] Harvey E. Rose, *Linear Algebra: A Pure Mathematical Approach*, Springer, 2002.

[77] Kazuhito Ito, Lori E. Lucke, and Keshab K. Parhi, "ILP-based cost-optimal DSP synthesis with module selection and dataformat conversion", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 582–594, dec 1998.

[78] Vijay K. Madisetti and Bryce A. Curtis, "A quantitative methodology for rapid prototyping and high-levelsynthesis of signal processing algorithms", *IEEE Transactions on Signal Processing*, vol. 42, no. 11, pp. 3188–3208, nov 1994.

[79] Keshab K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis", *Journal of VLSI Signal Processing Systems*, vol. 9, no. 1-2, pp. 121–143, jan 1995.

[80] Xun Liu, Papaefthymiou, Marios C. Papaefthymiou, and Eby G. Friedman, "Retiming and clock scheduling for digital circuit optimization", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 184–203, feb 2002.

[81] George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk, "Optimum and heuristic synthesis of multiple word-length architectures", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 1, pp. 39–57, jan 2005.

[82] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1990.

[83] Lintao Zhang and Sharad Malik, "The quest for efficient Boolean satisfiability solvers", *Proceedings of the 14th International Conference on Computer Aided Verification*, pp. 17–36, jul 2002.

[84] Tracy Larrabee, "Test pattern generation using Boolean satisfiability", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4–15, jan 1992.

[85] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, "Chaff: engineering an efficient SAT solver", *Proceedings of Design Automation Conference*, pp. 530–535, 2001.

[86] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. G. Hwang, "Symbolic model checking 10exp20 states and beyond", *In Logic in Computer Science*, pp. 428–439, 1990.

[87] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs", *Proceedings of the 36th ACM/IEEE conference on Design automation*, pp. 317–320, 1999.

[88] Edmund M. Clarke and E. Allen Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic", in *Lecture Notes In Computer Science*. 1981, vol. 131, pp. 52–71, Springer-Verlag.

[89] Keshab K. Parhi and David G. Messerschmitt, "Pipelined VLSI recursive filter architectures using scattered look-ahead and decomposition", *International Conference on Acoustics, Speech, and Signal Processing (ICASSP-88)*, vol. 4, pp. 2120–2123, apr 1988.

[90] Keshab K. Parhi and David G. Messerschmitt, "Pipeline interleaving and parallelism in recursive digital filters.i. pipelining using scattered look-ahead and decomposition", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 7, pp. 1099–1117, jul 1989.

[91] ", SystemC community, http://www.systemc.org/.

[92] "SystemC version 2.0 users guide", Synopsys Inc, 2003. http://www.systemc.org/.

[93] Itai Yarom and Gabi Glasser, "SystemC opportunities in chip design flow", *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems ICECS 2004*, vol. 13, no. 15, pp. 507–510, 2004.