Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 80 (2007) 1256–1270
The Journal of Systems and Software
www.elsevier.com/locate/jss

Verification method of dataflow algorithms in high-level synthesis

Tsung-Hsi Chiang *, Lan-Rong Dung

Department of Electrical and Control Engineering, National Chiao Tung University, Hsinchu City 300, Taiwan, ROC

Received 9 December 2005; received in revised form 23 August 2006; accepted 19 December 2006
Available online 22 December 2006

Abstract

This paper presents a formal verification algorithm using the Petri Net theory to detect design errors for high-level synthesis of dataflow algorithms. Typically, given a dataflow algorithm and a set of architectural constraints, the high-level synthesis performs algorithmic transformation and produces the optimal scheduling. How to verify the correctness of high-level synthesis becomes a key issue before mapping the synthesis results onto a silicon. Many tools exist for RTL (Register Transfer Level) design, but few for high-level synthesis. Instead of applying Boolean algebra, this paper adopts the Petri Net theory to verify the correctness of the synthesis result, because the Petri Net model has the nature of dataflow algorithms. Herein, we propose three approaches to realize the Petri Net based formal verification algorithm and conclude the best one who outperforms the others in terms of processing speed and resource usage.

Keywords: Formal verification; High-level synthesis; Petri Net; Dataflow graph

1. Introduction

With increasing design complexity of digital signal processing system, verification becomes a more and more important within the design flow. In modern circuits, it is observed that up to 80% of the overall design costs are due to verification. Formal verification techniques which ensure 100% coverage of function and system model correctness have gained large attention. In Gupta (1992) and Kern and Greenstreet (1999), authors give excellent survey of major trends of formal verification techniques which can be classified into two categories, equivalence checking (Kern and Greenstreet, 1999) and model checking (Clarke et al., 1999). Equivalence checking is used to proof the functional equivalence of two design representations modeled at the same or different levels of abstraction. Model checking is a process that checks the correctness of a design model with given properties. Although formal verification for logic synthesis has been studied very extensively, little work has been done for high-level synthesis.

This paper presents a novel verification algorithm to verify high-level synthesis (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 (Madisetti and Curtis, 1994; Dung and Yang, 2004; Ito et al., 1998; Parhi, 1995; Chao and Sha, 1997). 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 paper 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

* Corresponding author.
E-mail addresses: aries.ece89g@nctu.edu.tw (T.-H. Chiang), lennon@cn.nctu.edu.tw (L.-R. Dung).

0164-1212/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2006.12.547

computing and hence a good tool for the verification of algorithmic transformations and datapath scheduling. The use of the Petri Net is twofolded. 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 is correct if and only if the initiation of each task is allowed in the Petri Net model. Comparing with the traditional model checking 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 flow graph (FSFG) (Madisetti, 1995). 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. Later, the matrix will be used to verify primary properties, such as reachability, liveness, safeness, and boundedness.

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 candidates to trace the firing sequence of the given schedule.

Based on the inputs, the proposed verification first extracts the initial firing pattern and uses it to determine the candidate reconstructed FSFGs. The candidates will then be verified with the Petri Net model. If there exist at least one candidate 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 paper, we propose three approaches to realize the PN-based formal verification and conclude the best one who outperforms the others in terms of processing speed and resource usage.

1.1. Related work

The existential verification methods utilize the technologies like BDD (Binary Decision Diagram) (Bryant, 1992; Brace et al., 1990), SAT (Satisfiability) solver (McMillan, 2002; Parthasarathy et al., 2001), symbolic model checking (Burch et al., 1991, 1994; Kang and Park, 2003; Parthasarathy et al., 2004) and theorem proving (Kljaich et al., 1989). These technologies are extremely powerful but must be applied in RTL level. Herein, in order to verify the HLS result, most literatures were focus on developing a strategy for RTL validation in which the equivalence between RTL level and its abstract level description. In Ashar et al. (1998) and Sarkar (2002), verification task is partitioning into two subtasks, verifying the validity of register sharing and verifying correct synthesis of the RTL interconnection and control. Similarly, in Karfa et al. (2006), Borrione et al. (2000), Mansouri and Vemuri (2000) and Bolchini et al. (2000), 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) (Gajski and Ramachandran, 1994). By applying such decomposition methods, a high-level scheduled design is divided into the control and the datapath. Thus, the equivalent checking technique can be applied to check the correctness of datapath, and the model checking can be used to verify the validity of control by utilizing the existential verification technologies.

In this paper, the proposed verification method is based on Petri Net for verifying the correctness of the high-level faults induced by applying algorithmic transform and scheduling processing in HLS. When comparing with the traditional approaches, this paper adopts the Petri Net theory to verify the correctness of the synthesis result instead of applying Boolean algebra on it.

1.2. Outline

The remainder of this paper is organized as follows. Section 2 describes some useful definition and proposed modeling technique. Section 3 presents the schedule representation and system specification to the FSFG design. The proposed high-level verification technique and verification algorithms are presented in Section 4. In Section 5, we discuss the complexity analysis of three verification algorithms and some experimental results. Section 6 gives the conclusions of this paper.

2. Definition and modeling

In this section, we will discuss some useful definitions and proposed transformation technique to transform a FSFG into PN model.

2.1. Fully-specified signal flow graph (FSFG)

Fully-Specified Signal Flow Graph (FSFG) (Madisetti, 1995) or DFG is a natural paradigm for describing DSP algorithms. A FSFG $G_{\mathrm{FSFG}}(V, E, D)$, where $V = \{v_1, \ldots, v_n\}$, $E = \{e_1, \ldots, e_m\}$ and $D \in \{0, 1, 2, \ldots\}$, is a three-tuple directed and edge-weighted graph which contains a vertex set V, a directed edge set E, and an ideal delay set D. Vertex set V represents atomic operation of functional units. A

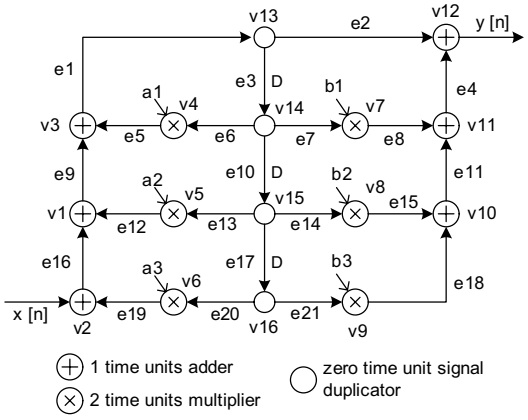


Fig. 1. A third-order IIR filter in the form of FSFG.

vertex may have a zero executing 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. Fig. 1, for instance, shows a third-order IIR filter in the form of FSFG.

2.2. Petri Net model

A Petri Net (PN) $G_{\mathrm{PN}}(P,T,W,M_0)$ is a four-tuple (Reisig and Rozenberg, 1998), 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. A marking is a function $M: P\to\mathbb{Z}$. If $M(p_i)=k$ for place p_i, we will say that p_i is marked with k tokens. 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 $W(u,v)$. 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 this paper, for each place $p_i\in P$, we only allow p_i has only one output transition, that is $\forall p_i\in P,\ |p_i^\bullet|=1$. A PN can *execute* by firing *enabled* transitions. A transition tr is *enabled* at marking M (denoted by $M[\mathrm{tr}\rangle$) if $\forall p\in{}^\bullet\mathrm{tr}: M(p)\geqslant W(p,\mathrm{tr})$. Once a transition tr is enabled at M, it may fire and then reach a new marking M' (denoted by $M[\mathrm{tr}\rangle M'$). The occurrence of tr lead to a new marking M', defined for each place p by

$$M'(p)=M(p)-W(p,\mathrm{tr})+W(\mathrm{tr},p). \tag{1}$$

A sequence of transitions $\sigma=\mathrm{tr}_1\ldots\mathrm{tr}_{k-1}\in T^*$ 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

$$M_i[\mathrm{tr}_i\rangle M_{i+1},\quad \text{for } 1\leqslant i\leqslant k-1. \tag{2}$$

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., $M'\in[M\rangle \iff \exists\sigma\in T^*: M[\sigma\rangle M'$. $[M_0\rangle$ 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

$$A_{ij}=W(\mathrm{tr}_j,p_i)-W(p_i,\mathrm{tr}_j). \tag{3}$$

The matrix representation usually gives a complete characterization of PN. Let $x_j=\{\mathrm{tr}_j\}=(\ldots,0,1,0,\ldots)$ be the unit $|T|\times 1$ column vector which is zero everywhere except in the jth 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 $M_0[\mathrm{tr}_j\rangle$) 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

$$\mu'=\delta(\mu_0,\mathrm{tr}_j)=\mu_0+A\cdot x_j. \tag{4}$$

For a sequence of transition firing $\sigma: M_0[\sigma\rangle M_k$ and $M_i[\ \mathrm{tr}_i\rangle M_{i+1}$, $1\leqslant i\leqslant k-1$, we have

$$\begin{aligned}\delta(\mu_0;\sigma)&=\delta(\mu_0;\mathrm{tr}_1\mathrm{tr}_2\mathrm{tr}_3\ldots\mathrm{tr}_{k-1})\\&=\mu_0+\sum_1^{k-1}A\cdot x_j=\mu_0+A\cdot f(\sigma).\end{aligned} \tag{5}$$

The vector $f(\sigma)$ is *firing vector* of the sequence $\sigma=\mathrm{tr}_1\ldots\mathrm{tr}_{k-1}$. The ith element of $f(\sigma)$, $f(\sigma)_i\in\mathbb{Z}$, is the number of times that transition tr_i fires in the σ.

2.3. Transformation 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 dynamical 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 dynamical behavior of the algorithm.

The FSFG $G_{\mathrm{FSFG}}(V,E,D)$ of a DSP algorithm can be modeled as PN $G_{\mathrm{PN}}(P,T,W,M_0)$ 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 token in place in PN domain. In static analysis, tokens in an PN model can represent delay elements of an FSFG; thus, moving tokens between places in a PN model can be seen

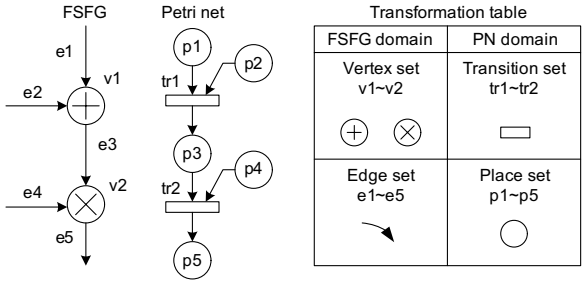


Fig. 2. Transformation from FSFG to Petri Net.

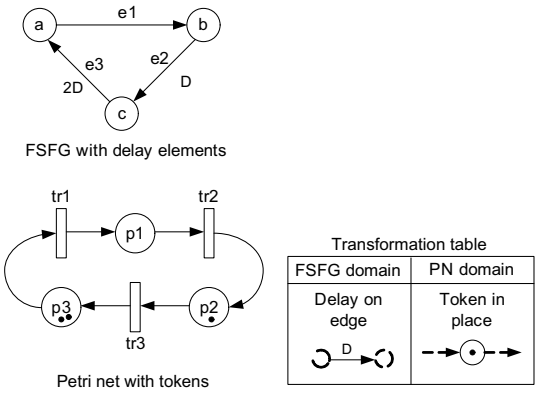


Fig. 3. FSFG with delay elements and Petri Net with tokens.

as retiming delay elements between edges in an FSFG. In dynamical analysis, moving tokens between places in an PN model represents the executions of vertex elements in an FSFG.

Fig. 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 $T=\{\mathrm{tr}_1,\mathrm{tr}_2\}$ 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 Fig. 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 $T=\{\mathrm{tr}_1,\mathrm{tr}_2,\mathrm{tr}_3\}$ 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 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 Fig. 4. Another example illustrating the PN model of the third-order IIR filter of Fig. 1 by applying

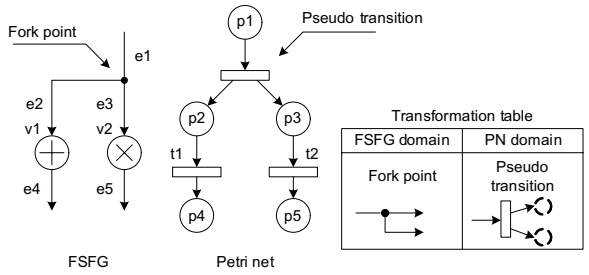


Fig. 4. Fork point in FSFG domain and pseudo-transition in PN domain.

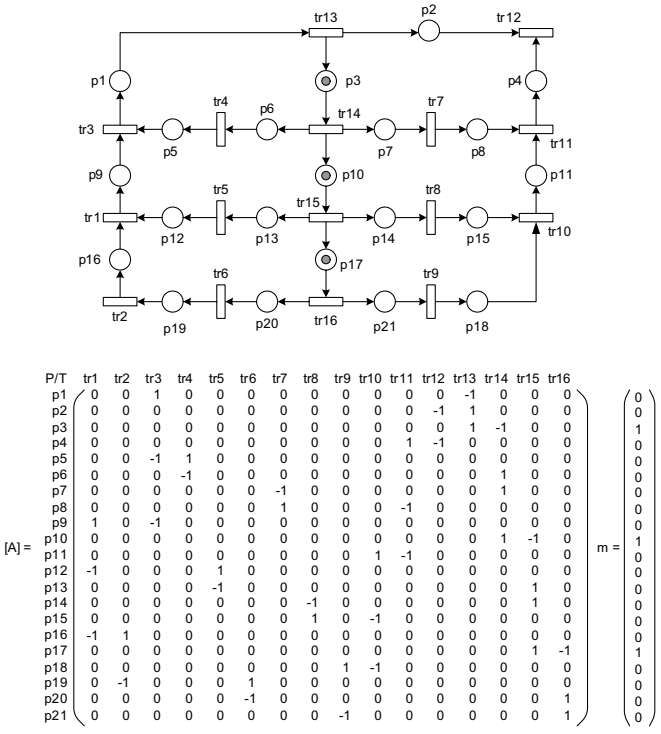


Fig. 5. A PN graph and the matrix representation to the third-order IIR filter of Fig. 1.

above transformation rules is showing in Fig. 5 while the characteristic matrix A with the initial marking m shows the matrix representation of the PN model.

3. Schedule and system specification

In this section, we describe the representation of schedule to a FSFG design and its PN model. The system specification, which dominates the correctness of the given schedule, is also presented.

3.1. Schedule to the FSFG

In HLS, a FSFG design may contain cycles to model a DSP application with loops. The intra-iteration precedence relation is represented by the edge without delay and the inter-iteration precedence relation is represented by the edge with delays. Given an edge $e(v_i,v_j)\in E$ in FSFG

design, $d(e)$ means the data used as inputs in node v_j are generated by node v_i at $d(e)$ inter-iteration before. 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 $V=\{v_i|i=1,2,\ldots,n\}$ is in one-to-one correspondence with the set of the FSFG design, and edge set $E=\{(v_i,v_j)|i,j=1,2,\ldots,n;\ i\neq j\}$ is representing their dependencies. An example of sequencing graph is showing upper in Fig. 6a. Different scheduling algorithms have been proposed in Parhi and Messerschmitt (1991), Chao and Sha (1997) and Hwang et al. (1991) addressing different constrained problems to find the desired schedule. The desired schedule have to satisfy the precedence constraints specified by the sequencing graph.

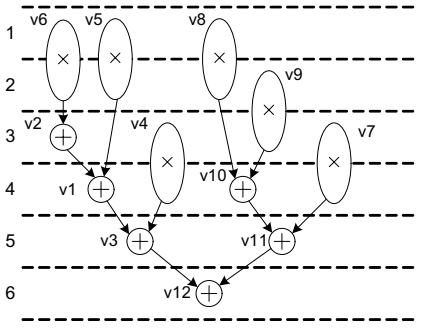
A schedule S to the FSFG design is represented in space–time ($P\times T$) domain. The *abscissa* denotes time axis, $[1,\mathrm{le}(S)]$, where $\mathrm{le}(S)$ is the length of the schedule. The *ordinate* denotes the processor space, $[1,n_{\mathrm{res}}]$, where n_{res} is the total number of processors used for the scheduling. During the period of the ith iteration, schedule determines the start times of all nodes in FSFG. Let op_j^i be one of the task, which is corresponded to the vertex $v_j\in V$ of a FSFG in the ith iteration. The start time of each task is a mapping $\varphi: V\rightarrow \mathbb{Z}^+$, which arranges each task node op_j^i to begin its execution at the time step $\varphi(\mathrm{op}_j^i)$, where $\mathbb{Z}^+=\{1,2,\ldots\}$ is the positive integers. The task assignment function $\tau: V\rightarrow\{1,2,\ldots,n_{\mathrm{res}}\}$ is a mapping from each task node op_j^i to a single processor $\tau(\mathrm{op}_j^i)$.

Let d_j^i be the executing delay for each task node op_j^i, the length $\mathrm{le}(S)$ of a schedule S is the latest finish time of all the operations scheduled, that is $\mathrm{le}(S)=\max\{\varphi(\mathrm{op}_j^i)+d_j^i-1\mid\forall\mathrm{op}_j^i\in V\}$. For each task node $\mathrm{op}_j^i\in V$, a schedule of the FSFG design is given as following:

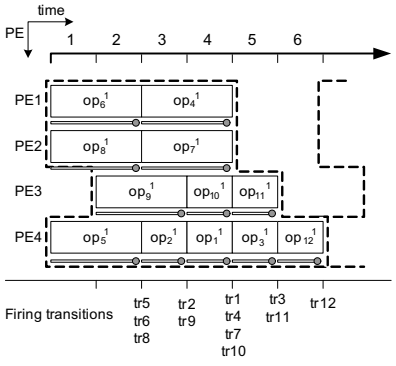
- Start time: $t_j^i=\varphi(\mathrm{op}_j^i),\varphi: V\rightarrow\mathbb{Z}^+=\{1,2,\ldots\}$.
- Executing delay: $d_j^i\in\mathbb{Z}=\{0,1,2,\ldots\}$.
- Finish time: $\varepsilon_j^i=\varphi(\mathrm{op}_j^i)+d_j^i$.
- Task assignment: $\mathrm{pe}_j^i=\tau(\mathrm{op}_j^i),\tau: V\rightarrow\{1,2,\ldots,n_{\mathrm{res}}\}$.
- Length of the schedule: $\mathrm{le}(S)=\max\{\varphi(\mathrm{op}_j^i)+d_j^i-1\mid\forall\mathrm{op}_j^i\in V\}$.
- The earliest task-finished step: $t_{\mathrm{etf}}=\min\{\varphi(\mathrm{op}_j^i)+d_j^i-1\mid\forall\mathrm{op}_j^i\in V\}$.

3.2. Execution of a task

As mentioned in previous section, 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.



(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

Fig. 6. A sample schedule of the third-order IIR filter in Fig. 1.

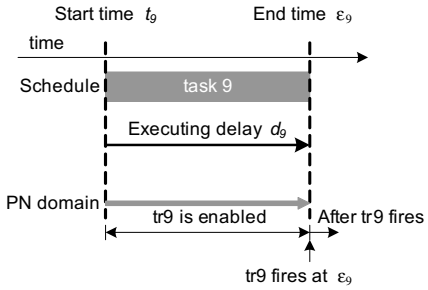


Fig. 7. Executing duration of a task.

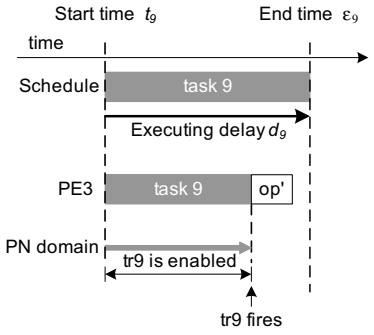


Fig. 8. Task 9 is preempted by operation node op′.

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 Fig. 7, 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 schedule is showing in Fig. 6c. At each step of the schedule, enabled transitions may fire at the end of each step. The tasks schedule and its firing transitions are illustrated.

3.3. 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 ith iteration of S. The properties to the system are defined as following. We also describe the false-cause and its detection 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 ith 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:

$$\mathrm{PE}_k(u) = \mathrm{pe}_j^i, \quad \mathrm{pe}_j^i = \tau(\mathrm{op}_j^i), \quad t_j^i \leqslant u < t_j^i + d_j^i, \tag{6}$$

where $\mathrm{PE}_k(u)$ is the assignment function for resource k, $1 \leqslant u \leqslant \mathrm{le}(S)$.

- *Fault-cause*: In Fig. 8, 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.
- *Detection*: At the sth step of schedule length $\mathrm{le}(S)$, the marking state μ_s to the sth 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

$$\mu_s(p) > 0, \quad \forall p \in {}^\bullet \mathrm{tr}_j, \ 1 \leqslant s \leqslant \mathrm{le}(S). \tag{7}$$

(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:

$$t_j^i > 0, \quad t_j^i = \varphi(\mathrm{op}_j^i), \qquad 1 \leqslant i \leqslant f, \quad \forall \mathrm{op}_j^i \in V. \tag{8}$$

- *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 sth step of S. Let firing vector κ_s be a $(|T| \times 1)$-column vector. If the jth 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 sth 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 $\mathrm{le}(S)$. On the other word,

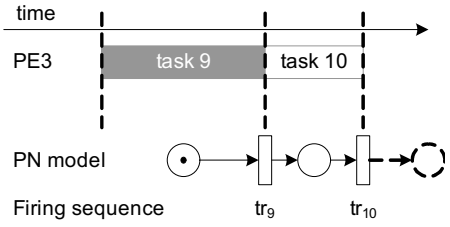


Fig. 9. Precedence property of two operation tasks.

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

$$t_k^i \geqslant t_j^i + d_j^i, \tag{9}$$

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 Fig. 9, transition tr10 is a successor of transition tr9, thus the execution order of these two operation nodes must ensure tr9 → tr10. Schedule S violates precedence property if these two operation nodes execute in reverse order.
- *Detection*: At the sth step of S, the firing vector κ_s is a $(|T| \times 1)$-column vector. If the jth element of the firing vector is $\kappa_s(j) = 1$, it indicates the operation node op_j^i fires at the sth step. Let μ_s be the marking state at the sth step of S, next marking μ_{s+1} can be obtained by using Eq. (4), that is

$$\mu_{s+1} = \mu_s + A \cdot \kappa_s, \quad 1 \leqslant s \leqslant le(S). \tag{10}$$

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.

4. 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.

4.1. Verification flow

A flowchart illustrating our verification flow is shown in Fig. 10. 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

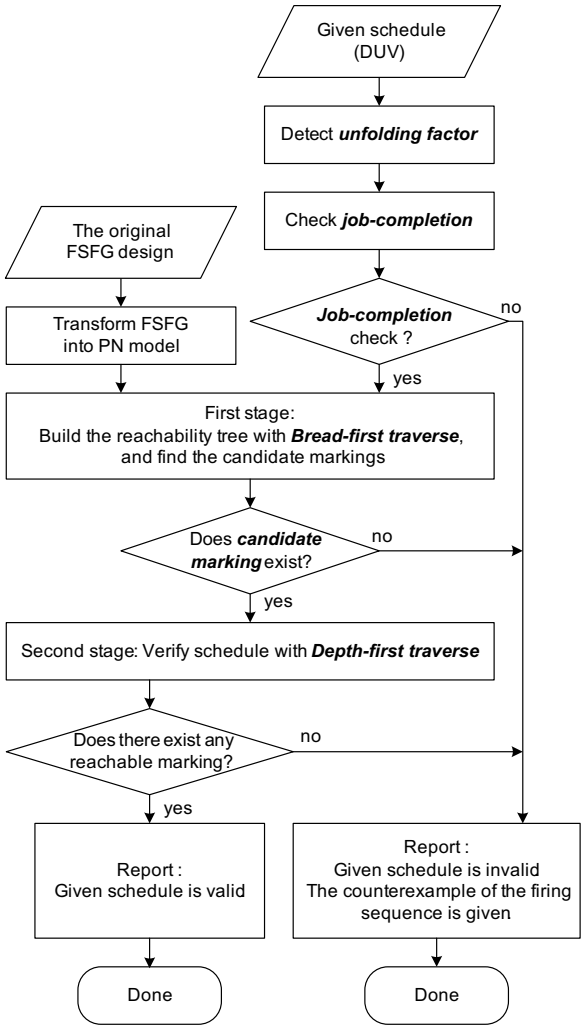


Fig. 10. Flowchart for the proposed high-level verification method.

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.

4.2. 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

$$etf\_set = \{\text{op}_j^i \mid \varepsilon_j^i = t_{\text{etf}}, \forall \text{op}_j^i \in V\}. \tag{11}$$

Assuming m is a marking of the PN model. A firing sequence $\sigma : \text{tr}_1 \dots \text{tr}_k$ of transitions is denoted by $m \xrightarrow{\sigma} m_k$, where $m_1 \dots m_k$ are valid states, such that

$$m \xrightarrow{\text{tr}_1} m_1 \xrightarrow{\text{tr}_2} \cdots \xrightarrow{\text{tr}_k} m_k. \tag{12}$$

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 $\sigma : \text{tr}_1 \dots \text{tr}_k$, such that for all tasks $\text{op} \in etf\_set$ are covered by all the transitions in σ, i.e., $etf\_set \subseteq \sigma$. And it is also satisfied that each firing transition $\text{tr}_j \in \sigma$ is either a pseudo-transition, $d_j = 0$, or an earliest task-finished transition, $\varepsilon_j = t_{\text{etf}}$.

As an example, Fig. 11 shows the procedure to check whether a marking is candidate. For a given schedule in Fig. 12, the earliest task-finished step is $t_{\text{etf}} = 1$, and the earliest task-finished set is $etf\_set = \{\text{op}_j^i \mid \varepsilon_j^i = t_{\text{etf}} = 1\} = \{v_5, v_9\}$. Marking $m = [00000112000000]$ is said to be a ***candidate marking*** of the corresponded PN model. Since,

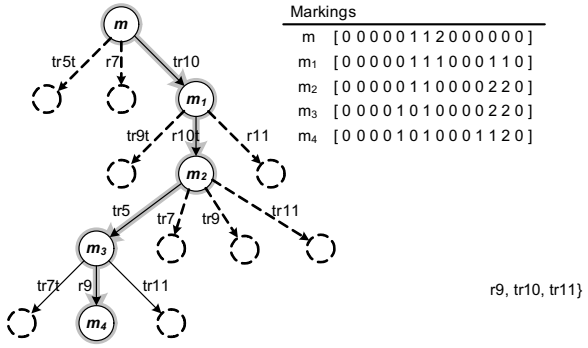


Fig. 11. Check whether a marking is a candidate marking.

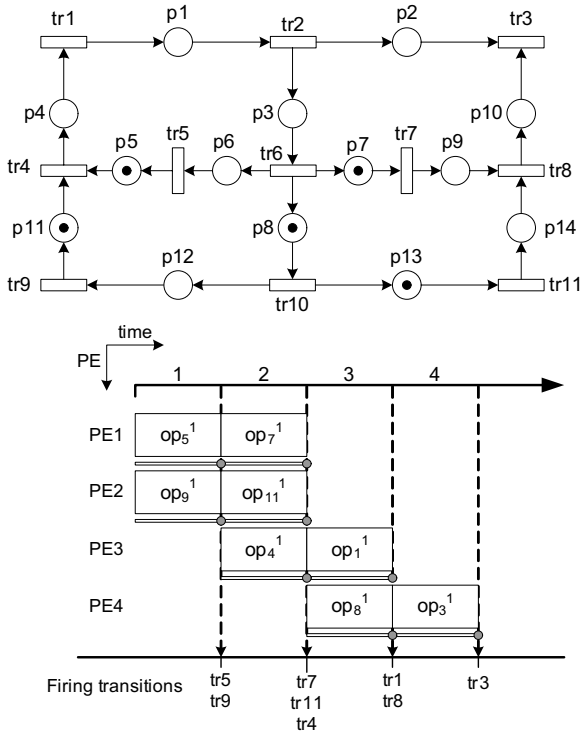


Fig. 12. An example schedule and second order IIR filter.

there exists a firing sequence $\sigma : \text{tr}_{10}\text{tr}_{10}\text{tr}_5\text{tr}_9$, $m \xrightarrow{\sigma} m_4$, such that $etf\_set \subseteq \sigma$. The markings, $m_1 \dots m_k$, of the firing sequence, $m \xrightarrow{\text{tr}_{10}} m_1 \xrightarrow{\text{tr}_{10}} m_2 \xrightarrow{\text{tr}_5} m_3 \xrightarrow{\text{tr}_9} m_4$, are valid states, where $m_1 = [00000111000110]$, $m_2 = [00000110000220]$, $m_3 = [00001010000220]$, and $m_4 = [00001010001120]$.

4.3. 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 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.

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 $H_{\mathrm{up}} = f \cdot n$. 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 Fig. 13, 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 $S2$.

4.4. 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 paper and discussed in the following sections.

4.4.1. 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, Fig. 14 shows a reachability tree of the Petri Net in Fig. 12. In Fig. 14, 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 $H_{\mathrm{up}} = f \cdot n$ for a finite n-tasks schedule. The pseudo-code of the exhaustive approach is shown in Fig. 15.

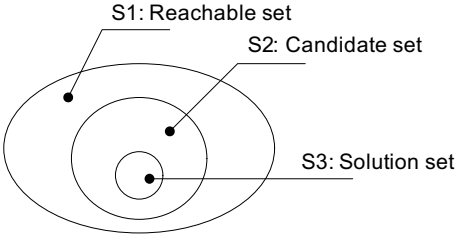
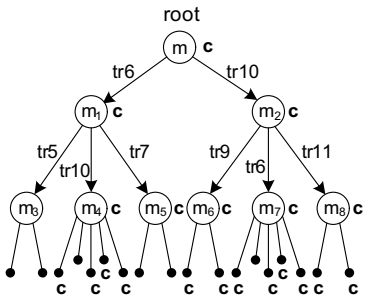


Fig. 13. The relation between reachable, candidate and solution marking sets.



#.	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}

Fig. 14. Build reachability tree with Breadth-First search algorithm.

In the beginning of Fig. 15, 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 10–27.

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 Fig. 14, the traverse order of the reachability tree which is rooted by marking m is $m, m_1, m_2, \ldots, m_8, \ldots$

```
1: procedure BFS_BUILD_TREE_EARLY_TERMINATED(μ0)   ▷ root marking μ0
2:     Initialize Queue structure Q
3:     Allocate new node nn                          ▷ initialize root node
4:     nn.visited ← false
5:     nn.marking ← μ0
6:     nn.height ← 0
7:     nn.candidate ← IS_CANDIDATE(nn)
8:     Q.ENQUEUE(nn)
9:
10:    for all unvisited node n ∈ Q do
11:        n.visit ← true
12:        if n.candidate = true then
13:            continue to the next node          ▷ Early-terminate Lemma 1
14:        end if
15:        if n.height ⩽ H then
16:            ebl_set ← FIND_ENABLED_TRANS(n.marking)
17:            for all transition tr ∈ ebl_set do
18:                μ ← n.marking
19:                κ ← MAKE_FIRING_VECTOR_FROM(tr)
20:                μ′ ← μ + [A] · κ                          ▷ Eq.4
21:                Allocate new node nn
22:                nn.marking ← μ′
23:                nn.visited ← false
24:                nn.height ← n.height + 1
25:                nn.candidate ← IS_CANDIDATE(nn)
26:                CREATE_BRANCH(n, nn)
27:                Q.ENQUEUE(nn)
28:            end for
29:        end if
30:    end for
31: end procedure
```

Fig. 15. The exhaustive approach.

4.4.2. 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 1.

Lemma 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 1. 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 $S3$. □

The early-terminated approach is based on the exhaustive approach and uses Lemma 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 Fig. 16, 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 Fig. 17. In lines 12–14, it ignores the candidate marking and proceeds other unvisited nodes.

4.4.3. 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.

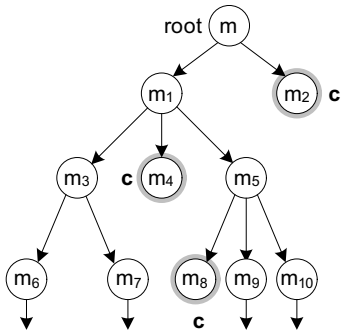


Fig. 16. The traverse order of early-terminated approach.

Let m be an unvisited node to be processed. If m is a candidate marking, it ignores this node by using Lemma 1 and proceeds other unvisited nodes in 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 Fig. 18, 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.

The pseudo-code of the optimal approach is shown in Fig. 19. In lines 12–14, if the node n is a candidate marking, then it ignores this node by using Lemma 1 and proceeds the other nodes in queue Q. In lines 26–32, function

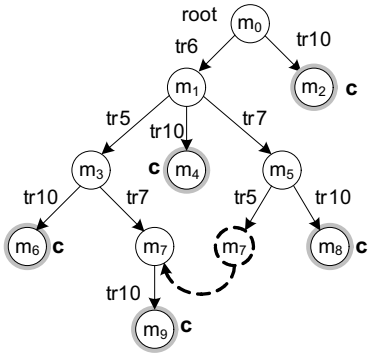


Fig. 18. Merge the redundant node in optimal traverse approach.

```
1:  procedure BFS_BUILD_TREE_EARLY_TERMINATED(μ0)   ▷ root marking μ0
2:      Initialize Queue structure Q
3:      Allocate new node nn                          ▷ initialize root node
4:      nn.visited ← false
5:      nn.marking ← μ0
6:      nn.height ← 0
7:      nn.candidate ← IS_CANDIDATE(nn)
8:      Q.ENQUEUE(nn)
9:
10:     for all unvisited node n ∈ Q do
11:         n.visit ← true
12:         if n.candidate = true then
13:             continue to the next node          ▷ Early-terminate Lemma 1
14:         end if
15:         if n.height ⩽ H then
16:             ebl_set ← FIND_ENABLED_TRANS(n.marking)
17:             for all transition tr ∈ ebl_set do
18:                 μ ← n.marking
19:                 κ ← MAKE_FIRING_VECTOR_FROM(tr)
20:                 μ′ ← μ + [A] · κ                  ▷ Eq.4
21:                 Allocate new node nn
22:                 nn.marking ← μ′
23:                 nn.visited ← false
24:                 nn.height ← n.height + 1
25:                 nn.candidate ← IS_CANDIDATE(nn)
26:                 CREATE_BRANCH(n, nn)
27:                 Q.ENQUEUE(nn)
28:             end for
29:         end if
30:     end for
31: end procedure
```

Fig. 17. The early-terminated approach.

```
1: procedure BFS_BUILD_TREE_OPTIMAL(μ0)                    ▷ root marking μ0
2:     Initialize Queue structure Q
3:     Allocate new node nn                                 ▷ initialize root node
4:     nn.visited ← false
5:     nn.marking ← μ0
6:     nn.height ← 0
7:     nn.candidate ← IS_CANDIDATE(nn)
8:     Q.ENQUEUE(nn)
9:
10:    for all unvisited node n ∈ Q do
11:        n.visit ← true
12:        if n.candidate = true then
13:            continue to the next node                    ▷ Early-terminate Lemma 1
14:        end if
15:        if n.height ⩽ H then                             ▷ max. height H
16:            ebl_set ← FIND_ENABLED_TRANS(n.marking)
17:            for all transition tr ∈ ebl_set do
18:                μ ← n.marking
19:                κ ← MAKE_FIRING_VECTOR_FROM(tr)
20:                μ′ ← μ + [A] · κ                         ▷ Eq.4
21:                Allocate new node nn
22:                nn.marking ← μ′
23:                nn.visited ← false
24:                nn.height ← n.height + 1
25:                nn.candidate ← IS_CANDIDATE(nn)
26:                dual_node ← FIND_DUPLICATE_NODE(nn)
27:                if dual_node ≠ 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
```

Fig. 19. The optimal traverse approach.

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.

4.5. 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 Fig. 20, a candidate marking ***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 Fig. 12, 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 $\sigma : tr_6tr_5tr_{10}tr_9$, that is $m \xrightarrow{\sigma} m_1$, where transition tr_6 and

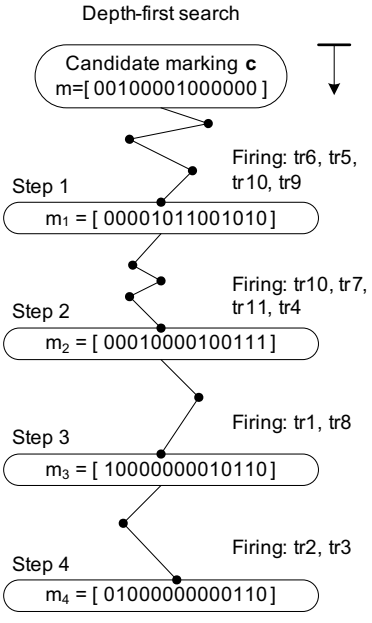


Fig. 20. Verify schedule with Depth-First search algorithm.

```
1: procedure DFS_TRAVERSE_PATH(Q)                          ▷ queue structure Q
2:     Initialize schedule structure sch
3:     Mark all node n ∈ Q unvisited
4:
5:     for all unvisited candidate node n ∈ Q do
6:         n.visit ← true
7:         μ ← n.marking
8:         for all control step s ∈ [1, 2, ..., sch.max_step] do
9:             if there is no task end at step s then
10:                continue to next s
11:            end if
12:            valid ← GO_FURTHER_DEPTH_FIRING(s, μ, μ′)
13:            if valid = true then                  ▷ depth firing is valid at step s
14:                μ ← μ′
15:            else                                  ▷ depth firing is not valid at step s
16:                break
17:            end if
18:        end for
19:        if valid = true then
20:            sol_set ← {sol_set ∪ n.marking}                          ▷ solution set
21:        end if
22:    end for
23: end procedure
```

Fig. 21. Path traverse with Depth-First search algorithm.

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 Fig. 21. 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'. The pseudo-code for procedure go_further_depth_firing is illustrated in Fig. 22, and 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 19–21, all solution markings are added to *sol_set*.

5. The complexity analysis and result

Assuming there are n non-nop operations in a given FSFG, thus there are n non-nop transitions in the corresponded PN model. 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
. . . . . .
Level $f \cdot n$ has $(n^{f \cdot n-1})(n) = n^{f \cdot n}$ nodes

The total number of nodes is

$$1 + n + n^2 + \cdots + n^{f \cdot n} = (n^{f \cdot n+1} - 1)/(n - 1). \quad (13)$$

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, $p \leqslant (f \cdot n)$, be the deepest level that Breadth-First traverse procedure can reach. The complexity of the second approach is $O(N^p)$, $p \leqslant f \cdot n$.

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 $x \in \mathbb{Z} = \{1, 2, \ldots\}$ be the merging radio in the reachability tree. The complexity of the three approach is $O((N/x)^p)$, $p \leqslant f \cdot n$. Thus, the relation of the complexity between three approaches is:

$$O(N^{f \cdot n}) > O(N^p) > O((N/x)^p). \quad (14)$$

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.

```
1: procedure GO_FURTHER_DEPTH_FIRING(s, μ, out_mark)
2:     tran_set ← FIND_ALL_TASKS_FINISHED_AT_STEP(s)
3:
4:     cm ← μ                                          ▷ current marking cm
5:     violate ← false
6:     while ({tran_set} is not empty) and (violate = false) do
7:         fired ← false
8:         ebl_set ← FIND_ENABLED_TRANS(cm)
9:         for i ← 1, |T| do                  ▷ total number of transitions |T|
10:            tr ← the ith transition
11:            if tr ∈ {ebl_set} ∩ {nop} then                ▷ fire nop transition
12:                μ′ ← FIRE_TRANSITION(cm, tr)
13:                fired ← true
14:            else if tr ∈ {ebl_set} ∩ {tran_set} then ▷ fire tran_set transition
15:                tran_set ← {tran_set} \ tr          ▷ delete tr from tran_set
16:                μ′ ← FIRE_TRANSITION(cm, tr)
17:                fired ← true
18:            end if
19:            if fired = true then                       ▷ one firing each time
20:                break
21:            end if
22:        end for                                              ▷ next transition
23:        if fired = false then                  ▷ there is no firing occurred
24:            violate ← true
25:        else if IS_MARKING_VALID(μ′) = false then  ▷ marking is not valid
26:            violate ← true
27:        else
28:            cm ← μ′                                        ▷ next iteration
29:        end if
30:    end while
31:    if violate ← false then
32:        out_mark ← cm                    ▷ out_mark is the output marking
33:        return true
34:    else
35:        return false
36:    end if
37: end procedure
```

Fig. 22. The pseudo-code for procedure go_further_depth_firing of DFS_traverse_path in Fig. 21.

Fig. 23 shows the experimental results of using three approaches. The optimal approach outperforms the others in terms of time and resource usage.

Test schedule	Exhaustive		Early-terminated		Optimal	
	Time (sec)	Res. usage	Time (sec)	Res. usage	Time (sec)	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

Fig. 23. The experimental results.

6. Conclusion

This paper 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 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.

Acknowledgement

This work was supported by the National Science Council, ROC, under the grant number NSC 94-2220-E-009-039.

References

Ashar, P., Bhattacharya, S., Raghunathan, A., Mukaiyama, A., 1998. Verification of rtl generated from scheduled behavior in a high-level synthesis flow. In: Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design, pp. 517–524.

Bolchini, C., Montandon, R., Salice, F., Sciuto, D., 2000. Design of VHDL-based totally self-checking finite-state machine and data-path descriptions. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 8 (1), 98–103.

Borrione, D., Dushina, J., Pierre, L., 2000. A compositional model for the functional verification of high-level synthesis results. IEEE Transactions on VLSI Systems, 526–530.

Brace, K.S., Rudell, R.L., Bryant, R.E., 1990. Efficient implementation of a BDD package. ACM/IEEE Design Automation Conference, 40–45.

Bryant, R.E., 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24 (3), 293–318.

Burch, J., Clarke, E., Long, D., 1991. Symbolic model checking with partitioned transition relations. In: International Conference on Very Large Scale Integration, pp. 49–58.

Burch, J., Clarke, E., Long, D., MacMillan, K., Dill, D., 1994. Symbolic model checking for sequential circuit verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13 (4), 401–424.

Chao, L.-F., Sha, E.H.-M., 1997. Scheduling data-flow graphs via retiming and unfolding. IEEE Transactions on Parallel and Distributed Systems 8 (12), 1259–1267.

Clarke, E.M., Grumberg, O., Peled, D.A., 1999. Model Checking. The MIT Press.

Dung, L.-R., Yang, H.-C., 2004. On multiple-voltage high-level synthesis using algorithmic transformations. IEICE Transactions on Fundamentals.

Gajski, D.D., Ramachandran, L., 1994. Introduction to high-level synthesis. IEEE Design and Test 11 (4), 44–54.

Gupta, A., 1992. Formal hardware verification methods: a survey. Formal Methods in System Design 1, 151–238.

Hwang, C., Lee, J., Hsu, Y., 1991. A formal approach to the scheduling problem in high level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 10 (April), 464–475.

Ito, K., Lucke, L.E., Parhi, K.K., 1998. Ilp-based cost-optimal dsp synthesis with module selection and data format conversion. IEEE Transactions on Very Large Integration Systems 6 (4), 582–594.

Kang, H.-J., Park, I.-C., 2003. SAT-based unbounded symbolic model checking. In: Proceedings of the 40th Conference on Design Automation, pp. 840–843.

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

Kern, C., Greenstreet, M., 1999. Formal verification in hardware design: a survey. ACM Transactions on Design Automation of E. Systems 4 (April), 123–193.

Kljaich, J., Smith, B.T., Wojcik, A.S., 1989. Formal verification of fault tolerance using theorem-proving techniques. IEEE Transactions on Computers 38 (3), 366–376.

Madisetti, V.K., 1995. VLSI Digital Signal Processors. IEEE Press.

Madisetti, V.K., Curtis, B.A., 1994. A quantitative methodology for rapid prototyping and high-level synthesis of signal processing algorithms. IEEE Transactions on Signal Processing 32 (11), 3188–23208.

Mansouri, N., Vemuri, R., 2000. Automated correctness condition generation for formal verification of synthesized RTL designs. Journal of Formal Methods in System Design 16 (1).

McMillan, K.L., 2002. Applying SAT methods in unbounded symbolic model checking. In: 14th Conference on Computer Aided Verification, pp. 250–264.

Parhi, K.K., 1995. High-level algorithm and architecture transformations for dsp synthesis. Journal of VLSI Signal Processing 9, 121–143.

Parhi, K., Messerschmitt, D., 1991. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. IEEE Transactions on Computers 40 (2), 178–195.

Parthasarathy, G., Cheng, K.-T., Huang, C.-Y., 2001. An analysis of ATPG and SAT algorithms for formal verification. In: Proceedings of International High Level Design Validation and Test Workshop, pp. 177–182.

Parthasarathy, G., Iyer, M.K., Cheng, K.-T., Wang, L.-C., 2004. Safety property verification using sequential SAT and bounded model checking. IEEE Design and Test of Computers 21 (2).

Reisig, W., Rozenberg, G., 1998. Lectures on Petri Nets I: Basic Models. Springer-Verlag.

Sarkar, D., 2002. Register transfer operation analysis during data path verification. In: Proceedings of the 2002 Conference on Asia South Pacific Design automation/VLSI Design, p. 172.