國立交通大學

電機學院IC設計產業研發碩士班

碩士論文

高階合成中使用臆測編碼轉換技術之排程等效驗證

Equivalence Checking of Scheduling with Speculative Code Transformations in High-Level Synthesis

研究生：李季慧

指導教授：周景揚　教授

中華民國九十九年一月

高階合成中使用臆測編碼轉換技術之排程等效驗證

Equivalence Checking of Scheduling with Speculative Code Transformations in High-Level Synthesis

研 究 生：李季慧　　Student：Chi-Hui Lee

指導教授：周景揚 教授　　Advisor：Prof. Jing-Yang Jou

國 立 交 通 大 學

電機學院 IC 設計產業研發碩士班

碩 士 論 文

A Thesis

Submitted to College of Electrical and Computer Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Industrial Technology R & D Master Program on

IC Design

January 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年一月

高階合成中使用臆測編碼轉換技術之排程等效驗證

學生：李季慧　　指導教授：周景揚 教授

國立交通大學電機學院產業研發碩士班

摘　要

高階合成是一個將演算法層次的描述轉換成暫存器轉移層次設計的程序，它可以提昇生產力。然而這轉換程序容易錯誤。排程，高階合成的子工作，是驗證高階合成中最大的挑戰，因為排程會改變原來的執行順序。這篇論文的研究主題是排程驗證。我們提出一個正規方法可用來驗證排程前後的描述是否相等。排程前後的描述由包含資料路徑的有限狀態機所表示。它們一開始先被分解成有限條路徑；接著，在這些路徑中找到相等的路徑。兩個包含資料路徑的有限狀態機的相等和兩條路徑的相等在這篇論文中被定義。提出的方法不只適合驗證保留控制架構的排程，也適合驗證會改變控制架構的排程。排程改變控制架構藉由合併連續的路徑或將某些程式碼在不同的基本塊中移動。由驗證高階合成的測試程式的實驗結果顯示我們演算法可以有效地驗證排程。

Equivalence Checking of Scheduling with Speculative Code Transformations in High-Level Synthesis

Student：Chi-Hui Lee　　Advisors：Prof. Jing-Yang Jou

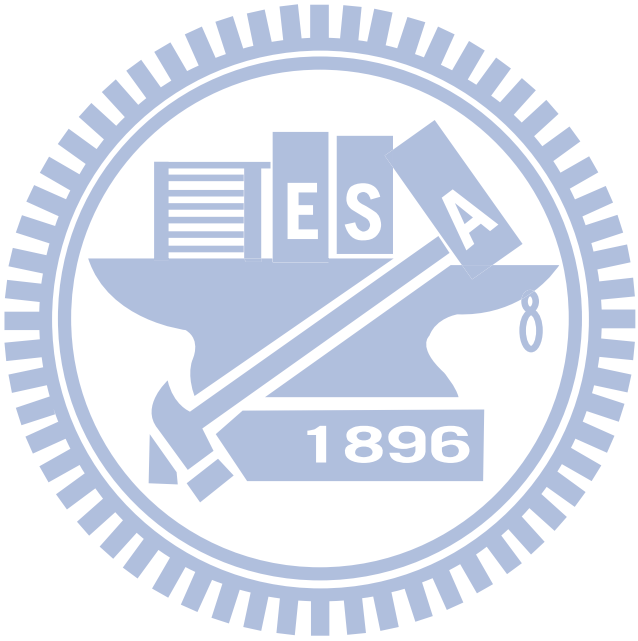
Industrial Technology R & D Master Program of
Electrical and Computer Engineering College
National Chiao Tung University

ABSTRACT

High-level synthesis (HLS) is a process of generating a register-transfer level design from an algorithm level description. It can increase the design productivity. However, this translation process can be buggy. Scheduling, a sub-task of HLS, makes the major challenge of the HLS verification since it usually changes the original cycle-by-cycle behavior. This thesis focuses on the scheduling verification. A formal method for checking equivalence between the descriptions before and after scheduling is described. Finite state machine with datapaths (FSMDs) are used to represent both descriptions. Two FSMDs are both decomposed into finite paths, and the method finds equivalent paths between them. The equivalence of FSMDs and paths are also defined. The proposed method is suited to verify not only the scheduling preserving the control structure but also the scheduling changing the control structure by merging some consecutive paths or moving some codes across the boundaries of the basic blocks. The experimental results on several HLS benchmarks show the effectiveness of the proposed algorithm.

Acknowledgements

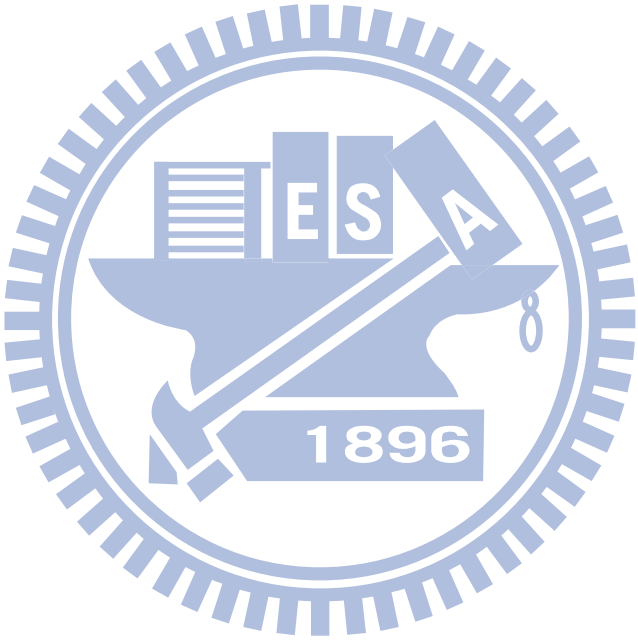
I greatly appreciate my advisors, Dr. Jing-Yang Jou and Dr. Juinn-Dar Huang, for their patient guidance, valuable suggestion, and encouragement during these years. I am also grateful to Che-Hua Shih and Wan-Hsien Len for their discussion and help on my research. Specially thank to all member of EDA Lab and Adar Lab for their friendship and company. Finally, I would like to express my sincerely acknowledgements to my family and my friends for their patient and support.



Contents

List of Figures

List of Tables



Chapter 1
Introduction

High-level synthesis (HLS), also called behavioral synthesis, is a process that converts a behavior or algorithm description into an RTL (register-transfer level) circuit [1]-[3]. It is helpful to gain much higher productivity than RTL or logic synthesis. In Fig. 1, HLS consists of the following sub-tasks:

- Intermediate description generation

  This task compiles a behavior description into an internal representation such as a control data flow graph (CDFG) which captures all the control and data-flow dependencies of the given behavioral description [1].

- Scheduling

  Scheduling assigns operations of the behavior description to specific control steps or clock cycles under data-dependencies and constraints.

- Allocation and binding

  Allocation and binding specify operations to functional units, and assign data to storage elements and interconnect units.

- Architecture generation

  This task builds a controller (a finite state machine, FSM) to control the data-path, depending on the information of scheduling, allocation and binding.

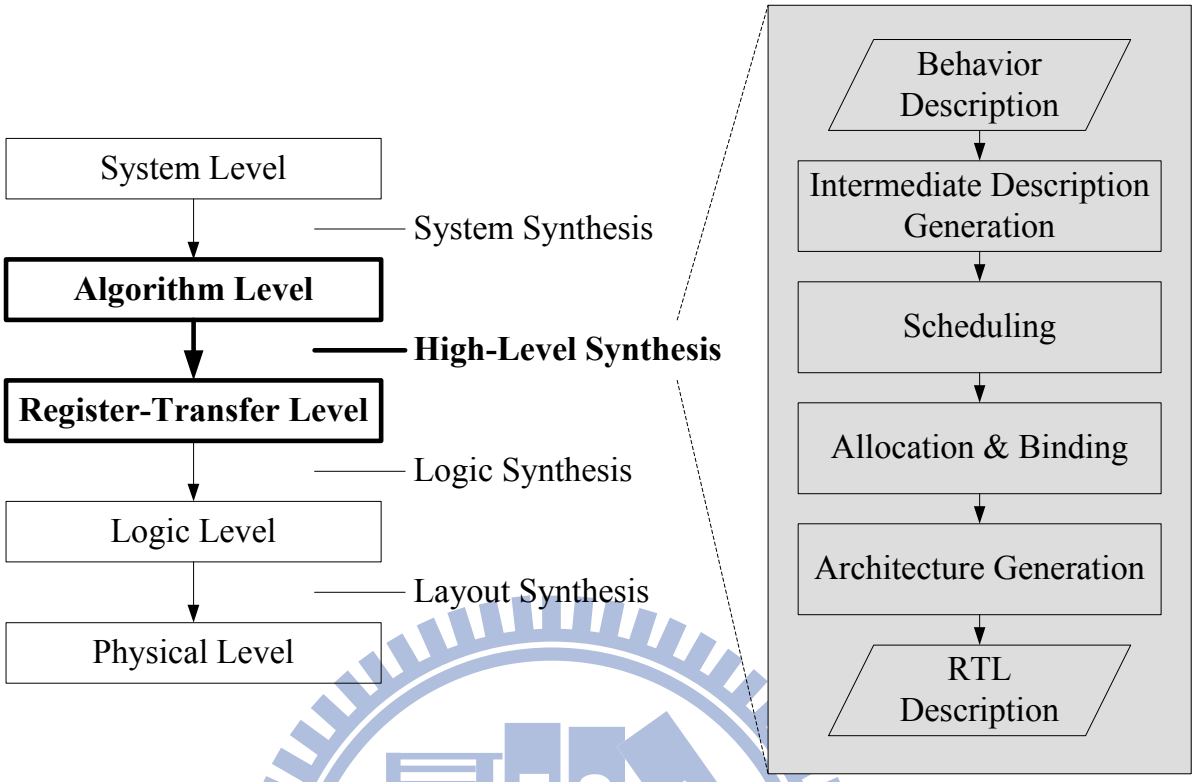


Fig. 1 An Overview of HLS

Obviously, the correctness of the HLS process is very important for the development of HLS. In HLS, scheduling usually changes the original cycle-by-cycle behavior. Moreover, many scheduling techniques may even change the control structure. Hence, most of scheduled results can not be one-to-one mapped to their original structures. Therefore, scheduling becomes the main challenge in the HLS verification. As a result, we focus on the scheduling verification. The following sections give an overview of the scheduling methodologies in HLS and a review of many previous researches of the scheduling verification.

1.1 Scheduling in HLS

Scheduling is an important task of HLS. It assigns operations of a behavior description to control steps or clock cycles under some given constraints on area or delay. Thus it impacts the tradeoff between the design cost and the performance. Scheduling algorithms, traditionally,

can be classified into two categories: data-flow-based and control-flow-based algorithms.

Most of early scheduling algorithms are data-flow-based (DF-based) or basic-block-based (BB-based) algorithms. BB-based algorithms focus on taking advantage of parallelism among sequences of operations in a basic block (BB); a BB is a straight-line sequence of statements containing no branches or internal entrances or exit points. In other words, they do not change the control structure. That is, the branch states (the states have more than one outgoing edge) and the merge states (the states have more than one incoming edge) of the scheduled result can be one-to-one and onto mapped to those of the behavior description. A BB-based algorithm is either to minimize the total number of control steps under resource constraints or to minimize the resources requirement in a given number of control steps under timing constraints. List scheduling [4][5] and force-directed scheduling [6] are two well-known BB-based algorithms.

Fig. 2 shows two FSMDs, M_β and M_α, representing the descriptions before and after BB-based scheduling. In an FSMD, a node is a state and an edge is a control step. Each edge of an FSMD consists of status and assignment statements; a status consists of predicates. A slash separates the statuses and assignment statements; "-" denotes that no status needs to be satisfied. A branch state or a merge state is depicted as a gray node. Obviously, the branch state and the merge state of M_β have a bijective mapping to those of M_α. That is, M_β and M_α have the same control structure.

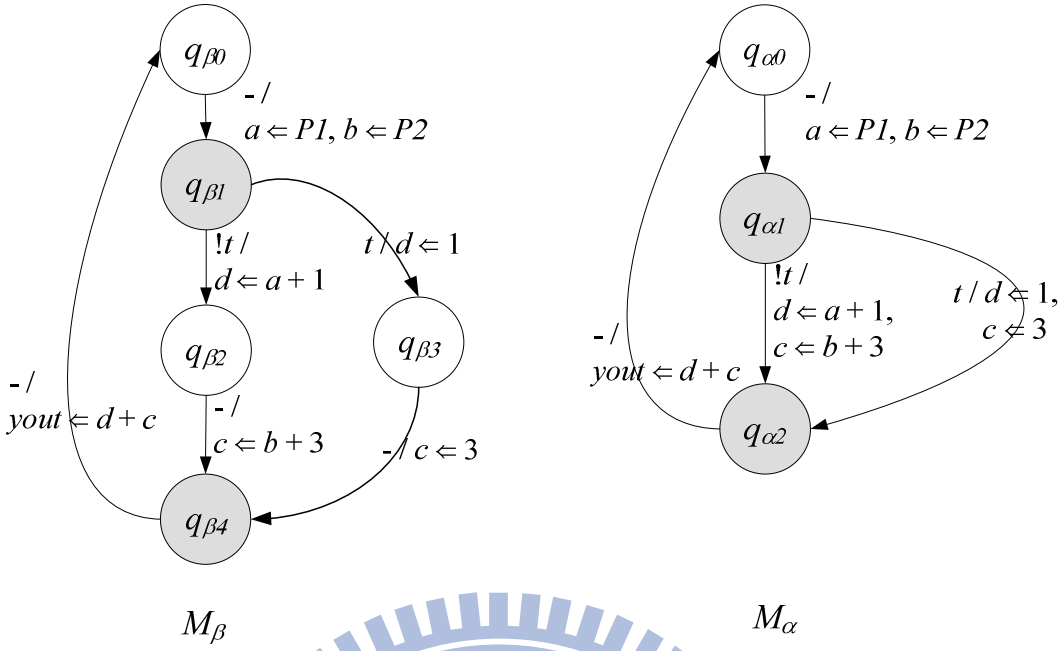


Fig. 2 An Example of BB-based Scheduling

Control-flow-based (CF-based) algorithms focus on taking the advantage of the mutual exclusion of operations in the description by analyzing the conditional constructs. These algorithms may modify the control structure. The only goal of CF-based algorithms is to minimize the number of control steps in all sequences of operations under resource constraints. Path-based scheduling (PBS) is the main algorithm of CF-based scheduling [7]. Fig. 3 shows that PBS changes the control structure. The FSMD before PBS, M_β, has two branch states and the FSMD after PBS, M_α, has only one.

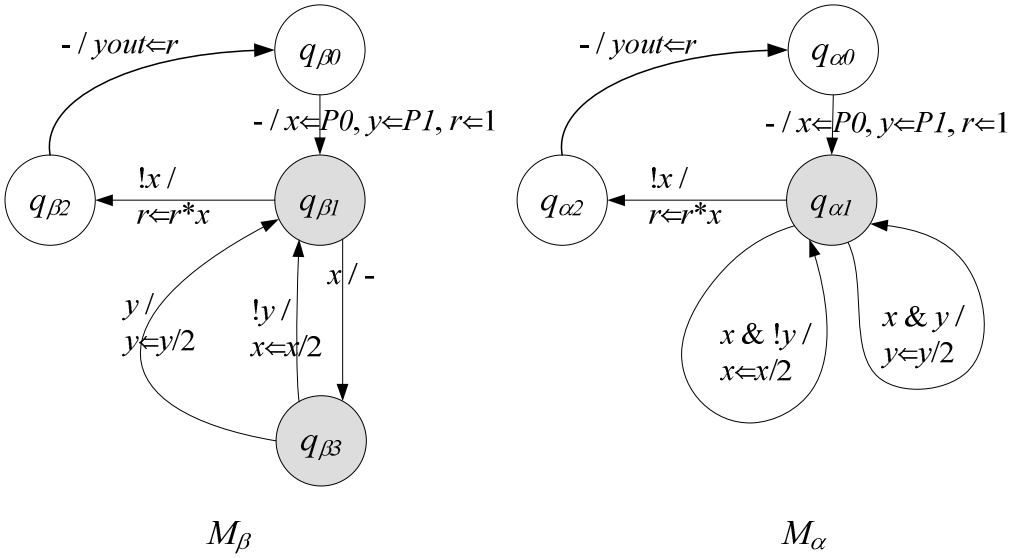


Fig. 3 An Example of Path-based scheduling

In practice, designs tend to use significant amounts of the control flow as well as the data flow. To increase the performance and the resource utilization, many code transformation techniques such as speculation – derived from the compiler – have been employed by scheduling [8]-[15].

This thesis only discusses the global code transformation techniques. A transformation is local if it only looks at the statements in a BB; otherwise, it is global. Here, two kinds of the global code transformation techniques are introduced, code motions and common subexpression elimination (CSE).

Fig. 4 illustrates code motions. Code motions attempt to extract the inherent parallelism in designs and increase the resource utilization. They move operations across the boundaries of BBs. Each square block is a BB and the node having circular shape is a branch state or a merge state. A BB between the branch state and the merge state is called a branch BB. The solid lines represent the control flow and the dotted lines represent the direction of code motions. Fig. 4 contains following code motions:

- Duplicating down (DD): It moves operations from the BB preceding a branch state into all branch BBs. This is shown by arcs marked 1.
- Duplicating up (DU): It moves operations from the BB succeeding a merge state into all branch BBs. This is shown by arcs marked 2.
- Merging up (MU): If all branch BBs have the same operations, it moves these operations from all branch BBs to the BB preceding the branch state. This is shown by arcs marked 3.
- Merging down (MD): If all branch BBs have the same operations, it moves these operations from all branch BBs to the BB succeeding the merge state. This is shown by arcs marked 4.
- Useful move (UM): Move operations from the BB succeeding the merge state into the BB preceding the branch state as these operations are independent to all branch BBs; or vice versa. Arc 5 indicates this. It moves operations from BB3 to BB0 or from BB0 to BB3 if these operations are independent to BB1 and BB2.
- Speculation (Sp): Move operations from one of branch BBs into the BB preceding the branch state if the outputs of the system are the same. It is shown as arc 6.
- Reverse speculation (RSp): Move operations from the BB preceding a branch state into some of branch BBs if the outputs of the system are the same. It is shown as arc 7.

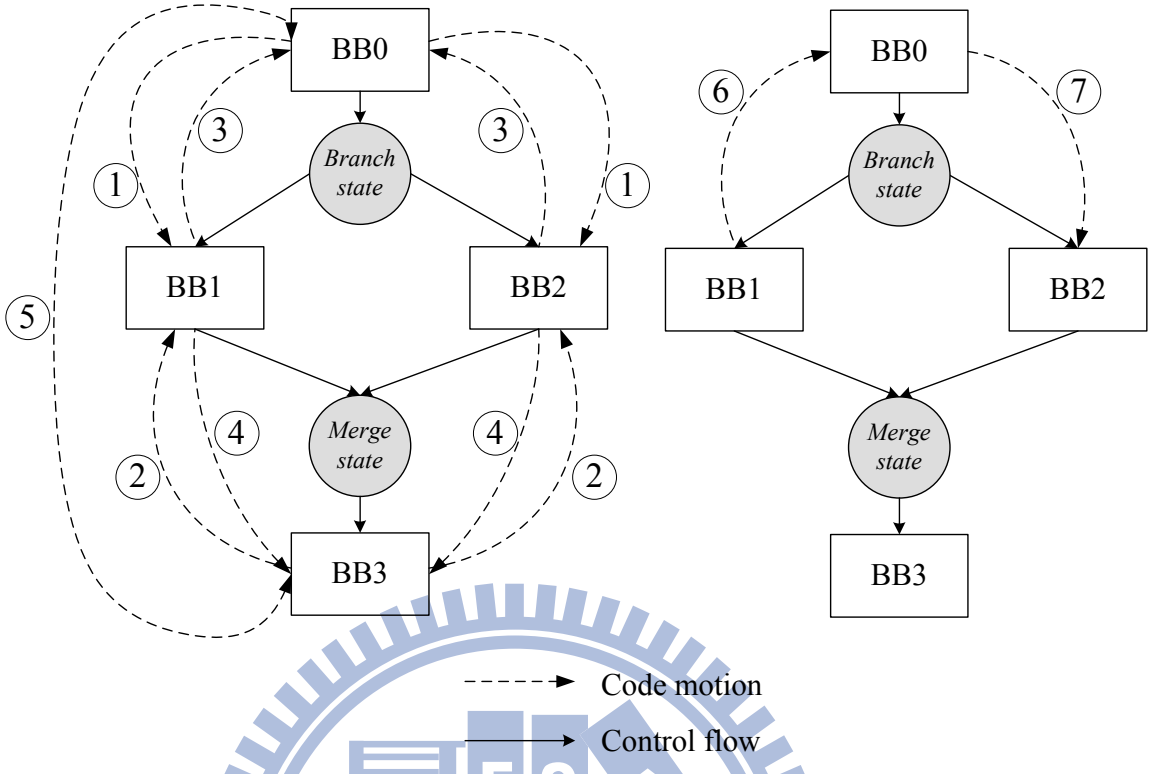


Fig. 4 Code Motions

Fig. 5 gives an example of CSE. Gupta et al. [15] have shown that CSE can often expose opportunities for optimizations. The original FSMD is M_β. M_α is the result of CSE. $q_{\beta 0}$ and $q_{\alpha 0}$ are the initial states where the system starts from. In M_β, the statements *st2*(*e'*⇐*a*+*b*) and *st3*(*e*⇐*a*+*b*) compute the same expression "*a*+*b*". Obviously, *st2* always executes prior to *st3* and no statement redefines *a*, *b* and *e'* between them. Therefore CSE replaces "*a*+*b*" of *st3* with *e'* in M_α. Note that although statement *st6*(*g*⇐*a*+*b*) has "*a*+*b*", "*a*+*b*" of *st6* can not be replaced. *st2* and *st6* do not compute the same expression because statement *st5*(*b*⇐*a*+*f*) redefines *b* between *st2* and *st6*.

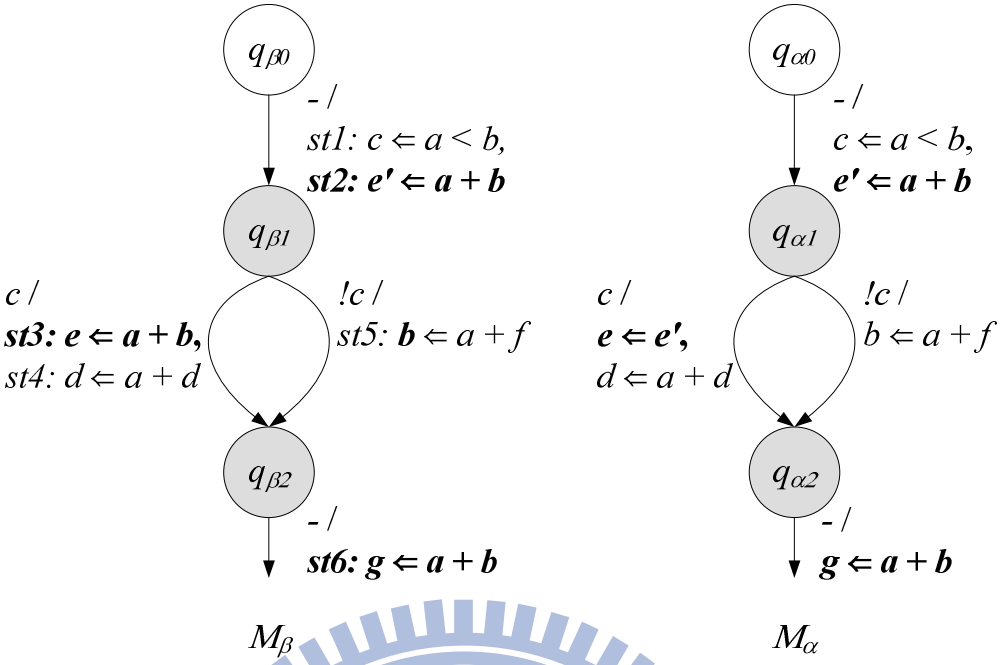


Fig. 5 An Example of Common Subexpression Elimination

1.2 Related Works

Ernst et al [16] and Bergamaschi et al [17] propose a simulation-based verification method of HLS verification. Unfortunately, simulation is becoming inadequate with the increasing in system complexity.

Chiang et al [18] propose a model checking technique by using Petri Net model as the formal description to check the correctness of BB-based algorithms. Narasimhan et al [19] prove the correctness of the force-directed list scheduler (FDLS) [28] algorithm in PVS [27] and insert invariant properties as program assertions in the implementation of the FDLS algorithm. Radhakrishnan et al [20][21] propose a method based on the precondition-based correctness of register transfer split (RTS) [29] and on the completeness of RTS transformations to perform the scheduling task with the schedule table generated by the scheduling algorithm. However, those methodologies are difficult to identify properties or

complete transformations in a large and complex scheduling task.

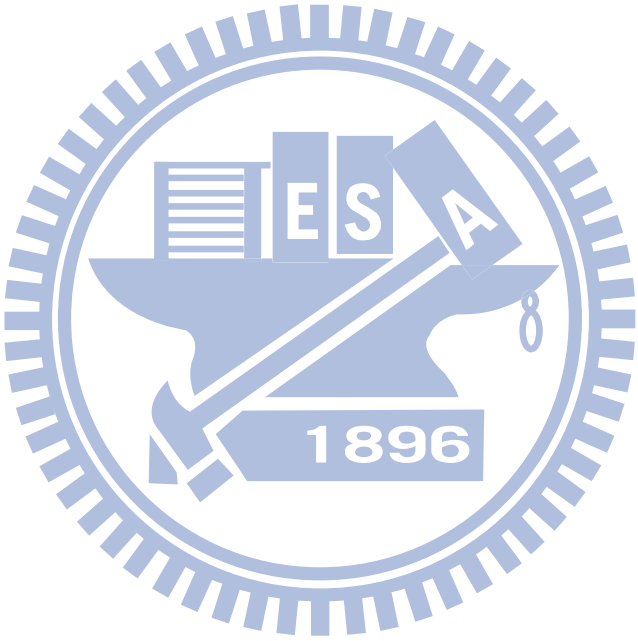
Recently, equivalence checking technique is used to prove the functional equivalence of two designs. Equivalence checking can be directly applied to the scheduling verification. It reads descriptions before and after scheduling as inputs. If two descriptions are functionally equivalent, it works successfully; otherwise, it gives a counter example for diagnosis. Neither the knowledge of scheduling nor the creation of properties or transformations is needed.

Mansouri et al [22] introduce the critical states to define the critical path. The equivalence between critical paths of behavior description and those of scheduled description was proved using the PVS theorem prover. Kim et al [23] define the functional equivalence between two FSMDs representing the descriptions before and after scheduling and proved the equivalence with PVS. The break-points are introduced to decompose an FSMD into a set of path predicates. Both [22] and [23] assume that the control structure is not modified during scheduling. The aforementioned methodologies, [18]-[23], are only well suited to BB-based scheduling.

Eveking et al [24] represent the behavioral description and the scheduled description in LLS (language of labeled segments) language and give basic transformations to prove the computational equivalence of LLS. If the behavioral description can be transformed to the scheduled result according to the basic transformations, they are computationally equivalent. However, the completeness of basic transformations is hard to define and the transformation from the behavioral LLS to the scheduled LLS is tough and tedious. Kim et al [25] extend the equivalence checking method of [23] to handle the scheduling employing MU and DD by concatenating the critical paths. However, the paths affected by the code motions need to be identified. Karfa et al [26] propose an equivalence checking method suited to PBS as well as BB-based scheduling. FSMDs are used to represent the descriptions before and after scheduling and they are characterized by a finite set of paths. The equivalence of FSMDs is transformed into the equivalence of paths. However, it does not support some code

transformation techniques, such as Sp and CSE.

The rest of this thesis is organized as follows. In Chapter 2, the theoretical concepts of the equivalent checking method proposed by Karfa et al [26] are discussed. Chapter 3 presents two motivational examples, and Chapter 4 describes our proposed method in details. The experimental results and analyses are provided in Chapter 5. Followed Chapter 6 gives a conclusion and identifies some directions for future works.



Chapter 2
Preliminaries

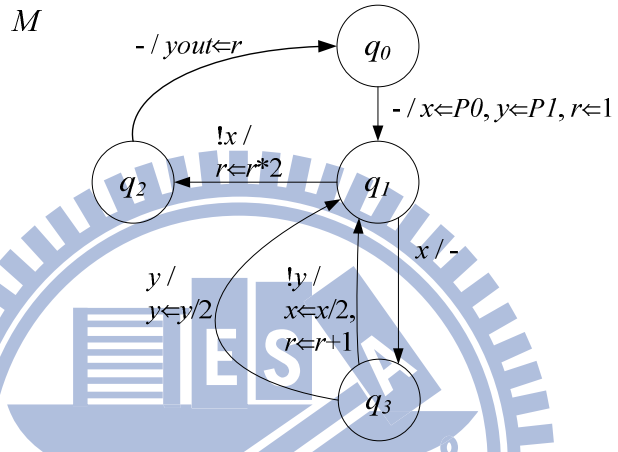
Here, we discuss a verification method and its theoretic conceptions proposed by Karfa et al [26]. Our approach is based on those conceptions and their verification method. The notion of computations, paths, and a path cover of an FSMD and the transformations between them are defined. The equivalence of two FSMDs is also derived.

2.1 Finite State Machine with Data-path (FSMD)

An FSMD was proposed by Gajski et al. in [1]. It can trivially implement the descriptions of algorithm level (or behavior level) and RT-level. FSMDs are used to represent the descriptions before and after scheduling with an additional initial state. An initial state is also called a reset state. The FSMD was defined as a 7-tuple, $M = <Q, q_0, I, O, V, f, h>$, where

1) Q is the finite set of states,
2) q_0 is the reset state,
3) I is the finite set of inputs,
4) O is the finite set of outputs,
5) V is the finite set of variables,
6) f: $Q \times S \rightarrow Q$ is the state transition function,
   - S is the set of status expressions consisting of arithmetic predicates over $I \cup V$,
7) h: $Q \times S \rightarrow U$ is the update function of the outputs and variables, where
   - $U = \{x \Leftarrow e \mid x \in O \cup V$ and e is an arithmetic predicate or expression over $I \cup V\}$ is a set of variables or output assignment statements.

An FSMD is inherently deterministic. For any state *q*, if the status expressions *s1* = *s2*, it implies that *f*(*q*,*s1*) = *f*(*q*,*s2*) and *h*(*q*,*s1*) = *h*(*q*,*s2*). In an FSMD, the concept of time can be thought of as the order in which the statements are executed. An example of a behavioral description and its FSMD are shown in Fig. 6. A behavior description *E1* using C language is described in (b). An FSMD *M* in (a) represents *E1* and the detail of *M* is shown in (c).



(a) An FSMD model *M* for *E1*

```
int yout;
void E1(P0,P1){
      int x = P0;
      int y = P1;
      int r = 1;
      while(x){
            if(y) y = y/2;
            else{
                  x = x/2;
                  r = r+1;}
      }//endwhile
      r = r * 2;
      yout = r;
}//endofE1
```

(b) The behavior *E1* in C

M = <*Q*, q_0, *I*, *O*, *V*, *f*, *h*>

Q = {q_0, q_1, q_2, q_3}; *I* = {*P0*,*P1*}; *O* = {*yout*}; *V* = {*x*,*y*,*r*}

$f(q_0,\text{TRUE}) = q_1$,

$f(q_1,!x) = q_2$, $f(q_2,\text{TRUE}) = q_0$,

$f(q_1,x) = q_3$, $f(q_3,y)=q_1$, $f(q_3,!y) = q_1$;

$h(q_0,\text{TRUE}) = \{x \Leftarrow P0, y \Leftarrow P1, r \Leftarrow 1\}$,

$h(q_1,!x) = \{r \Leftarrow r*2\}$, $h(q_2,\text{TRUE}) = \{yout \Leftarrow r\}$,

$h(q_1,x) = \emptyset$, $h(q_3,y) = \{y \Leftarrow y/2\}$, $h(q_3,!y) = \{x \Leftarrow x/2, r \Leftarrow r+1\}$.

(c) The details of *M*

Fig. 6 An FSMD

In an FSMD, a walk represents a sequence of transitions.

Definition 1 [26] *Walk* ω of M from q_i to q_j

A walk ω from a *start state* q_i to an *end state* q_j is a transition sequence of the states. It is formulated as $q_i \xrightarrow[C_i]{} q_{i+1} \xrightarrow[C_{i+1}]{} \ldots \xrightarrow[C_{i+n-1}]{} q_{i+n} = q_j$ where $q_k \in Q$ for all k, $i \le k \le i+n$ and there exits $c_k \in S$ and transition functions f_k such that $f_k(q_k, c_k) = q_{k+1}$ for all k, $i \le k \le i+n-1$. For short, $q_i \Rightarrow q_j$.

A state of a walk is called an *internal state* if it is neither the start state nor the end state.

2.1.1 Paths

A path is a finite walk. Its definition and characteristic formula are described here.

Definition 2 [26] *Path* β of M from q_i to q_j

A path β from q_i to q_j is a finite walk where all the states are distinct or the end state is only identical to the start state of β.

Fig. 7 illustrates paths that *p1* and *p2* are paths and *p3* is NOT a path.

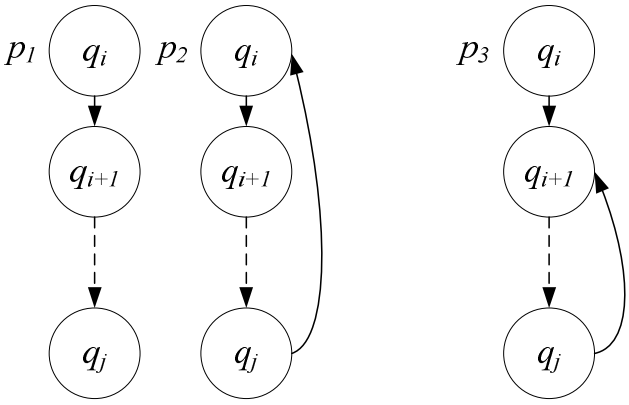


Fig. 7 Paths

Definition 3 [26] *Condition* R_β of a path β

The condition R_β of β is a logical expression over $I \cup V$ which should be satisfied before executing β.

Definition 4 [26] *Result* r_β of a path β

The result r_β of β is an ordered pair consisting of an *updated variable set* ϖ and an *updated output list* o after executing β and is denoted as $<\varpi_\beta, o_\beta>$.

1) Updated variable set ϖ_β is an ordered tuple $<e_i>$ of arithmetic predicates or expressions over $I \cup V$ for all variables in V and each e_i represents the final value of the variable $v_i \in V$ at the end state q_j.

2) Updated output list o_β is a list of the output assignment statements with the order in which the outputs occur in β.

For a path β, its condition R_β and its result r_β can be computed by substitution. Substitution is a symbolic execution and based on the rule: If a predicate $c(y)$ is true after assignment statement $y \Leftarrow g(y)$, the predicate $c(g(y))$ is true before $y \Leftarrow g(y)$ [30].

Fig. 8 describes an example of how to compute R_β and r_β of β in M by substitution. Let $V = \{a, b, c, d, x\}$, $I = \{i1, i2\}$, and $O = \{o1, o2\}$ of M. At first, the initial values are $\{a, b, c, d, x\}$ for each variable of V. Hence, at the start state $q_{\beta 1}$, $R_\beta = \emptyset$, $\varpi_\beta = <a, b, c, d, x>$ and $o_\beta = \emptyset$. The first transition from $q_{\beta 1}$ to $q_{\beta 2}$ has only one status, thus, the condition becomes "$R_\beta = \boldsymbol{a} \leq \boldsymbol{x}$"; it also has only one assignment statement "$c \Leftarrow a+d$ ", thus, "$a+d$ " substitutes for "c" and ϖ_β becomes "$<a, b, \boldsymbol{a+d}, d, x>$". Then, the initial values for the next transition, "$q_{\beta 2} \longrightarrow q_{\beta 3}$", become $\{a, b, a+d, d, x\}$. Similarly, after executing "$q_{\beta 2} \longrightarrow q_{\beta 3}$", the updated variable set becomes "$\varpi_\beta = <\boldsymbol{a*b}, b, a+d, d, x>$" and the updated output list becomes "[$\boldsymbol{o1 \Leftarrow a}$]". Therefore, the initial values of "$q_{\beta 3} \longrightarrow q_{\beta 4}$" are $\{a*b, b, a+d, d, x\}$. Finally, after "$q_{\beta 3} \longrightarrow q_{\beta 4}$", "$R_\beta =$

$(a \le x)$ & $(x <$ ***a+d***$)$" and "$r_\beta = <\varpi_\beta, o_\beta> = <<a*b, b, a+d, d,$ ***a*b+a+d***$>, [o1 \Leftarrow a,$ ***o2*⇐*a*b***$]>$" at $q_{\beta 4}$ are computed. Note that since the status is always prior to the assignment statements of a transition, the execution of "$x \Leftarrow a+c$" will not redefine the "x" of "$x < c$". And only the initial value "$a + d$ " substitutes for "c" of "$x < c$" in "$q_{\beta 3} \longrightarrow q_{\beta 4}$".

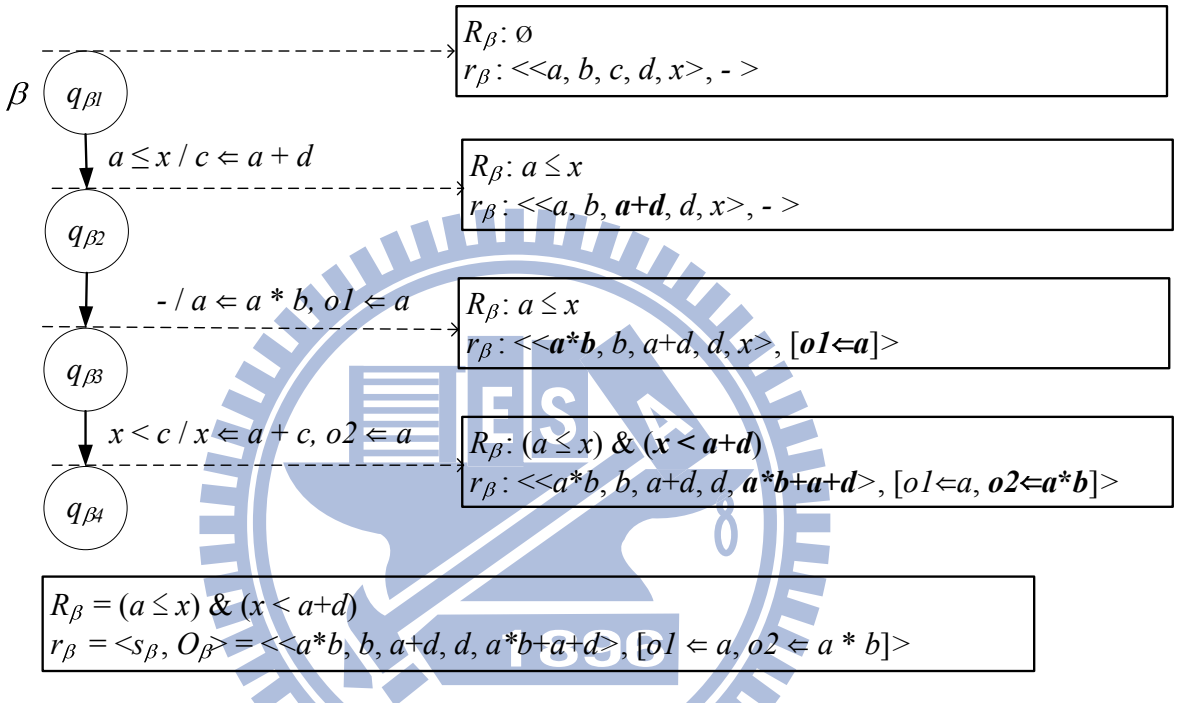


Fig. 8 Compute R_β and r_β

2.1.2 Characterization of a Path

Depending on the aforementioned definitions and the substitution method, a path can be characterized as "$R_\beta(v)$ & $r_\beta(v)$" or "$R_\beta(v)$ & $(v_f = \varpi_\beta(v))$ & $o_\beta(v)$" where $v \in I \cup V_\beta$ and $v_f \in V_\beta$.

A concatenation of a sequence of paths can also be characterized. Let "$\beta = q_i \Rightarrow q_k$" and "$\alpha = q_k \Rightarrow q_j$" are paths of an FSDM M and β is a preceding path of α; their characteristic formulas are "$R_\beta(v) \& (v_f = \varpi_\beta(v)) \& o_\beta(v)$" and "$R_\alpha(v) \& \varpi_\alpha(v) \& o_\alpha(v)$", respectively. Then β and

α can be concatenated, denoted as $\beta\alpha$, and the characteristic formula of $\beta\alpha$ is "$R_\beta(v)$ & $R_\alpha(v_f)$ & $\varpi_\alpha(v_f)$ & $(o_\beta(v)o_\alpha(v_f))$".

2.1.3 Computational Equivalence of Paths

$M_\beta = <Q_\beta, q_{\beta 0}, I_\beta, O_\beta, V_\beta, f_\beta, h_\beta>$ and $M_\alpha = <Q_\alpha, q_{\alpha 0}, I_\alpha, O_\alpha, V_\alpha, f_\alpha, h_\alpha>$ are two FSMDs.

Definition 5 [26] *Computational equivalence* of paths ($\beta \simeq \alpha$)

Let β and α be paths of M_β and M_α, respectively. β and α are computationally equivalent, denoted as $\beta \simeq \alpha$, if $R_\beta = R_\alpha$ and $r_\beta = r_\alpha$ over $V_\beta \cap V_\alpha$.

Notice that the equivalence of paths is restricted to $V_\beta \cap V_\alpha$. That is, it is not allowed if R and r have some variables not in the intersection and the final values of variables which are not in the intersection are ignored. An example of equivalent paths is given below.

Example 1: Let $V_\beta = \{a, b\}$ and $V_\alpha = \{a, b, x\}$, then $V_\beta \cap V_\alpha = \{a, b\}$. Let β and α be two paths of M_β and M_α, respectively. Assume $R_\beta = R_\alpha$ and both have no output assignment statements.

1. If $r_\beta = <<a*b, b>, - >$ and $r_\alpha = <<a*b, b, a*b + a >, - >$, ignore the value of $x \notin V_\beta \cap V_\alpha$.

   $\Rightarrow \beta \simeq \alpha$.

2. If $r_\beta = <<a*b, b>, - >$ and $r_\alpha = <<a*x, b, a*b+a >, - >$, clearly, the final value of a uses $x \notin V_\beta \cap V_\alpha$ in r_α.

   $\Rightarrow \beta \not\simeq \alpha$.

2.2 FSMD Equivalence

Let $M_\beta = <Q_\beta, q_{\beta 0}, I_\beta, O_\beta, V_\beta, f_\beta, h_\beta>$ and $M_\alpha = <Q_\alpha, q_{\alpha 0}, I_\alpha, O_\alpha, V_\alpha, f_\alpha, h_\alpha>$ be two FSMDs representing the descriptions before and after scheduling, respectively. The objective is to verify whether M_β behaves exactly as M_α. That is, whether they produce the same output sequences for all possible input sequences.

2.2.1 Assumptions

Since the design synthesized by HLS is usually applied to a component of a system, the interface should be unchanged after scheduling. Therefore, we have the Assumption 1.

Assumption 1:

The input set and output set of two FSMDs are identical. That is, $I_\beta = I_\alpha$ and $O_\beta = O_\alpha$.

Assumption 2:

The system works in an infinite outer loop. That is, an FSMD represents a system, and every walk of an FSMD starting from the reset state can always go back to the reset state. More specifically, every inner loop of an FSMD must have at least one exit point.

Assumption 2 is reasonable since a hardware design is usually designed to have a reset state to prevent the dead lock. According to Assumption 2, all inner loops of an FSMD can be cut by a cutpoint introduced in a latter section.

2.2.2 Computations of an FSMD

Based on Assumption 2, a walk of an FSMD from the reset state to the reset state can be seen as a complete computation. In other words, revisiting the reset state implies a termination of a computation and a beginning of a new computation. Hence, an FSMD can be thought of

as a set of computations. Then the equivalence of FSMDs can transform to the equivalence of computations.

Definition 6 [26] *Computation* μ

A computation μ is a finite walk from the reset state q_0 to q_0 and it has no intermediary occurrence of q_0.

Definition 7 [26] *Computational equivalence* of walks

Two walks ω_β of M_β and ω_α of M_α are computationally equivalent, denoted as $\omega_\beta \simeq \omega_\alpha$, if $R_{\omega\beta} = R_{\omega\alpha}$ and $r_{\omega\beta} = r_{\omega\alpha}$ over $V_\beta \cap V_\alpha$.

Definition 8 [26] M_β is *computationally contained* in M_α ($M_\beta \subseteq M_\alpha$)

M_β is computationally contained in M_α, denoted as $M_\beta \subseteq M_\alpha$, if, for each computation μ_β of M_β, there exists a computation μ_α of M_α such that $\mu_\beta \simeq \mu_\alpha$.

Definition 9 [26] *Computational equivalence* of FSMDs ($M_\beta \cong M_\alpha$)

M_β and M_α are computationally equivalent, denote as $M_\beta \cong M_\alpha$, if $M_\beta \subseteq M_\alpha$ and $M_\alpha \subseteq M_\beta$.

2.2.3 Path Covers of an FSMD

Since an FMSD may have inner loops, there may be infinite computations of an FSMD. According to Section 2.1.2, a walk is a concatenation of a sequence of paths; a computation can be a concatenation of a set of paths.

Definition 10 [26] A *path cover P* of an FSMD *M*

A path cover *P* of *M* is a finite set of paths, if, for each computation of *M*, it can be composed of the paths of *P*.

Notice that a path cover of an FSMD is not unique. Fig. 9 illustrates path covers of an FSMD *M*. *M* has only two computations, one is "$\mu 1 = q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow q_3 \longrightarrow q_4 \longrightarrow q_6 \longrightarrow q_0$" and another is "$\mu 2 = q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow q_5 \longrightarrow q_6 \longrightarrow q_0$". $P1 = \{p1, p2, p3\}$ is a path cover of *M* since $\mu 1$ can be composed of *p1* and *p2* by concatenating them and $\mu 2$ can be composed of *p1* and *p3*. $P2 = \{\mu 1, \mu 2\}$ is also a path cover of *M*.

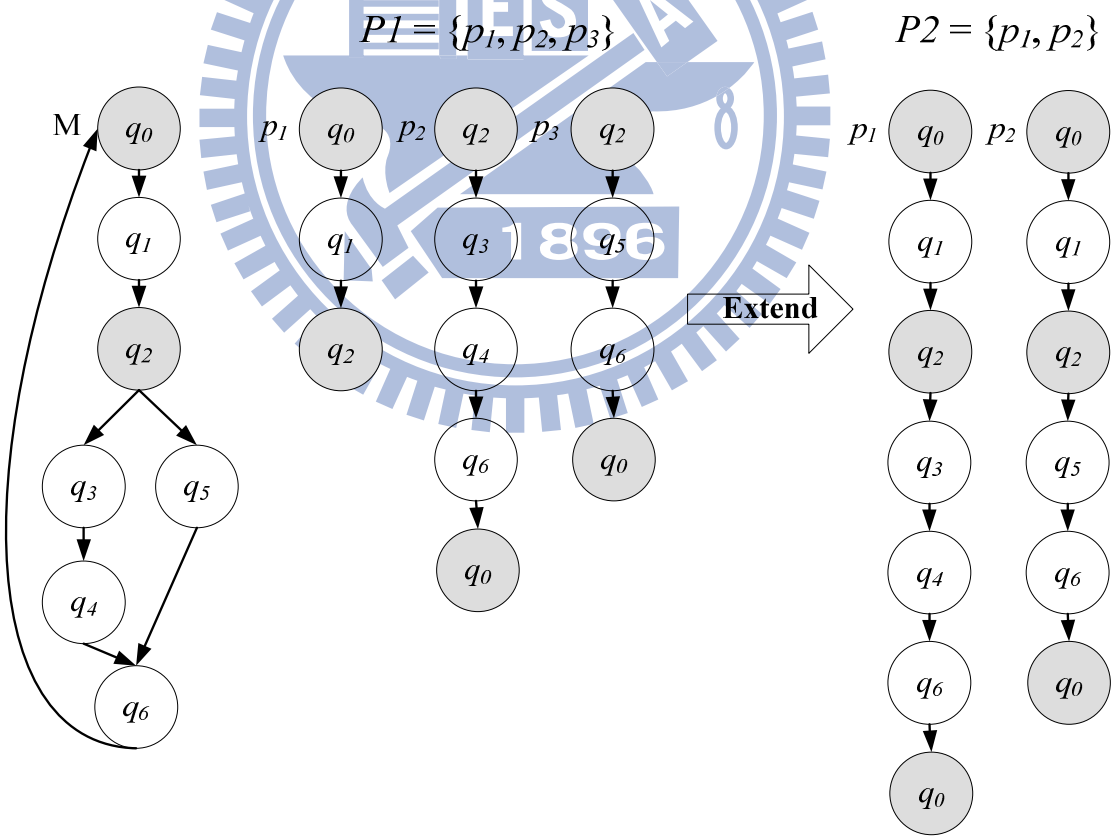


Fig. 9 Path Covers of *M*

Therefore, following Theorem 1 is derived and it has been proved by Karfa et al [26].

Theorem 1 [26] $M_\beta \subseteq M_\alpha$ if there exists a path cover $P_\beta = \{p_{\beta 0}, \ldots, p_{\beta n}\}$ of M_β and a set of path $P_\alpha = \{p_{\alpha 0}, \ldots, p_{\alpha n}\}$ of M_α such that $p_{\beta i} \simeq p_{\alpha i}$ for all i, $0 \le i \le n$.

Theorem 1 transforms the problem of equivalent FSMDs to the problem of equivalent paths. To find an equivalent path $p_{\alpha i}$ in M_α for each $p_{\beta i}$ of M_β, we define the corresponding states where the comparison of two paths starting from.

Definition 11 [26] *Corresponding state* (*CS*)

1) The reset states $q_{\beta 0} \in Q_\beta$ and $q_{\alpha 0} \in Q_\alpha$ are native corresponding states and
2) $q_{\beta m} \in Q_\beta$ and $q_{\alpha n} \in Q_\alpha$ are corresponding states if $q_{\beta i} \in Q_\beta$ and $q_{\alpha j} \in Q_\alpha$ are corresponding states and there exist paths $\beta = q_{\beta i} \Rightarrow q_{\beta m}$ and $\alpha = q_{\alpha j} \Rightarrow q_{\alpha n}$, such that $\beta \simeq \alpha$.

It is hard to find a path cover constituting all possible computations of an FSMD because of the loops of an FSMD. Therefore, Karfa et al [26] introduce the cutpoints. Each loop of an FSMD can be cut by at least one cutpoint and the set of paths between cutpoints without any intermediary occurrence of cutpoint is a path cover. According to Assumption 2, each inner loop of an FSMD must have an exit point, i.e. a branch state; hence, the reset state and the branch states are selected to be the *initial cutpoints*. The set of paths between the initial cutpoints without any intermediary occurrence of initial cutpoint is named an *initial path cover* and a path of the initial path cover is named an *initial path*.

2.2.4 Verification Method

Based on Theorem 1, for each path of the initial path cover P_β of M_β, we want to find an

equivalent path in M_α. Because scheduling may change the control structure, a path "$\beta = q_{\beta i} \Rightarrow q_{\beta j}$" of P_β in M_β may not find a computationally equivalent path in M_α. The path extension method, proposed by Karfa et al [26], is a solution to handle this situation. It extends a path to build a new path cover. The path extension method extends β with following steps:

1. Find the path sets *ps* and *pe*.
   1) *ps* is the set of all paths of P_β ending at $q_{\beta j}$.
   2) *pe* is the set of all paths of P_β starting from $q_{\beta j}$.
2. For each path β_s of *ps*, concatenate β_s and each path in *pe*. *Bm* is the set of all such concatenated paths and $q_{\beta j}$ is not a cutpoint now.
3. Remove each path, which is a path of *ps* or a path of *pe*, from P_β.
4. Add all paths of *Bm* into P_β.

After the process, P_β becomes a new path cover of M_β. A path extension is invalid if it becomes a loop or it needs to extend via the reset state.

Fig. 10 illustrates the verification flow of [26]. Two FSMDs, which are M_β and M_α representing the descriptions before and after the scheduling, are the inputs of the algorithm. At first, the algorithm inserts the cutpoints only into M_β and finds initial path cover P_β of M_β. Then, it starts from the reset states to find a computationally equivalent path from M_α for each path of P_β. First, it finds "$\beta = q_{\beta i} \Rightarrow q_{\beta j}$" from P_β depending on the corresponding states ($q_{\beta i}$, $q_{\alpha m}$), and then it find a computationally equivalent path in M_α. If a computationally equivalent path "$\alpha = q_{\alpha m} \Rightarrow q_{\alpha n}$" is found, it records the paths (β,α) as the computationally equivalent paths and their end states ($q_{\beta j}$, $q_{\alpha n}$) as the corresponding states. But if a computationally equivalent path is not found, extend path β to build a new path cover, P_β. Note that, if β is not extensible, the algorithm fails; otherwise, it repeats the process until all paths of P_β finding their computationally equivalent paths in M_α. Hence, $M_\beta \subseteq M_\alpha$ is proved. Then, it interchanges M_β and M_α and repeats the process to prove $M_\alpha \subseteq M_\beta$. As both $M_\beta \subseteq M_\alpha$

and $M_\alpha \subseteq M_\beta$ have been proved, $M_\beta \cong M_\alpha$.

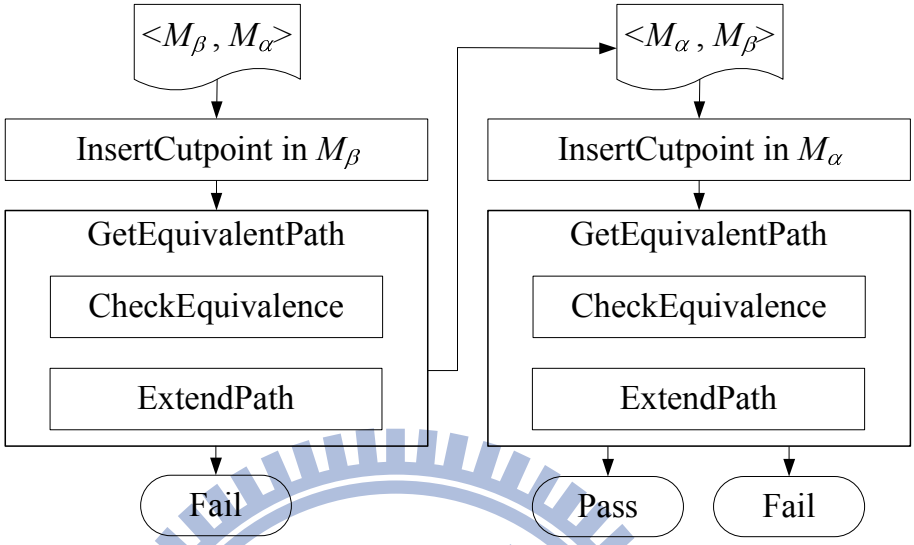


Fig. 10 Verification Flow

Obviously, Karfa's algorithm has the ability to cope with BB-based scheduling. Since BB-based scheduling does not change the control structure, the bijective mapping of cutpoints are preserved. It means that for each path of one FSMD, the algorithm can straightly find the computationally equivalent path of another one without any extension.

Karfa's algorithm is also capable of verifying PBS. PBS only merges some consecutive paths; it doesn't move the operations across the BB boundaries after merging the paths. Therefore, the algorithm with path extension method is strong enough to handle PBS.

Karfa's algorithm obviously supports some code motions: DD, DU, MU, MD, and UM through path extension; since moving the operations from one path to another path can be thought of as merging these consecutive paths. Note that a merge state is not a cutpoint, the algorithm without path extension inherently handles DD and MU.

Chapter 3
Motivation

Although Karfa's algorithm discussed in Chapter 2 can verify the BB-based and PBS, it is still weak in handling some code transformation techniques, such as Sp, RSp, and CSE. Following sections give two such examples and we proposed the solutions for these cases in the next chapter.

3.1 An Example of Speculation

The equivalence of two paths defined by Karfa et al can not handle the result of scheduling employing Sp or RSp. An example of Sp is shown in Fig. 11. M_β is the original FSMD, and it is functionally equivalent to M_α. Two systems are functionally equivalent if they produce the same output sequences for all possible input sequences. Let $V_\beta \cap V_\alpha = \{a, b, c, d\}$, $I = \{x, y\}$ and $O = \{out\}$. The scheduler moves "$b \Leftarrow x+y$" from "$q_{\beta 1} \longrightarrow q_{\beta 3}$" to "$q_{\beta 0} \longrightarrow q_{\beta 1}$" and generates M_α. The left computations are "$\mu_{\beta l} = q_{\beta 0} \longrightarrow q_{\beta 1} \xrightarrow[c]{} q_{\beta 2} \longrightarrow q_{\beta 4} \longrightarrow q_{\beta 0}$" of M_β and "$\mu_{\alpha l} = q_{\alpha 0} \longrightarrow q_{\alpha 1} \xrightarrow[c]{} q_{\alpha 2} \longrightarrow q_{\alpha 3} \longrightarrow q_{\alpha 0}$" of M_α shown in bold lines. Their characteristic formulas are "$(x<y)$ & $<x-y, \mathbf{b}, x<y, x-y+1>$ & $[out \Leftarrow x-y+1]$" and "$(x<y)$ & $<x-y, \mathbf{x+y}, x<y, x-y+1>$ & $[out \Leftarrow x-y+1]$", respectively. Obviously, $\mu_{\beta l}$ and $\mu_{\alpha l}$ are not computationally equivalent because $<x-y, \mathbf{b}, x<y, x-y+1> \neq <x-y, \mathbf{x+y}, x<y, x-y+1>$; $\mu_{\beta l}$ is not extensible. Therefore, M_β and M_α are not computationally equivalent. However, the conditions and outputs of $\mu_{\beta l}$ and $\mu_{\alpha l}$ are equivalent; the different values won't affect the outputs of all computations of M_β and M_α. That is, $\mu_{\beta l}$ and $\mu_{\alpha l}$ produce the same output values for any input. Thus, $\mu_{\beta l}$ and $\mu_{\alpha l}$ should be equivalent. Therefore, a

definition of equivalence between paths that captures the notion of functional equivalence is needed.

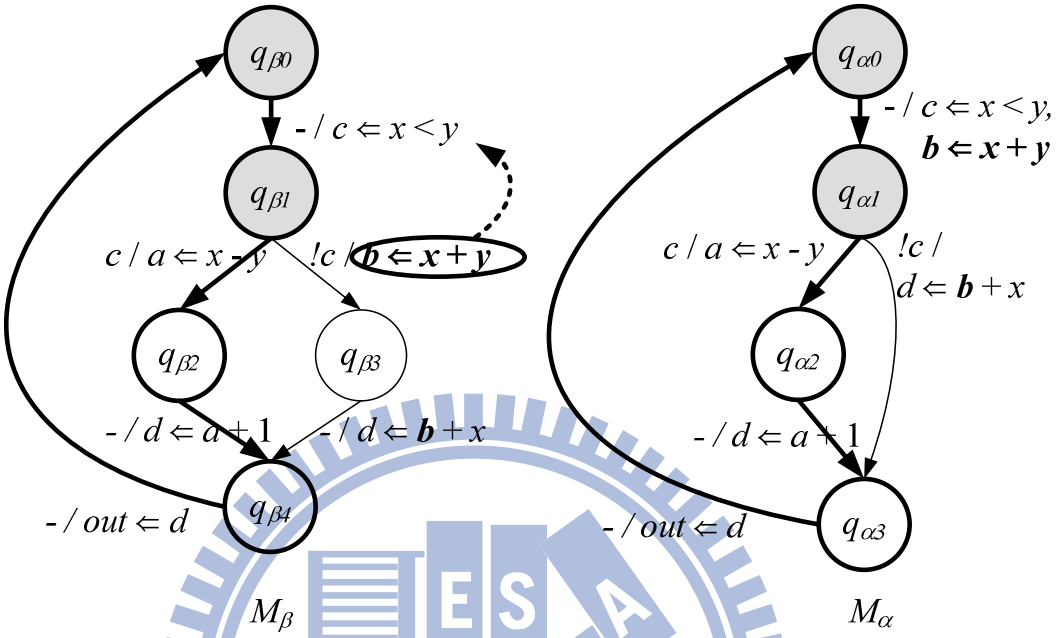


Fig. 11 An Example of Speculation

3.2 An Example of CSE

Fig. 12 illustrates an example of safe-speculation (SSp) and CSE. SSp is a method to realize Sp [15]. It attempts to move *se*($e \Leftarrow a+b$) from "$q_{\beta 1} \xrightarrow[c]{} q_{\beta 2}$" to "$q_{\beta 0} \longrightarrow q_{\beta 1}$". Unlike Sp, it introduces a new variable "*e'* " to store the value of "*a*+*b*", and then assigns "*e'* " to "*e*". After SSp, all walks starting from the reset state $q_{\alpha 0}$ to *sg*($g \Leftarrow a+b$) have *se'*($e' \Leftarrow a+b$) and there is no statement redefining "*a*", "*b*" or "*e'* " between *se'* and *sg* ; therefore, CSE replaces *sg* with "$g \Leftarrow e'$ ".

Karfa's algorithm fails in this case. At first, it calculates that "$q_{\beta 0} \longrightarrow q_{\beta 1} \simeq q_{\alpha 0} \longrightarrow q_{\alpha 1}$". Then it compares the paths, denoted in bold lines, starting from $q_{\beta 1}$ and $q_{\alpha 1}$.

Because the expression of *se* is replaced in M_α, the final values of "*e*" are not equivalent; therefore, the bold paths are not equivalent. However, if *se'*(*e'*⇐ *a*+*b*) is recorded in this case, both paths have the same final value "*a*+*b*" of "*e*".

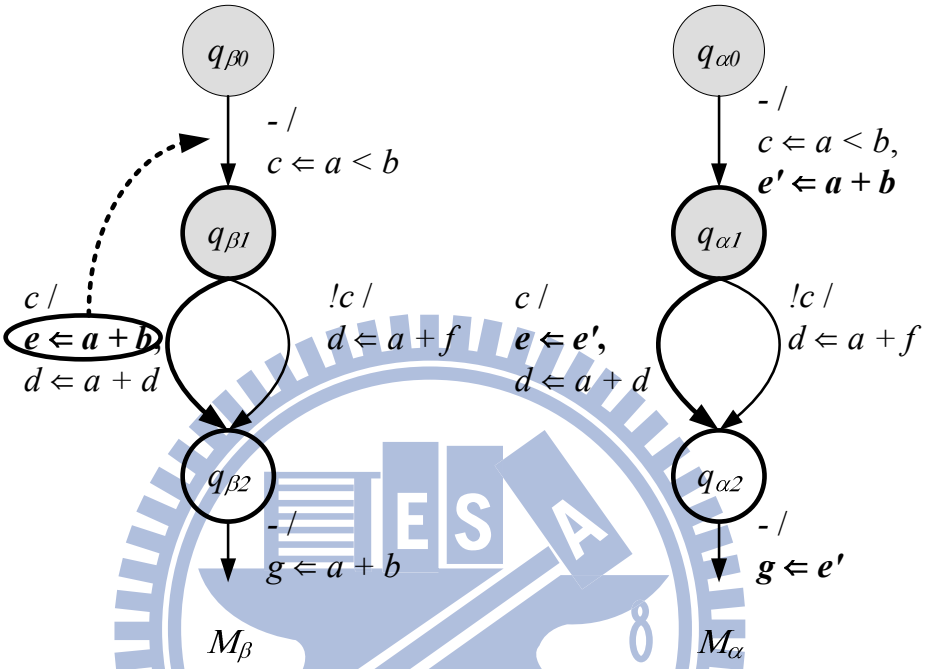


Fig. 12 An Example of Safe-speculation & CSE

Sp, Rsp, SSp, and CSE are common techniques in scheduling. None of the formal verification methods proposed in Section 1.2 can handle all of them. This thesis proposes an equivalence checking method for scheduling to support them, and our method is extended from Karfa's method [26]. Following chapter describes the detail of our method.

Chapter 4
Our Proposed Method

This chapter has three sections; first one defines equivalence of paths to cope with Sp and RSp; second one gives a solution to handle CSE; the last one is the detail of our proposed algorithm. The purpose of this thesis is to check whether two FSMDs, which are used to represent the descriptions before and after scheduling, produce the same output sequences for all possible input sequences. Our assumptions and the basic theoretical conceptions, which are proposed by Karfa et al [26], are described in Chapter 2.

In the rest of this thesis, $M_\beta = <Q_\beta, q_{\beta 0}, I_\beta, O_\beta, V_\beta, f_\beta, h_\beta>$ and $M_\alpha = <Q_\alpha, q_{\alpha 0}, I_\alpha, O_\alpha, V_\alpha, f_\alpha, h_\alpha>$ are the behavior FSMD and the scheduled FSMD, respectively. A gray node depicted in a figure represents an initial cutpoint.

4.1 Solution for Speculation

The rest thesis will use the terminologies: a statement, a used variable, and a defined variable; therefore, we give their definitions below.

Definition 12 *Statement*, *Use*, and *Define*

A *statement* "*st*:*d*⇐*e*" assigns an arithmetic predicate or expression (*e*) to a variable (*d*). Then *st* is said to *use* all variables occurring in *e* and to *define* "*d* ".

Example 2: A statement "*s*:*x*⇐*a*+*b*" is said to define "*x*" and to use "*a*" and "*b*". In this example, "*a*", "*b*" and "*x*" are the variables of *s* where "*a*" and "*b*" are called used variables and "*x*" is called defined variable.

4.1.1 Equivalence of Paths

Assume two paths which are β of M_β and α of M_α have the same condition and the same outputs, but their final values of some variables are not equivalent. If these final values will not be used in both M_β and M_α, the outputs will not be affected; hence, β and α can be thought of as two equivalent paths.

Definition 13 *Effective variable v* of a path β

A variable $v \in V_\beta$ is an effective variable of a path $\beta = q_{\beta i} \Rightarrow q_{\beta j}$ of an FSMD M_β if there exists a walk which starts from $q_{\beta j}$ and uses "v" before any defined "v".

In Fig. 13, $\beta = q_2 \longrightarrow q_4 \longrightarrow q_5$ is a path of an FSMD M, depicted in bold line. "d" is an effective variable of β because "d" is used by the walk "$q_5 \xrightarrow[!b]{} q_6$" starting from the end state q_5 of β. On the contrary, "v" is NOT an effective variable of β. Since for all possible walks starting from q_5, only the walks containing the path β have a statement "$d \Leftarrow a+v$" uses "v". Others have no statement having "v" as a used variable. However, "$d \Leftarrow a+v$" is always executed after the statement "$v \Leftarrow a+1$" redefining v. Therefore, there exists no walk starting from q_5 and using "v" before any defined "v".

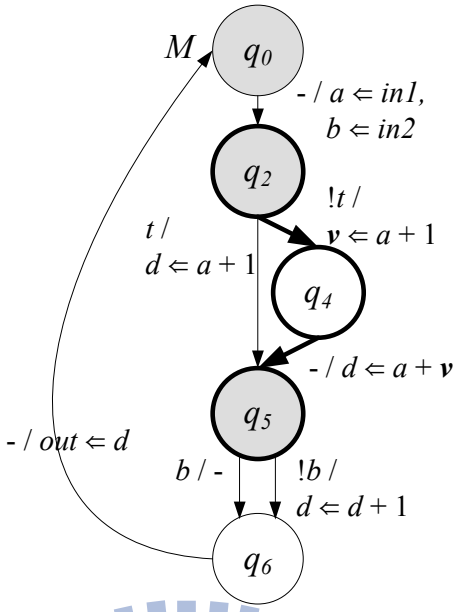


Fig. 13 An Example of an Effective Variable

Definition 14 *Equivalence* of two paths ($\beta \approx \alpha$)

Two paths, β of M_β and α of M_α, are equivalent if

1) $R_\beta = R_\alpha$, over $V_\beta \cap V_\alpha$ and

2) $o_\beta = o_\alpha$, over $V_\beta \cap V_\alpha$ and

3) $\forall v_i \in V_\beta \cap V_\alpha$ and its final value in updated variable set are $e_\beta \in \varpi_\beta$ and $e_\alpha \in \varpi_\alpha$,

- $e_\beta = e_\alpha$, over $V_\beta \cap V_\alpha$ or
- "v_i" is NOT an effective variable of β in M_β and of α in M_α.

Clearly, if $\beta \approx \alpha$, $\beta \approx \alpha$. The equivalence of two walks is defined in the similar way.

Definition 15 *Equivalence* of two walks ($\omega_1 \approx \omega_2$)

Two walks, ω_1 of M_β and ω_2 of M_α, are equivalent if

1) $R_{\omega 1} = R_{\omega 2}$, over $V_\beta \cap V_\alpha$ and

2) $o_{\omega 1} = o_{\omega 2}$, over $V_\beta \cap V_\alpha$ and

3) $\forall v_i \in V_\beta \cap V_\alpha$, $e_{\omega 1 i}$ and $e_{\omega 2 i}$ are the final values of "v_i" in $\varpi_{\omega 1}$ and in $\varpi_{\omega 2}$,

respectively,

- $e_{\omega 1i} = e_{\omega 2i}$, over $V_\beta \cap V_\alpha$ or
- "v_i" is NOT an effective variable of ω_1 in M_β and of ω_2 in M_α.

The equivalence of FSMDs is similar to the computational equivalence of FSMDs defined in Chapter 2.

Definition 16 M_β is *contained* in M_α

M_β is contained in M_α, if for each computation μ_β of M_β, there exist a computation μ_α of M_α, such that $\mu_\beta \approx \mu_\alpha$.

Definition 17 *Equivalence* of two FSMDs ($M_\beta \cong M_\alpha$)

Two FSMDs M_β and M_α are equivalent, denote as $M_\beta \cong M_\alpha$, if M_β is contained in M_α and M_α is contained in M_β.

Theorem 2 M_β is contained in M_α if there exists a path cover $P_\beta = \{p_{\beta 0}, \ldots, p_{\beta n}\}$ of M_β and a set of path $P_\alpha = \{p_{\alpha 0}, \ldots, p_{\alpha n}\}$ of M_α such that $p_{\beta i} \approx p_{\alpha i}$ for all i, $0 \le i \le n$.

The proof of Theorem 2 is similar to it of Theorem 1.

Definition 18 *Corresponding state pair* (*CSP*)

1) The pair of the reset states, $q_{\beta 0} \in Q_\beta$ and $q_{\alpha 0} \in Q_\alpha$, is a native CSP.

2) The pair of states, $q_{\beta m} \in Q_\beta$ and $q_{\alpha n} \in Q_\alpha$, is a CSP if the states, $q_{\beta i} \in Q_\beta$ and $q_{\alpha j} \in Q_\alpha$, is a CSP and there exists paths, $\beta = q_{\beta i} \Rightarrow q_{\beta m}$ and $\alpha = q_{\alpha j} \Rightarrow q_{\alpha n}$, such that $\beta \approx \alpha$.

4.1.2 Notion

An example of two equivalent FSMDs, M_β and M_α, before and after scheduling employing Sp is shown in Fig. 14. Sp transforms M_β to M_α by moving "$b \Leftarrow x+y$" from "$q_{\beta 1} \longrightarrow q_{\beta 3}$" to "$q_{\beta 0} \longrightarrow q_{\beta 1}$". The reset states are $q_{\beta 0}$ and $q_{\alpha 0}$ and the initial path covers are {"$\beta 0 = q_{\beta 0} \longrightarrow q_{\beta 1}$", "$\beta 1 = q_{\beta 1} \xrightarrow[c]{} q_{\beta 2} \longrightarrow q_{\beta 4} \longrightarrow q_{\beta 0}$", "$\beta 2 = q_{\beta 1} \xrightarrow[!c]{} q_{\beta 3} \longrightarrow q_{\beta 4} \longrightarrow q_{\beta 0}$"} and {"$\alpha 0 = q_{\alpha 0} \longrightarrow q_{\alpha 1}$", "$\alpha 1 = q_{\alpha 1} \xrightarrow[c]{} q_{\alpha 2} \longrightarrow q_{\alpha 3} \longrightarrow q_{\alpha 0}$", "$\alpha 2 = q_{\alpha 1} \xrightarrow[!c]{} q_{\alpha 3} \longrightarrow q_{\alpha 0}$"}. Our algorithm first compares $\beta 0$ and $\alpha 0$, and then computes "$\varpi_{\beta 0} = < a, \mathbf{x+y}, x<y, d >$" $\neq$ "$\varpi_{\alpha 0} = < a, \mathbf{b}, x<y, d >$". It is obvious that only the final values of variable "b" are not equivalent. Since "b" is immediately used in $\beta 2$ succeeding $\beta 0$, it is an effective variable of $\beta 0$; thus, $\beta 0 \not\approx \alpha 0$. Therefore, our algorithm extends $\beta 0$ to $\beta 0\beta 1$ and $\beta 0\beta 2$, and then compares $\beta 0\beta 1$ and $\alpha 0\alpha 1$. It results that "$\varpi_{\beta 0\beta 1} = < x-y, \mathbf{x+y}, x<y, x-y+1 >$" $\neq \varpi_{\alpha 0\alpha 1} = < x-y, \mathbf{b}, x<y, x-y+1 >$". Similarly, our algorithm checks if "b" is an effective variable of $\beta 0\beta 1$ and $\alpha 0\alpha 1$. In M_β, "b" is immediately redefined by "$b \Leftarrow x+y$" in $\beta 0\beta 2$. Therefore, "b" is NOT an effective variable of $\beta 0\beta 1$. In M_α, there are only two computations: $\alpha 0\alpha 1$ and $\alpha 0\alpha 2$. Since $\alpha 0\alpha 1$ is not used or defined "b" and $\alpha 0\alpha 2$ always redefined "b" before using it, therefore, "b" is NOT an effective variable. As a result, we can conclude that $\beta 0\beta 1 \approx \alpha 0\alpha 1$.

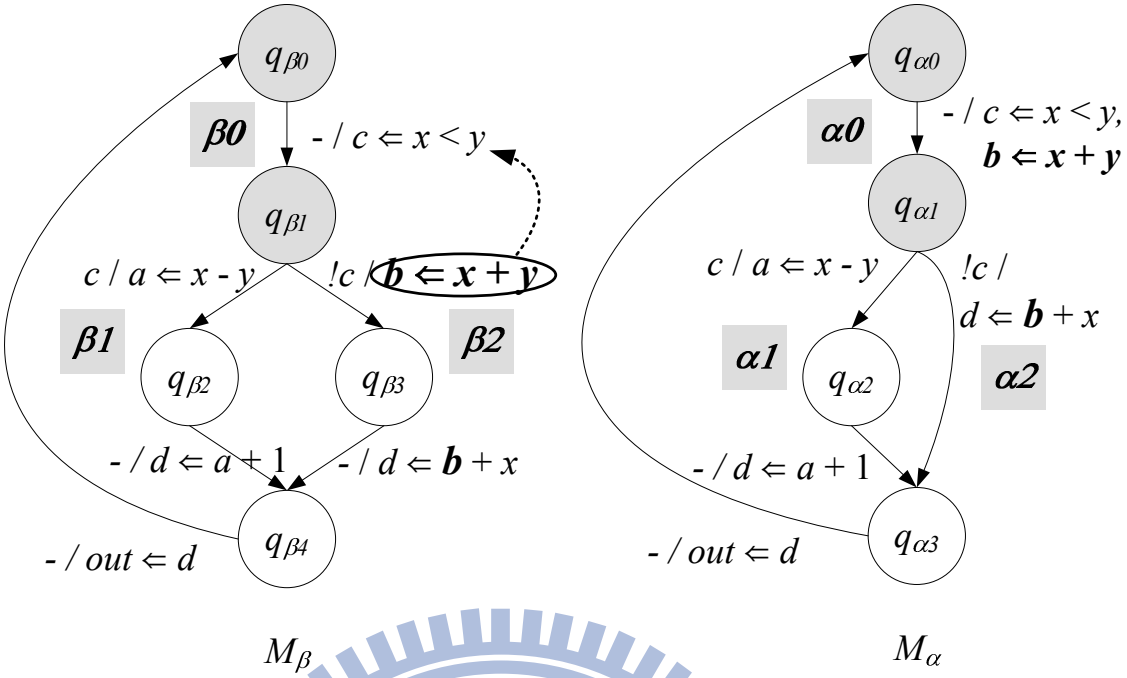


Fig. 14 An Example of Speculation

4.1.3 CE Algorithm

CE Algorithm checks whether a variable v is an effective variable of a path β of a path cover P in an FSMD M. Note that, it searches paths in M by breadth-first search (BFS) method. A state q is reachable from a state q_s means that there exists a walk starting from q_s and ending at q. Then BFS can discover all reachable states from q_s in M [31]. That is, BFS can discover all paths in M from the end state q_β of β.

CE Algorithm

Input: A variable v of a path β in a path cover P

Output: TRUE (v is NOT an effective variable of β) or FALSE

CheckEffect(v,β,P){

1 q_β = EndState(β);

```
2   // qβ is the source state where BFS starts from
3   llp = all paths starting from qβ in P;
4   tp = llp;
5   p = Pop(llp);
6   while(p){
7         if (p uses v)              return(FALSE);
8         else if(p defines v)       p = Pop(llp);
9         else{
10            qp = EndState(p);
11            pset = all paths starting from qp in P but not contained in tp;
12            tp = tp∪pset;
13            llp = llp∪pset;
14            p = Pop(llp);
15        }//endelse
16  }//endwhile
17  return(TRUE);
}//endCheckEffect
```

Proof of CE Algorithm

We prove CE Algorithm in two parts. The first part is for the termination and the second part is for the correctness.

1 (Termination)

CE Algorithm has only one loop, named p-while-loop. The termination of p-while-loop depends on the number of paths in *tp*. Since *tp* is a subset of *P*, CE Algorithm can always terminate.

2 (Correctness)

As line 7 returns FALSE, there exists a walk starting from q_β and using "v" before any defined "v". Therefore, "v" is an effective variable of β.

Assume, for the purpose of contradiction, that there exists a walk ω starting from q_β and using "v" before any defined "v" as CE Algorithm returns TRUE. Without loss of generality, let ω has only one path using "v", say p_v. Since line 7 is not executed, p_v is not in tp. Therefore, we can let $p_0 p_1 \ldots p_i$ be the first part of ω where each p_k is contained in tp for all k from 0 to i and let the succeeding path p_{i+1} of p_i is not in tp. According to line 8, p_i must define "v"; otherwise p_{i+1} should be in tp. Then we can conclude that "v" is not an effective variable as return TRUE.

Complexity of CE Algorithm

n = the number of states of M

e = the number of edges of M

ko = the number of outgoing edges of a state of M

In the worst case, p travels all edges; therefore, p-while-loop iterates at most e times. Line 11 scans all outgoing paths of a state and checks whether each path is contained in tp, it devotes $O(ko*e)$. Therefore, the complexity of CE Algorithm is $O(e^2*ko)$.

4.2 Solution for CSE

The equivalent problem introduced by CSE can be solved if we can record all possible available statements.

4.2.1 Available Statement

Definition 19 *Available statement st* of a state *q*

A statement *st*:*v*⇐*e* is available at a state *q* if

1) all possible walks from the reset state to *q* has *st* and

2) "*v*" and all used variables in *e* are not defined between the last *st* and *q*.

Fig. 15 illustrates an available statement of a state. q_0 is the reset state. "$q_0 \longrightarrow q_1 \longrightarrow q_2 \longrightarrow q_3 \longrightarrow q_4$" is the only one walk from q_0 to q_4 . Since "*b*" of "*s1*:*a*⇐*b*+*c*" is redefined in "$q_1 \longrightarrow q_2$", *s1* is NOT an available statement of the state q_2. On the contrary, "*s2*:*a*⇐*b*+*c*" is available at the state q_4 since the walk from q_0 to q_4 has *s2* and between *s2* and q_4, all variables of *s2* are not redefined.

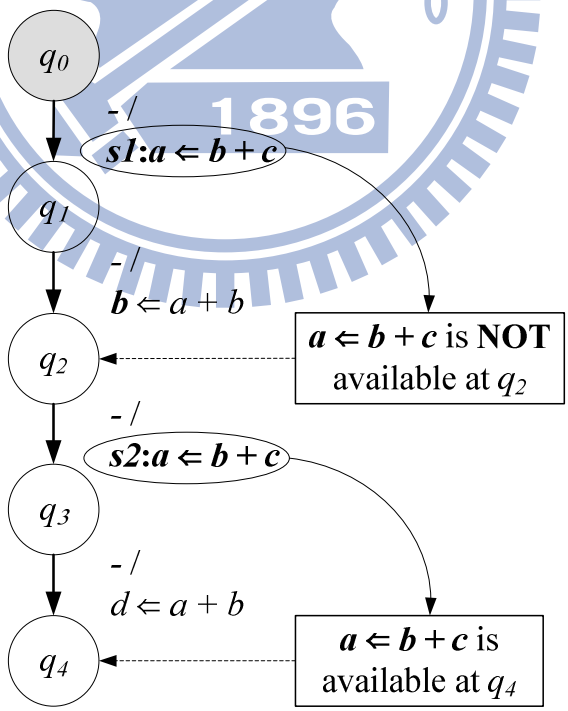


Fig. 15 An Example of an Available Statement

An available statement *st* of a state *q* holds a main property: all variables of *st* are

preserved the values from the last occurrence of *st* to *q*. Therefore, a system produces the same outputs as it re-computes *st* again between the last occurrence of *st* and *q*. The following lemma is derived.

Lemma 1

Let statement *st* be available at a state *q* in *M*. *M'* is transformed from *M* by two steps:

1. Add a new paths "$q \longrightarrow q'$" in *M* where "$q \longrightarrow q'$" has only a copy *st'* of *st*.
2. Change the start states of all path starting at *q* to *q'*.

Then $M \cong M'$.

We give a brief proof of Lemma 1. Since *st* is available at the state *q*, the values of all variables of *st* are preserved from the last occurrence of *st* to *q* for each possible computation μ having *q*. Therefore, let μ' of *M'* be the corresponding computation of μ. Clearly, they compute the same condition and the same result. Therefore, $\forall$ computation $\mu \in M$, $\exists \mu' \in M'$ such that $\mu \approx \mu'$, and vice versa. As the result, $M \cong M'$.

Fig. 16 gives an example of Lemma 1. *M'* is generated from *M* having an available statements "$e' \Leftarrow a+b$". Then, $M \cong M'$.

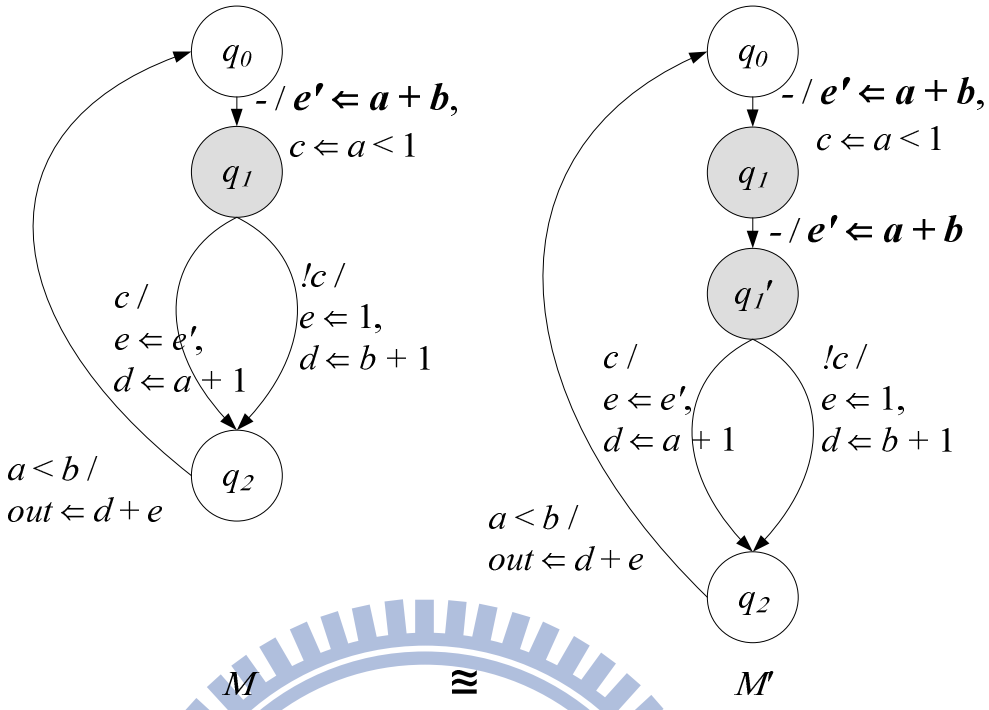


Fig. 16 An example of Lemma 1

Let M_β' and M_α' be transformed by Lemma 1 from M_β and M_α, respectively; then, $M_\beta \cong M_\beta'$ and $M_\alpha \cong M_\alpha'$. Therefore, if $M_\beta' \cong M_\alpha'$, $M_\beta \cong M_\alpha$.

4.2.2 Notion

Fig. 17 shows a solution for CSE. M_β and M_α are partial FSMDs. $V_\beta \cap V_\alpha = \{c, d, e, e', f\}$, $I = \{a, b\}$. $q_{\beta 0}$ and $q_{\alpha 0}$ are the reset states of M_β and M_α, respectively. Before finding all equivalent paths, our algorithm computes all available statements of each cutpoint of M_β and M_α. $in(q_{\beta 1})$ = {"$c \Leftarrow a<b$", "$e' \Leftarrow a+b$"} and $in(q_{\alpha 1})$ = {"$c \Leftarrow a<b$", "$e' \Leftarrow a+b$"} are the set of statements available at $q_{\beta 1}$ and $q_{\alpha 1}$, respectively. After computing all available statements, our algorithm compares $\beta 0$ and $\alpha 0$ and computes that "$\varpi_{\beta 0} = < a<b, d, e, a+b, f > = \varpi_{\alpha 0}$". Then it compares paths $\beta 1$ and $\alpha 1$ and computes that "$\varpi_{\beta 1} = < c, a+d, a+b, a+b > \neq \varpi_{\alpha 1} = < c, a+d, e', e' >$". According to Lemma 1, $in(q_{\beta 1})$ and $in(q_{\alpha 1})$ are thought of as paths; we concatenate

$in(q_{\beta 1})$ and $\beta 1$ and concatenate $in(q_{\alpha 1})$ and $\alpha 1$. As the result, $\varpi_{\beta 1} = < c<b, a+d, a+b , a+b > = \varpi_{\alpha 1}$ and $\beta 1 \simeq \alpha 1$.

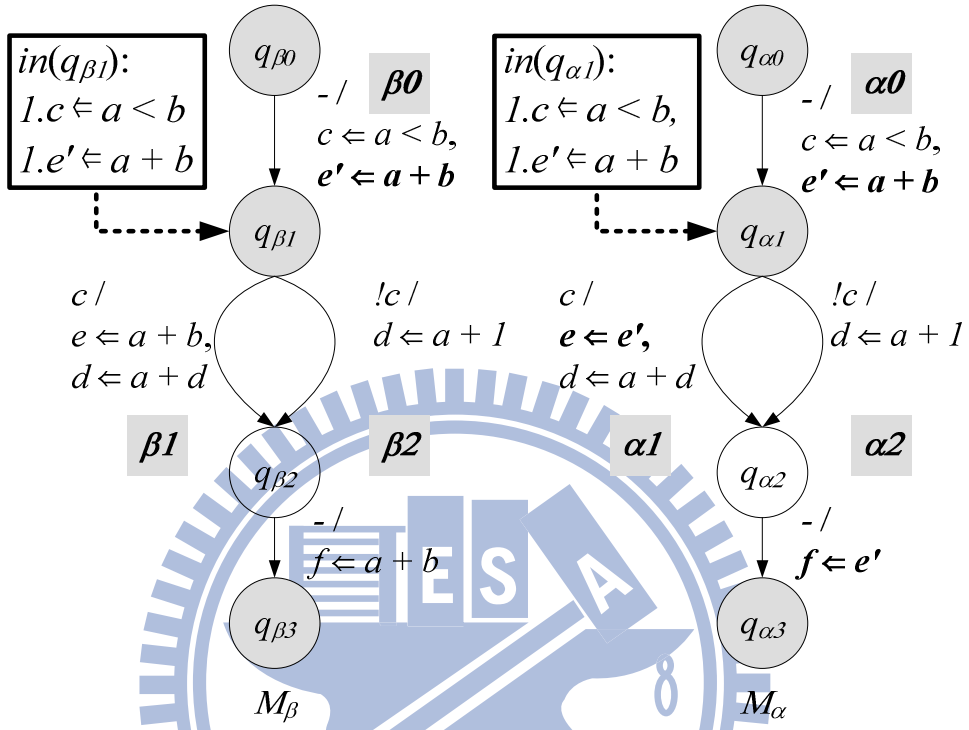


Fig. 17 A Solution for CSE

4.2.3 Compute Available Statements

For each path $\beta = q_{\beta i} \Rightarrow q_{\beta j}$ of an FSMD M, $gen(\beta)$, $in(\beta)$, $kill(\beta)$ and $out(\beta)$ are defined. $gen(\beta)$ is the set of statements of β which are available at the end state of β. At first, $gen(\beta)$ is computed for each path β of M. Therefore, three groups of equations can be created, shown in (3.1). $in(\beta)$ is the set of statements which are available at the start state $q_{\beta i}$ of β taking into account all the available statements of all possible walks starting from reset state to $q_{\beta i}$. It is the intersection of the sets consisting of possible available statements of all preceding paths of β. Oppositely, $kill(\beta)$ is a subset of $in(\beta)$ and contains all statements having some variables being redefined in β. $out(\beta)$ is the set of statements which are available at the end state of β. It is the union of $gen(\beta)$ and the statements in $in(\beta)$ but not in $kill(\beta)$.

1. $out(\beta) = gen(\beta) \cup (in(\beta) - kill(\beta))$
2. $in(\beta) = \cap_{\text{p is a preceding paths of } \beta} out(p)$ (3.1)
3. $in(\beta 0) = \emptyset$ where $\beta 0$ is the path starting from the reset state

Note that, $in(\beta)$ for the reset state is handled as a special case because nothing is available if the FSMD has just begun at the reset state. And more important, $in(\beta)$ uses intersection because an statement is available at the start state of β only if it is available at the end state of all preceding paths according to the definition of an available statement. All paths starting from $q_{\beta i}$ should have the same *in* set. Therefore, $in(q_{\beta i}) = in(\beta)$ for each β starting at $q_{\beta i}$.

Fig. 18 gives an example of how to compute available statements of cutpoints. M_α is a partial FSMD having three cupoints ($q_{\alpha 0}$, $q_{\alpha 2}$, $q_{\alpha 3}$) and three initial paths ($\alpha 0 = q_{\alpha 0} \longrightarrow q_{\alpha 1}$, $\alpha 1 = q_{\alpha 1} \xrightarrow[c]{} q_{\alpha 2} \longrightarrow q_{\alpha 3}$, $\alpha 2 = q_{\alpha 1} \xrightarrow[!c]{} q_{\alpha 2} \longrightarrow q_{\alpha 3}$). At first, our algorithm compute $gen(\alpha 0)$, $gen(\alpha 1)$ and $gen(\alpha 2)$. As computing $gen(\alpha 1)$, there are two time steps (i.e. two transitions) needed to be computed. The first transition contains two statements, "$e \Leftarrow e'$ " and "$d \Leftarrow a+d$ ". Since "$d \Leftarrow a+d$ " redefines "d " by itself, it is not available at $q_{\alpha 3}$ and can not be contained in $gen(\alpha 1)$. And in the second transition, all its statements are available at $q_{\alpha 3}$. Our algorithm cascades statements with their order, and then $gen(\alpha 1)$ consists of "$e \Leftarrow e'$ " having order 1, "$c \Leftarrow d+e$" having order 2, and "$g \Leftarrow e'$ "having same order 2. It is worth to notice that the time is preserved by recording the order of statements and the statements with the same order are executed simultaneously. Next, our algorithm computes *in* and *out* of each path. Since $\alpha 0$ starts from reset state, $in(q_{\alpha 0}) = in(\alpha 0) = \emptyset$ and $out(\alpha 0) = gen(\alpha 0)$. Because $\alpha 0$ is the only preceding path of $\alpha 1$ and $\alpha 2$, this derives that $in(q_{\alpha 1}) = in(\alpha 1) = in(\alpha 2) = out(\alpha 0)$. As calculating $out(\alpha 1)$, "c" of statement "$c \Leftarrow a<b$" in $in(\alpha 1)$ is redefined in $\alpha 1$; thus, $out(\alpha 1)$ is the union of $gen(\alpha 1)$ and the rest statements of $in(\alpha 1)$, "$e' \Leftarrow a+d$ ". $out(\alpha 2)$ is computed in the similar way. Finally, our algorithm computes $in(q_{\alpha 3})$. Since there are only two preceding paths $\alpha 1$ and $\alpha 2$ of $q_{\alpha 3}$, $in(q_{\alpha 3})$ is the intersection of $out(\alpha 1)$ and $out(\alpha 2)$. From $out(\alpha 1)$, we find all

equivalent statements in *out*(*α2*) in order, then *in*($q_{α3}$) = {"*e'*⇐*a*+*b*", "*c*⇐*d*+*e*", "*g*⇐*e'* "}.

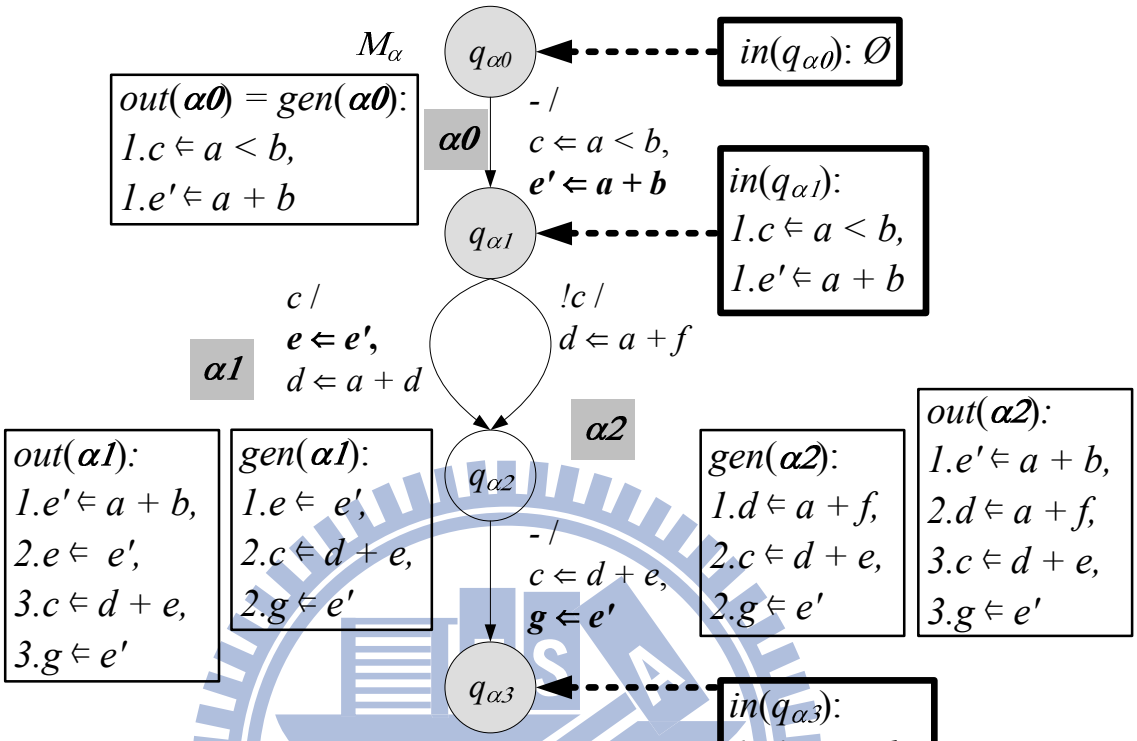


Fig. 18 Compute Available Statements

4.2.4 FAS Algorithm

The algorithm of computing all available statements in an FSMD *M* is shown below. Note that the initial paths have been sorted by BFS and the *gen* set of each path has been calculated.

FAS Algorithm

Input: An FSMD *M* with reset state q_0 and initial path cover P_0

gen(*β*) has been computed for each initial path

Output: *in*(*β*) set for each initial path

```
FindAvailableStatement(M){
1   for(all paths p of P0){
2       p.travel = FALSE;
3       p.oldout = Ø;
4       in(p) = Ø;
5       out(p) = Ø;} //endfor
6   Qp = a queue of paths of P0 sorted by BFS and BFS starts from q0;
7   change = TRUE;
8   while(change){
9     llp = Copy(Op);
10    pp = Pop(llp);
11    change = FALSE;
12    while(pp){
13      oldout = out(pp);
14      qs = StartState(pp);
15      if(qs is reset state)
16        out(pp) = gen(pp);
17      else{
18        if(some preceding path are not traveled)
19          in(pp) = ∩ p is a preceding paths of pp and p.travel is TRUE out(p);
20        else
21          in(pp) = in(pp) ∩ p is a preceding paths of pp out(p);
22        out(pp) = gen(pp) ∪ (in(pp) - kill(pp));}//endelse
23      pp.travel = TRUE;
24      if(out(pp) ≠ oldout) change = TRUE;
25      pp = Pop(llp);
```

26 }//endwhile(*pp*)

27 }//endwhile(*change*)

}//endFindAvailableStatement

Proof of FAS Algorithm

We prove FAS Algorithm in two parts.

1 (Termination)

FAS Algorithm has only two loops, change-while-loop and pp-while-loop. Pp-while-loop depends on the number of paths in *llp*. Since *llp* is the initial path cover, *llp* is finite and pp-while-loop can terminate for each iteration of change-while loop.

Then prove change-while-loop can terminate. A walk starting from the reset set is called an r-walk. Change-while-loop depends on the number of all possible r-walks. The paths starting from the reset state have the constant *out* set, and then each other path can compute the constant *in* set from an intersection of the constant *out* sets of all possible preceding r-walks. Let *Wp* be the set consisting of those possible preceding r-walks for a path.

Let *l1* be a loop and let *B* be an *in* set of *l1*. Let's pass *B* through *l1* and compute the *out* set *B'* of *l1* with the rule (3.1). Clearly, as passing $B \cap B'$ through *l1* for any number of runs, even infinite, we always compute the same constant *B'*. In other words, let *Wf* be a subset of *Wp* and each walk, called rf-walk, of *Wf* has either no loop or only one. The intersection of the *out* set of all walks in *Wp* is same as it in *Wf*. Therefore, the constant *in* set of each path is an intersection of the constant *out* sets of finite walks.

Since *llp* is the initial path cover sorted by BFS, after the first run of pp-while-loop, each path has an initial *in* set of an r-walk or an intersection of some r-walks. After a finite runs of change-while loop, the constant *out* sets of all preceding r-walks containing no loop for a path are computed. (An r-walk containing no loop may have reverse order because of BFS; for example, let an r-walk be *p1p2p3*, and after BFS, the order of *p3* is prior to *p2* in *llp*.

Therefore, the r-walk needs at most 2 runs of change-while loop to compute the constant *out* set.) Similarly, all preceding r-walks of a path containing only one loop can compute the constant *out* sets after finite runs of change-while loop. Since there are finite r-walks, change-while-loop can terminate.

2 (Correctness)

Since a path has only finite preceding rf-walks and each rf-walk can compute the constant *out* set after finite runs of change-while loop, all available statements of a state can be computed. Assume, for the purpose of contradiction, that there is a statement *st* which is not available at the start state of a path *p*, but is in the *in* set of *p*. It means that some preceding rf-walks don't have *st* or a variable of *st* is redefined in a preceding rf-walk. In the former case, the *out* sets of some rf-walks don't contain *st* and the process in line 19 or line 21 will remove it. In the latter case, *st* will be killed by line 22 as compute the *out* sets of the paths defining *st*. If their original *out* sets has *st*, *change* will be set to TRUE. As a result, all statements in the *in* set of *p* are available at the start state of *p*.

Complexity of FAS Algorithm

n = the number of states of *M*

e = the number of edges of *M*

ki = the largest indegree of all states of *M*

An iteration of pp-while-loop takes $O(ki)$ and pp-while-loop has at most *e* iterations. Change-while-loop depends on the largest number of the reverse order of all r-walks and the longest loop; it has $O(e)$ runs. Therefore, the complexity of FAS Algorithm is $O(e^2*ki)$.

4.3 Our Algorithm

According to the Theorem 2 and Definition 17, M_β is contained M_α if there exists a path cover $P_\beta = \{ p_{\beta 0}, \ldots , p_{\beta n} \}$ of M_β and a set of path $P_\alpha = \{ p_{\alpha 0}, \ldots , p_{\alpha n} \}$ of M_α such that $p_{\beta i} \approx p_{\alpha i}$ for all i, $0 \le i \le n$. If a set of path P_α of M_α is also a path cover of M_α, it implies that $M_\beta \cong M_\alpha$.

Lemma 2 $M_\beta \cong M_\alpha$ if there exists a path cover $P_\beta = \{ p_{\beta 0}, \ldots , p_{\beta n} \}$ of M_β and a path cover $P_\alpha = \{ p_{\alpha 0}, \ldots , p_{\alpha n} \}$ of M_α such that $p_{\beta i} \approx p_{\alpha i}$ for all i, $0 \le i \le n$.

Since extending a path means to find a new path cover, we insert the cutpoints into not only M_β but also M_α; then, we obtain initial path covers P_β and P_α of M_β and M_α, respectively. For each path β of P_β, our algorithm attempts to find an equivalent path in M_α by finding all possible paths in M_α depending on CSP. Let β find an equivalent path α' of M_α where $\alpha' = \alpha 1 \alpha 2$ and $\alpha 1$ and $\alpha 2$ are initial paths. In this case, at first, β can't find an equivalent path from the initial path cover of M_α, and then our algorithm extends $\alpha 1$ to find a new path cover, P_α. Therefore, β can find an equivalent path $\alpha 1 \alpha 2$ from P_α. If each path of the final P_β has an equivalent path of the final P_α and each path of the final P_α has an equivalent path of the final P_β, $M_\beta \cong M_\alpha$.

Fig. 19 illustrates the overview of our algorithm and the detail is depicted in Fig. 20. Our algorithm reads two FSMDs, M_β and M_α, as the inputs and produces a set of equivalent paths, E, as an output. Initially, the set of equivalent paths E is empty and the set of working list L contains no path waiting to find an equivalent path in M_α. Let "$(q_{\beta 0}, q_{\alpha 0})$" be the only one member of corresponding state pair set, i.e. $CSP_{\beta\alpha} = \{(q_{\beta 0}, q_{\alpha 0})\}$. After the initialization, our algorithm inserts the cutpoints into both M_β and M_α. Subsequently, it finds initial path cover $P_{\beta 0}$ of M_β and $P_{\alpha 0}$ of M_α. Next, it finds all possible available statements, i.e. the *in* sets of

each cutpoint in both M_β and M_α. Hence, it is ready for finding equivalent paths.

First, our algorithm finds "$\beta = q_{\beta i} \Rightarrow q_{\beta j}$" from $P_{\beta 0}$ depending on "$(q_{\beta i}, q_{\alpha m})$" which is in $CSP_{\beta\alpha}$, and then it finds an equivalent path, starting from $q_{\alpha m}$, of $P_{\alpha 0}$ in M_α. If an equivalent path "$\alpha = q_{\alpha m} \Rightarrow q_{\alpha n}$" is found, it records the equivalent path pair "(β, α)" in E and adds "$(q_{\beta j}, q_{\alpha n})$" to $CSP_{\beta\alpha}$. Subsequently, it removes β from $P_{\beta 0}$ and α from $P_{\alpha 0}$. Checking two paths are equivalent or not may have three situations. In the first situation, they are exactly equivalent. In the second situation, if they are not equivalent because of some final values of variables, our algorithm checks those variables by CE Algorithm. If all of them are not effective variables, β finds an equivalent path. Otherwise, in the third situation, according to Lemma 1, our algorithm generates β' by concatenating $in(q_{\beta i})$ and β, and it also generates each α' by concatenating $in(q_{\alpha m})$ and each α in M_α. It compares β' with each α'. If the equivalent path of β' is not found, it extends β (go to step 11). Otherwise, β finds an equivalent path (go to step 10). Our algorithm repeats the process (GetEquivalentPath) until all paths of $P_{\beta 0}$ finding an equivalent path in M_α. Finally, check if $P_{\alpha 0}$ is empty. If it is, it implies that all paths in E constitute path covers of M_β and M_α. Hence, $M_\beta \cong M_\alpha$.

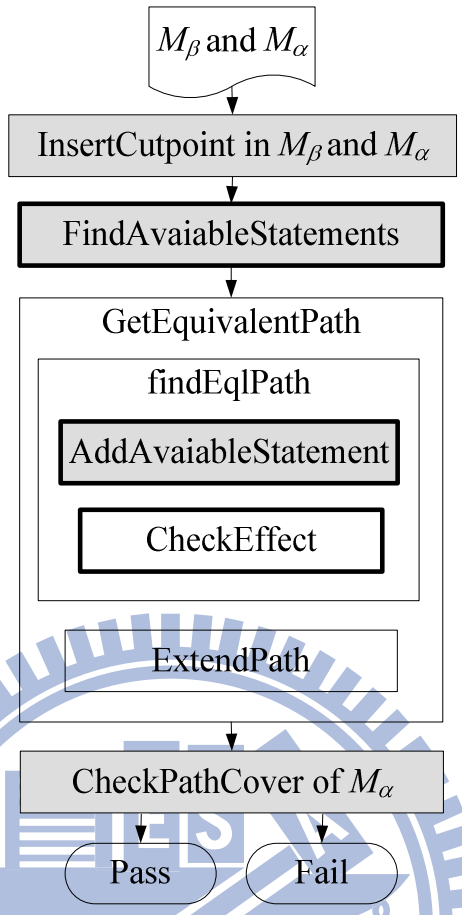


Fig. 19 An Overview of Our Algorithm

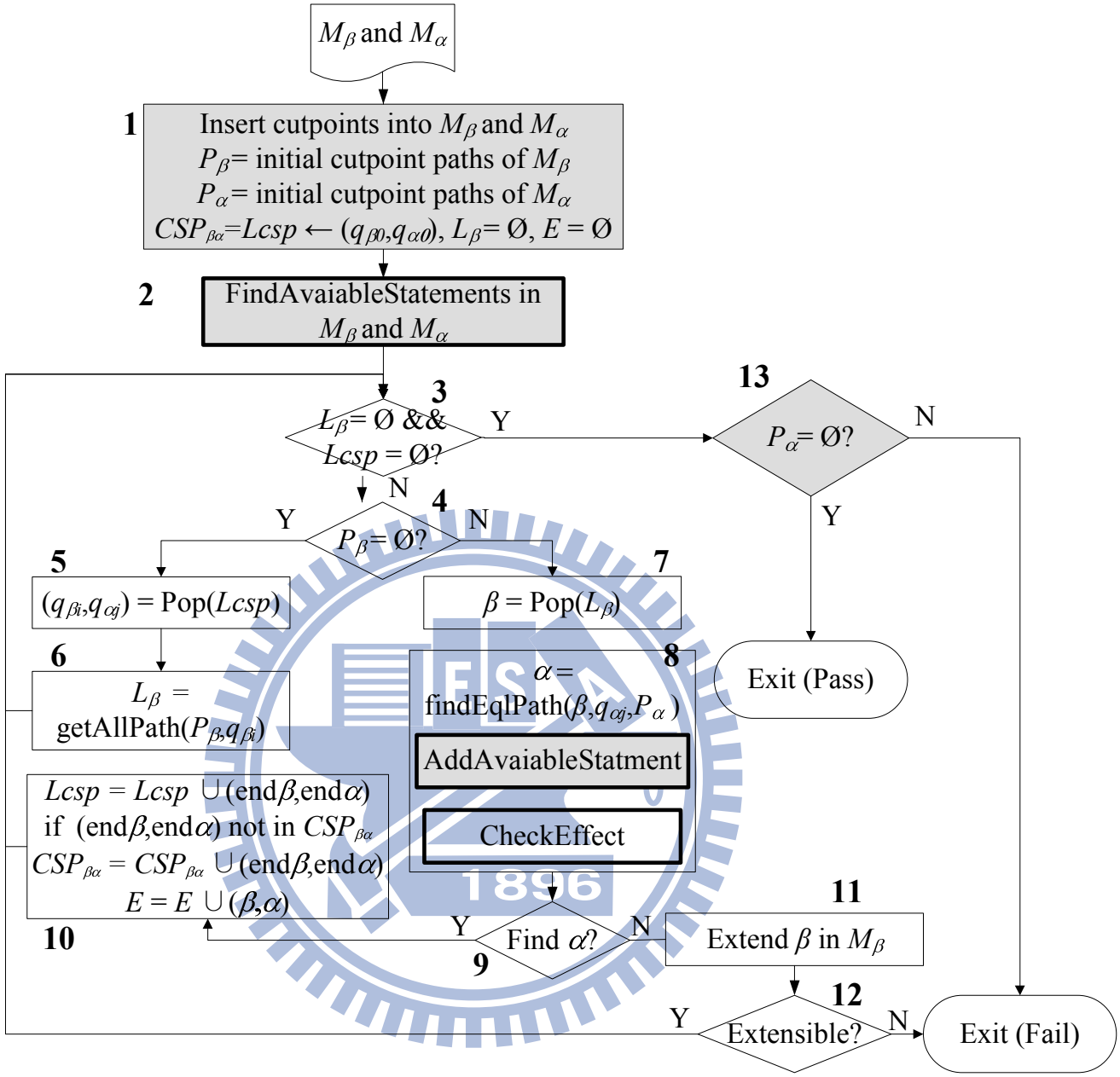


Fig. 20 Our Proposed Algorithm

Proof of our algorithm

Obviously, the termination of our algorithm depends on the number of paths needed to be checked and the number of CSPs of $CSP_{\beta\alpha}$. Since there are finite cutpoints and finite initial paths in both M_β and M_α, the combinations of CSPs are finite and our algorithm can only extend finite times to generate finite concatenated paths. Therefore, our algorithm can always

terminate.

Since step 8, 9 and 10 of our algorithm in Fig. 20 ensure that E contains only pairs of equivalent paths of P_β and P_α and as extending a path, our algorithm concatenates all relevant paths to build a new path cover and removes the relevant cutpoint and paths from E, P_β, and P_α; therefore, all paths of E has no cutpoint be an internal state and all are the paths starting from a cutpoint and ending at a cutpoint. As a result, as our algorithm executes successfully, all equivalent paths are contained in E and they constitute the path covers of M_β and M_α.

Complexity

n = the number of states of M

e = the number of edges of M

ki = the largest indegree of all states in M

ko = the largest outdegree of all states in M

The complexities of FAS Algorithm takes $O(e^2*ki)$. Let the complexity of comparing two statements to be $|F|$. As findEqlPath, a path in M_α can extend at most n times and each path extension needs to concatenate ($ki*ko$) paths. Therefore, the complexity of findEqlPath is $O(n*ki*ko*|F|)$. Note that $|F|$ usually takes much longer time than CheckEffect and AddAvailableStatement. When check if the new CSP is in the $CSP_{\beta\alpha}$, it has the n^2 possible combinations. In the worst case, one path is extended n times. There are $ki*ko*(n-1) + ki*ko^2*(n-1)*(n-2) + \ldots + ki*ko^{(n-1)}*(n-1)*(n-2)\ldots 2.1 \approx ki*ko^{(n-1)}*(n-1)^{(n-1)}$ number of paths. Therefore, the complexity of our algorithm is $O(n^n*ki^2*ko^n*|F|)$.

In the best case, each path in M_β can directly find an equivalent path in M_α. No path extension and CheckEffect are needed. Therefore, the complexity is $O(e^2*ki+n*|F|) \approx O(n*|F|)$.

Chapter 5
Experimental Results

Our equivalence checking algorithm and Karfa's algorithm [26] have been implemented in C. We compare two paths with symbolic execution. All benchmarks have been run on a 1.86 GHz Intel Core 2 CPU PC with 2 GB RAM. The run time of all benchmarks is less than 2 seconds. One benchmark, *diffeq*, is data intensive; some are control intensive, such as *gcd*, *barcaode*, *tlc* and *lru* ; some are control and data intensive, such as *mode* and *kalman* [32][33]. *test1* and *test2*, which are built by ourselves, are control and data intensive. The results shown in tables are sorted by our run time.

Table I, Table III and Table V list the characteristics of the benchmarks in terms of the number of states in M_β, states in M_α, variables in $V_\beta \cap V_\alpha$, statements in M_β, and statements in M_α. Table II, Table IV and Table VI show the verification results. They list in terms of the number of initial cutpoints in M_β, initial paths in M_β, initial cutpoints in M_α, initial paths in M_α, path extensions in M_β with the iterations back to step3, and equivalent paths and in terms of the employed scheduling techniques for each case, the run time of our algorithm, and the run time of Karfa's algorithm.

Table I and Table II are the benchmarks where those cases are all equivalent cases and scheduled by PBS or SPARK [15] with removing the SSp in manually. Therefore, our algorithm and Karfa's can handle the cases. Table III and Table IV are the equivalent cases scheduled by SPARK except *test1* and *test2* which are transformed manually. Karfa's algorithm does not support Sp, RSp, SSp and CSE; therefore, it fails to all the cases in Table IV. The cases in Table V and Table VI are not equivalent. Each case is added error manually. *barcode_err* and *kalman_err* are transformed from *barcode* and *kalman* by adding an improper MU; *mode_err* removes one operation from M_α; *lru_err* changes the end state of a

path of M_α; *test2_err* has an improper CSE. Our algorithm can find that they are not equivalent.

For most cases, run time depends mainly on the numbers of iterations and path extensions. The run time of *test2_err* is larger than *kalman_err* because the algorithms extend 80 paths in *kalman_err* as they extend 524 paths in *test2_err*. The number of iterations contained in a pair of braces is the number of paths compared by the algorithms. The number of path extensions is the number of failed paths as finding equivalent paths. Since our algorithm only runs one pass, our run time of *barcode* of Table II is less than Karfa's. However, if two paths are not equivalent in findEquivalentPath, our algorithm performs CheckEffect or/and AddAvaiableStatement, and then compares again. Besides, FindAvaiableStatement also consumes time. Therefore, our average run time is lager than Karfa's about 2 times.

The experimental results, including one high complexity case *kalman*, indicate that our algorithm is usable for verifying BB-based scheduling, PBS, code motions and CSE.

Abbreviations of each scheduling technique:

PBS: Path-based scheduling,

MU: Merging up, MD: Merging down

DU: Duplicating up, DD: Duplicating down, UM: Useful move,

Sp: Speculation, RSp: Reverse speculation, SSp: Safe-speculation,

CSE: Common subexpression elimination.

Table I Characteristics of Equivalent Cases 1

case	$\#Q_\beta$	$\#Q_\alpha$	#variables in $V_\beta \cap V_\alpha$	$\#statements_\beta$	$\#statements_\alpha$
barcode	9	6	4	12	16
gcd	7	3	3	11	11
tlc	11	12	5	29	31
modn	6	4	5	11	21
lru	23	22	13	18	26
kalman	105	101	64	104	112

Table II Results of Equivalent Cases 1

Case	#cutpoints in M_β	$\#P_\beta$	#cutpoints in M_α	$\#P_\alpha$	# of Path Extensions (iterations)	$\#E$	scheduling	Ours		[26]	
								P/F	Time (ms)	P/F	Time (ms)
barcode	5	10	5	10	2(10)	8	DD,UM,MU	P	24.9	P	31.8
gcd	6	11	2	7	4(11)	7	PBS	P	30.4	P	27.4
tlc	10	19	10	19	1(19)	18	DU	P	36.4	P	33.5
modn	5	9	4	10	2(18)	11	PBS	P	39.8	P	29.2
lru	21	41	21	41	0(41)	41	DU	P	47	P	44.9
kalman	101	202	101	202	32(202)	170	DU,UM	P	481.5	P	215.5

Table III Characteristics of Equivalent Cases 2

case	$\#Q_\beta$	$\#Q_\alpha$	#variables in $V_\beta \cap V_\alpha$	$\#statements_\beta$	$\#statements_\alpha$
test1	8	8	8	12	12
diffeq	7	7	12	15	20
gcd	4	5	2	4	9
barcode	9	10	4	12	27
tlc	11	12	5	29	44
lru	23	40	13	18	79
test2	29	21	23	43	45
kalman	105	150	64	104	279

Table IV Result of Equivalent Cases 2

case	$\#C_\beta$	$\#P_\beta$	$\#C_\alpha$	$\#P_\alpha$	# of Path Extensions (iterations)	$\#E$	scheduling	Ours		[26]	
								P/F	Time (ms)	P/F	Time (ms)
test1	3	5	3	5	1(5)	4	**Sp**,**RSp**,**CSE**	P	22.9	F	24.7
diffeq	2	3	2	3	0(3)	3	**SSp**	P	23.8	F	22.6
gcd	4	7	4	7	0(7)	7	**SSp**	P	24.4	F	19.5
barcode	5	10	6	11	2(10)	8	**SSp**,DD,UM,MU	P	38.1	F	26.8
tlc	10	19	10	19	1(19)	18	**SSp**,DU	P	46.1	F	29.5
lru	21	41	21	41	2(41)	39	**SSp**,DU	P	85.5	F	40.1
test2	9	17	9	17	5(29)	24	**Sp**,**RSp**,**SSp**,**CSE**, UM,MU,MD,DU,DD	P	111.7	F	553.5
kalman	101	202	102	203	34(202)	168	**SSp**,DU,UM	P	1437	F	182.8

Table V Characteristics of Not Equivalent Cases

case	$\#Q_\beta$	$\#Q_\alpha$	#variables in $V_\beta \cap V_\alpha$	$\#statements_\beta$	$\#statements_\alpha$
barcode_err	9	6	4	12	15
mode_err	6	7	5	11	26
lru_err	23	22	13	18	26
kalman_err	105	101	64	104	110
test2_err	28	21	23	43	44

Table VI Results of Not Equivalent Cases

case	$\#C_\beta$	$\#P_\beta$	$\#C_\alpha$	$\#P_\alpha$	# of Path Extensions (iterations)	Ours		[26]	
						P/F	Time (ms)	P/F	Time (ms)
barcode_err	5	10	5	11	4(10)	F	33.5	F	28.3
mode_err	4	8	4	12	2(15)	F	39.2	F	25.9
lru_err	21	41	21	41	3(44)	F	52.9	F	38.4
kalman_err	101	202	101	203	19(56)	F	273	F	123.6
test2_err	9	17	9	17	9(9)	F	562	F	257.4

Chapter 6
Conclusion & Future Works

In this thesis, a formal verification method is proposed for the scheduling verification in HLS. This method is capable of BB-based scheduling and PBS. It is also well suited to verify some popular code transformation techniques: DD, DU, MD, MU, UM, Sp, RSp, SSp, and CSE. But it still not supports some code transformation techniques, such as loop invariant and copy propagation.

Fig. 21 shows an example of loop invariant. Since all variables of the statement "$c \Leftarrow x+y$" are not modified in the loop ($q_{\beta i1} \longrightarrow q_{\beta i2} \longrightarrow q_{\beta i1}$), loop invariant technique moves "$c \Leftarrow x+y$" out from the loop. Our algorithm fails to this situation. Since the path ending at $q_{\beta i1}$ and the path ending at $q_{\alpha j1}$ are not equivalent, our algorithm needs to extend the path. After the path extension, i.e. removing the cutpoints $q_{\beta i1}$ and $q_{\beta i2}$, it becomes a walk which is not a path. Therefore, our algorithm fails.

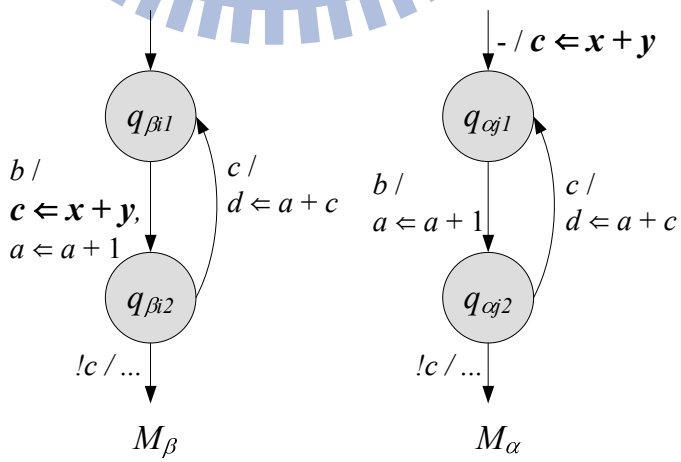


Fig. 21 An Example of Loop Invariant

Copy propagation is derived from the compiler. The statements in "$g \Leftarrow h$" form are called copy statements where "g" and "h" are variables. Copy propagation replaces "g" with "h" in all the statements that have flow dependencies with "$g \Leftarrow h$". It is illustrated in Fig. 22. Since "$a \Leftarrow x+y$" and "$c \Leftarrow x+y$" have the common subexpression, CSE replaces the expression of "$c \Leftarrow x+y$" with "a". Then, copy propagation finds "$d \Leftarrow c$" having data dependence with "$c \Leftarrow a+b$" and it replaces "c" with "a". Our approach fails to this case. Since start from the CSP "$(q_{\beta i2}, q_{\alpha j2})$", the final values of "$d$ " of "$q_{\beta i2} \longrightarrow q_{\beta i3}$" and "$q_{\alpha i2} \longrightarrow q_{\alpha i3}$" are not equivalent because "$c \neq a$".

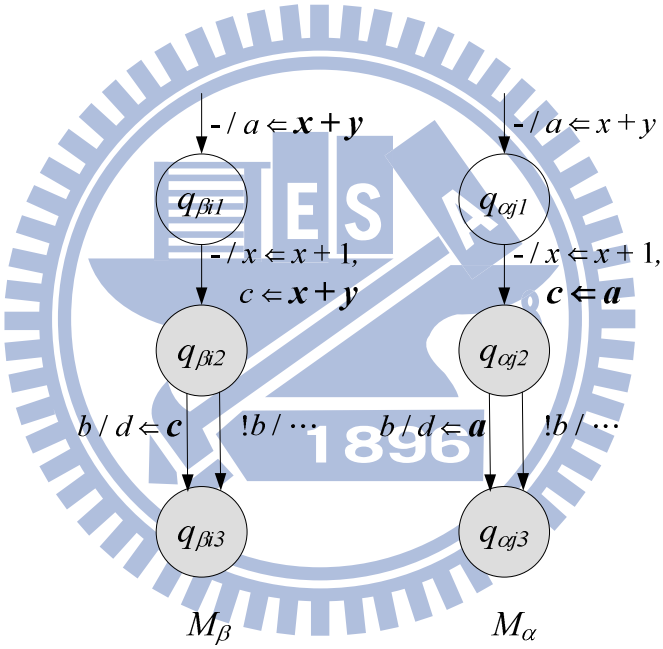


Fig. 22 An Example of Copy Propagation

References

[1] D. D. Gajski, N.D. Dutt, A.C.-H. Wu, and S.Y.-L. Lin, “High-Level Synthesis: Introduction to Chip and System Design,” Kluwer, 1992.

[2] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni, “Behavioral Synthesis and Component Reuse with VHDL,” Kluwer, 1997.

[3] P. Coussy, D.D. Gajski, M. Meredith, and A. Takach, “An Introduction to High-Level Synthesis,” IEEE Design & Test of Computers, vol. 26, page 8-17, July-Aug. 2009.

[4] C. Y. Hitchcock and D.E. Thomas, “A Method of Automatic Data Path Synthesis,” Design Automation Conference, page 484-489, Jun. 1983.

[5] B. M. Pangrle and D.D. Gajski, “Slicer: A State Synthesizer for Intelligent Silicon Compilation,” IEEE International Conference Computer Design: VLSI in Computers & Processors, Oct. 1986.

[6] P. G. Paulin and J.P. Knight, “Force-directed Scheduling for the Behavioral Synthesis of ASIC’s,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, page 661-679, Jun. 1989.

[7] R. Camposano, “Path-based Scheduling for Synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, page 85-93, Jan. 1991.

[8] G. Lakshminarayana, A. Raghunathan, and N.K. Jha, “Incorporating Speculative Execution into Scheduling of Control-flow Intensive Behavioral Descriptions,” Design Automation Conference, page 108-113, Jun. 1998.

[9] L.C.V. dos Santos and J.A.G. Jess, “A Reordering Technique for Efficient Code Motion,” Design Automation Conference, page 296-299, Jun. 1999.

[10] M. Rim, Y. Fann, and R. Jain, “Global Scheduling with Code-motions for High-level Synthesis Applications,” IEEE Transactions on VLSI Systems, vol. 3, page 379-392,

Sept. 1995.

[11] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Dynamic Conditional Branch Balancing during the High-level Synthesis of Control-intensive Designs,” Design, Automation and Test in Europe Conference and Exhibition, vol. 1, page 270-275, Dec. 2003.

[12] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Spark: A High-level Synthesis Framework for Applying Parallelizing Compiler Transformations,” International Conference on VLSI Design, page 461-466, Jan. 2003.

[13] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Loop Shifting and Compaction for the High-level Synthesis of Designs with Complex Control Flow,” Design, Automation and Test in Europe Conference and Exhibition, vol. 1, page 114-119, Feb. 2004.

[14] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, “Using Global Code Motions to Improve the Quality of Results for High-level Synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, page 302-312, Feb. 2004.

[15] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, “Spark: A Parallelizing Approach to the High-level Synthesis of Digital Circuits,” Kluwer, 2004.

[16] R. Ernst and J. Bhasker, “Simulation-based Verification for High-Level Synthesis Applications,” IEEE Design & Test of Computers, vol.8, page 14-20, Mar. 1991.

[17] R. A. Bergamaschi and S. Raje, “Observable Time Windows: Verifying High-Level Synthesis Results,” IEEE Design & Test of Computers, vol. 14, page 40-50, April-Jun. 1997.

[18] T.-H. Chiang and L.-R. Dung, “Verification Method of Dataflow Algorithms in High-Level Synthesis,” Journal of Systems and Software, vol. 80, page 1256-1270, Aug. 2007.

[19] N. Narasimhan, E. Teica, R. Radhakrishnan, S. Govindarajan, and R. Vemuri, “Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis,” Formal Methods in System Design, vol. 19, page 237-273,

Nov. 2001.

[20] R. Radhakrishnan, E. Teica, and R. Vemuri, “An Approach to High-Level Synthesis Validation using Formally Verified Transformations,” IEEE International High-Level Design Validation and Test Workshop, page 80-85, August-Oct. 2000.

[21] R. Radhakrishnan, E. Teica, and R. Vemuri, “Verification of Basic Block Schedules using RTL Transformations,” Lecture Notes in Computer Science, vol. 2144, page 173-178, Jan. 2001.

[22] N. Mansouri and R. Vemuri, “A Methodology for Automated Verification of Synthesized RTL Designs and Its Integration with a High-Level Synthesis Tool,” Lecture Notes in Computer Science, vol. 1522, page 204-221, Jan. 1998.

[23] Y. Kim , S. Kopuri, and N. Mansouri, “Automated Formal Verification of Scheduling Process using Finite State Machines with Datapath (FSMD),” International Symposium on Quality Electronic Design, page 110-115, Aug. 2004.

[24] H. Eveking, H. Hinrichsen, and G. Ritter, “Automatic Verification of Scheduling Results in High-Level Synthesis,” Design, Automation and Test in Europe Conference and Exhibition, page 59-64, Mar. 1999.

[25] Y. Kim and N. Mansouri, “Automated Formal Verification of Scheduling with Speculative Code Motions,” Great Lakes symposium on VLSI, page 95-100, 2008.

[26] C. Karfa, D. Sarkar, C. Mandal, and P. Kumar, “An Equivalence-Checking Method for Scheduling Verification in High-Level Synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, page 556-569, Mar. 2008.

[27] S. Owre, J. M. Rushby, and N. Shankar, “PVS: A Prototype Verification System,” Lecture Notes in Computer Science, vol. 607, page 748-752, Jun. 1992.

[28] P. G. Paulin and J. P. Knight, “Scheduling and Binding Algorithms for High-Level Synthesis,” Design Automation Conference, page 1-6, Jun. 1989.

[29] E. Teica and R. Vemuri, “A Mechanical Proof of Completeness for a Set of Register-Level Transformation,” Technical Report 257/05/01/ECECS, University of Cincnnati, 2001.

[30] Z. Manna, “Mathematical Theory of Computation,” McGraw-Hill, 1974.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms,” McGraw-Hill, 2001.

[32] P. R. Panda and N. D. Dutt, “1995 high level synthesis design repository,” International Symposium on System Synthesis, page 170-174, Sept. 1995.

[33] http://computing.ece.vt.edu/~mhsiao/hlsyn.html