Practical Techniques and Applications of Binary Decision Diagrams in Property Verification Problems

プロパティ検証問題における二分決定グラフの実用的な技術と応用

Hiroaki Iwashita

岩下 洋哲

December 20, 2013

Hokkaido University

Graduate School of

Information Science and Technology

Acknowledgments

This thesis would not happen to be possible unless the help and support of the kind people around me. First of all, I would like to express my appreciation to my supervisor, Prof. Shin-ichi Minato, for his excellent guidance and for providing me with a superb atmosphere for the research. I would also like to thank my sub-supervisors, Prof. Hiroki Arimura and Prof. Yasuyuki Shirai, who gave me great advices to complete the doctoral dissertation.

I cannot find words to express my gratitude to Prof. Isao Shirakawa, who was my supervisor during my senior undergraduate and master courses at Osaka University. He showed me the basics and the fun of doing research.

I share the credit of my work at Fujitsu Laboratories with Mr. Fumiyasu Hirose and Dr. Tsuneo Nakata, who were my leaders. I have greatly benefited from discussions with many engineers in Fujitsu. They gave me real issues and live comments for the research.

I consider it as my honor to have worked under Prof. Randal E. Bryant and near Prof. Edmund M. Clarke during my one-year visit to Carnegie Mellon University. Discussions with Professor Bryant have been illuminating. I was greatly encouraged by Professor Clarke's positive comments on our model checking method.

For recent work, discussions with the members and collaborators of JST ERATO Minato Discrete Structure Manipulation Project have been insightful. Prof. Takeaki Uno contributed substantially to the ideas about our fast counting algorithm of SAWs.

I would like to offer my special thanks to my parents and my brother, who supported my student days and always encouraged me. Finally, I thank my wife, Miki, and my daughter, Nako. They are always beside me and cheering me up.

Contents

Chapter 1

Introduction

Design verification is one of the oldest important problems in computer-aided design of digital systems (CAD). Its computational cost grows exponentially against the design complexity, as long as we stay using the same verification methods. It would be no exaggeration to say that the limit of verification dominates the limit of design. The industrial world has always been asking researchers to improve the verification technology.

Binary decision diagrams (BDDs) and *zero-suppressed BDDs* (ZDDs) are important data structures for representing Boolean functions and families of sets on computers [Bry86, Min93, Knu11]. They have originally become popular in LSI CAD problems, such as logic synthesis and verification. Their range of applications is still expanding beyond LSI since Boolean functions and families of sets are fundamental elements for manipulating discrete structures.

This study is focused on *property verification*, which is a problem of checking if a given property holds on a verification target. In the early 1990s which we began this study, basic technology of BDDs became matured and its application to real LSI verification was attracting a great concern. A BDD-based model checking technique, called *symbolic model checking* [BCM$^+$92, TBK95, HTKB93, McM93], brought a breakthrough in property verification. Many researchers were trying to put model checking tools fit for practical use. On the other hand, it was expected (and is still correct now) that entire logic of a real LSI chip cannot be verified only using model checking. Therefore, our research interests were both in simulation-based validation and model checking.

First of all, we tackled simulation-based validation problems [INH92, INH93, IKNH94a, IKNH94b]. One of the urgent requests from LSI designers was improvement of functional test quality, which had been supported only by human efforts. We introduced a problem formulation as input pattern generation on a finite state machine (FSM). One key to it is formulation of mapping from *test cases*, which have been vague human intent to check something, to sets of states on the FSM. Another key is scalability of the algorithm to real-world problems. We have developed a functional test generation tool for real microprocessor designs.

Secondly, we proposed *forward model checking*, which is intended to be useful in the field of industrial LSI verification [INH96, IN97, IN99]. In those days, most properties for symbolic model checking were written in *computation tree logic* (CTL); because it can be evaluated efficiently using BDD-based symbolic techniques as well as it has good expressive power of properties. Evaluation of a CTL operator implies backward state traversal; however, it is often observed in some situation that backward symbolic state traversal costs much more than forward one. We developed a method to avoid this inefficiency by converting CTL operators in a property into our new forward traversal operators. The most essential idea is an efficient algorithm to check existence of a state transition loop without using backward state traversal. It was accepted by model checking community as a unique technique and was imported into VIS [BHSV$^+$96], which was a state-of-the-art academic tool at that time. The method covers the many CTL properties that are actually used in the field of LSI verification. We also developed the forward model checking method that use ω-regular expression to describe properties. It is a natural representation for the property class covered by forward model checking, and was actually preferred by many engineers rather than CTL. Our algorithm is often much faster than conventional ones and is especially fast to find a counterexample.

In 2000s, LSI verification tools based on model checking and other formal methods became matured. People who adopted them began to notice that they were very useful but were just tools; they do not work very well without a large amount of know-how, called verification methodology. Many corporate researchers of LSI verification shifted their works from tools to methodologies. It just overlapped with "winter of BDD" [Min13], when many researchers moved from BDD-related works to other areas.

Now, the winter period have ended and BDD-related works are active again especially in non-LSI applications such as data mining and geographic information systems, because memory size of a computer has become large enough to handle such real-life problems. We have developed a new ZDD manipulation framework optimized for huge ZDDs on a modern computer. There are many open source and in-house BDD/ZDD packages which have been used in such traditional applications as CAD problems. They are general-purpose packages for manipulating a collection of BDDs/ZDDs [BRB90, MIY90], allowing us to create primitive BDDs/ZDDs (variables and constants) and to construct complex BDDs/ZDDs by applying operations repeatedly to existing ones. They usually traverse given BDDs/ZDDs in a depth-first manner and construct the resulting BDD/ZDD in a bottom-up way. Our framework constructs a ZDD from top to bottom in a breadth-first manner, which have been preferred in the situations where memory access locality is a serious matter [OYY93, AC94, SRBSV96, CYB97]. Unlike the traditional BDD/ZDD libraries, it constructs a complex ZDD directly from the root to the terminal nodes based on *frontier-based methods* [Min13]. We have also demonstrated the power of frontier-based methods by computing the number of self-avoiding walks connecting opposite corners of a $n \times n$ square lattice up to $n = 26$ [IKM12, INK$^+$13a, INK$^+$13b], which is the current world record registered in the On-Line Encyclopedia of Integer Sequences [OEIa].

The rest of the thesis is organized as follows. Chapter 2 describes the backgrounds that underlie this work. Chapter 3 deals with simulation-based validation problems of functional LSI designs, in which processor pipelining is taken up as a typical problem. Chapter 4 shows technical details of forward CTL model checking and ω-regular language emptiness check on explicit property graph. In the first half of Chapter 5, the new ZDD manipulation framework for applications beyond LSI designs is developed. In the second half, the case study for self-avoiding walk problems demonstrates the possibilities of this work. Finally, the thesis is concluded in Chapter 6.

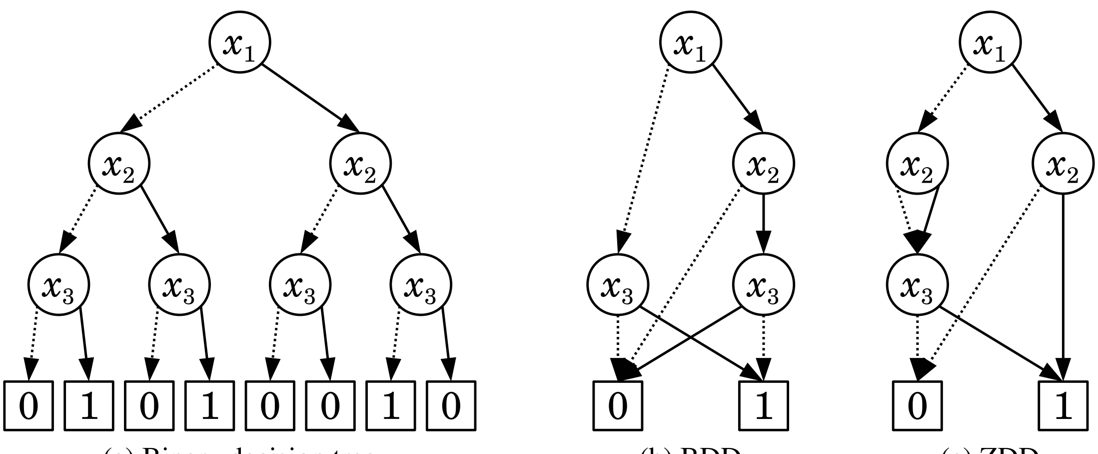
CHAPTER 2

Preliminaries

2.1 Binary decision diagrams

2.1.1 BDDs and ZDDs

Binary decision diagrams (*BDDs*) [Ake78, Bry86] and *zero-suppressed BDDs* (*ZDDs*) [Min93] are labeled directed acyclic graphs derived by reducing binary decision tree graphs, which represent decision making processes through binary input variables. As illustrated in Figure 2.1, there are two kinds of terminal nodes, *0-terminal* and *1-terminal*, which represent the output binary value. Every nonterminal node is labeled by an input variable and has two outgoing edges, namely *0-edge* and *1-edge*, which are drawn as dotted and solid arrows respectively. The 0-edge (1-edge) points to the node called *0-child*



(a) Binary decision tree (b) BDD (c) ZDD

Figure 2.1: Diagrams for $f(x_1, x_2, x_3) = x_1 x_2 \overline{x_3} + \overline{x_1} x_3$

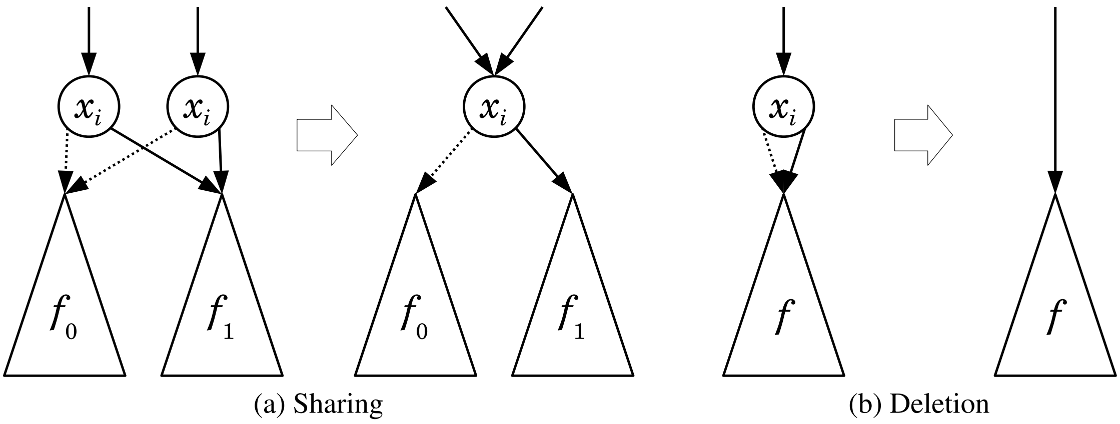


Figure 2.2: BDD reduction rules

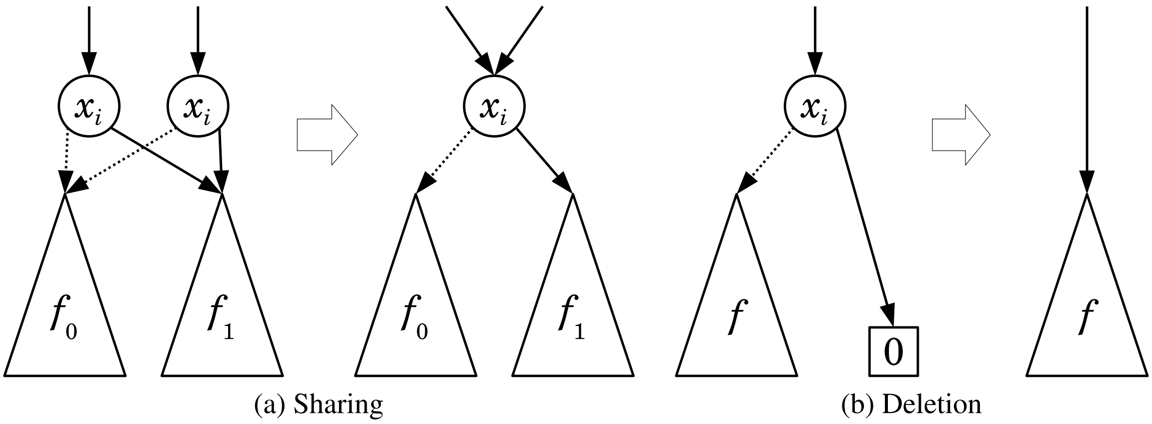


Figure 2.3: ZDD reduction rules

(*1-child*), which represent a state after the decision that 0 (1) is assigned to the variable. When the root node of a BDD/ZDD for Boolean function f is labeled by variable x, its 0-child and 1-child represent $f_{x=0}$ and $f_{x=1}$ respectively; it corresponds to the Shannon expansion: $f = \overline{x}\,f_{x=0} + x\,f_{x=1}$. We also write it as $f = (\overline{x} \mathbin{?} f_{x=0} : f_{x=1})$, implying structure of the diagram.

We only deal with ordered BDDs/ZDDs in this thesis, where input variables are indexed as $x_1, \ldots, x_n$ according to their total order. The index of the input variable of a nonterminal node is just called the index of the node, and the index of a terminal node is assumed to be $n+1$ for convenience. The index of any node is properly smaller than that of its children.

Figure 2.2 and Figure 2.3 show the reduction rules of BDDs and ZDDs respectively. Equivalent nodes, which have the same indices and the same 0- and 1-child nodes, can be shared both in BDDs and in ZDDs (Figure 2.2a and Figure 2.3a). A node with edges to the same destination can be deleted in BDDs (Figure 2.2b). In contrast, ZDDs have a different rule, called *zero-suppress rule*, by which a node with a 1-edge directly pointing to the 0-terminal is deleted (Figure 2.3b). If x has a smaller index than the top variable of f, $f_{x=0} = f_{x=1} = f$ in BDDs while $f_{x=0} = f$ and $f_{x=1} = 0$ in ZDDs. An entire BDD/ZDD can be reduced completely by applying the reduction rules from the bottom (index n) to the top (index 1) as follows:

```
REDUCE(f)
for i = n to 1 do
    for all node p at index i in the diagram rooted by f do
        for all b ∈ {0,1} do
            apply reduction rules to the b-child of p;
        end for
    end for
end for
return reduced node for f.
```

BDDs and ZDDs are efficient data structures for representing not only Boolean functions but also families of sets. A set of n items can be represented by input variables $x_1, \ldots, x_n$, where $x_i \in \{0,1\}$ indicates if the i-th item is contained in the set. The diagrams in Figure 2.1 can be considered as $\{\{x_1,x_2\},\{x_2,x_3\},\{x_3\}\}$ in that sense. Paths from the root to the 1-terminal in BDDs and ZDDs, called *1-paths*, correspond to item sets included in the family. ZDDs have the interesting property that every 1-path represents an individual set, while a 1-path may represent multiple sets in BDDs, because of the difference of their node deletion rules. ZDDs are especially suitable for representing families of sparse item sets. If the average appearance rate of each item is 1%, ZDDs are possibly up to 100 times more compact than BDDs. Such situations often appear in real-life problems.

2.1.2 Operations on BDDs/ZDDs

We can build up complex BDDs/ZDDs for various functions and sets by combinations of their rich algebraic operations such as Boolean operations and family algebra [Knu11]. They use divide-and-conquer scheme based on the Shannon expansion, which is accelerated by the memo cache that avoids recomputation of the same subproblems. The following algorithm outlines a typical depth-first implementation of binary operations:

DF_BINARYOPERATION$(\diamond, f, g)$
if $f \diamond g$ has a terminal value, **return** it;
if $f \diamond g = h$ is in the memo cache, **return** h;
$x \leftarrow$ the top variable of f and g;
$h_0 \leftarrow$ DF_BINARYOPERATION$(op, f|_{x=0}, g|_{x=0})$;
$h_1 \leftarrow$ DF_BINARYOPERATION$(op, f|_{x=1}, g|_{x=1})$;
$h \leftarrow (\bar{x}\ ?\ h_0 : h_1)$;
apply reduction rules to h;
put $f \diamond g = h$ into the memo cache;
return h.

This algorithm constructs a reduced diagram recursively from the bottom to the top.

Binary operations on BDDs/ZDDs can be implemented also in a breadth-first manner. We create a new node immediately after dividing the problem; the new node is incomplete as its descendants are not determined yet. The algorithm is outlined as follows:

BF_BINARYOPERATION$(\diamond, f, g)$
let i_0 be the top index of f and g;
create a new node h and label it as $\langle i_0, f, g\rangle$;
for $i = i_0$ to n **do**
  for all node r labeled $\langle i, p, q\rangle$ **do**
    for all $b \in \{0, 1\}$ **do**
      $p' \leftarrow p|_{x_i=b}$; $q' \leftarrow q|_{x_i=b}$;
      if $p' \diamond q'$ has a terminal value **then**
        set it to the b-child of r;
      else
        $i' \leftarrow$ the top index of p' and q';
        find or create node r' labeled $\langle i', p', q'\rangle$;
        set r' to the b-child of r;

```
            end if
        end for
    end for
end for
return REDUCE(h).
```

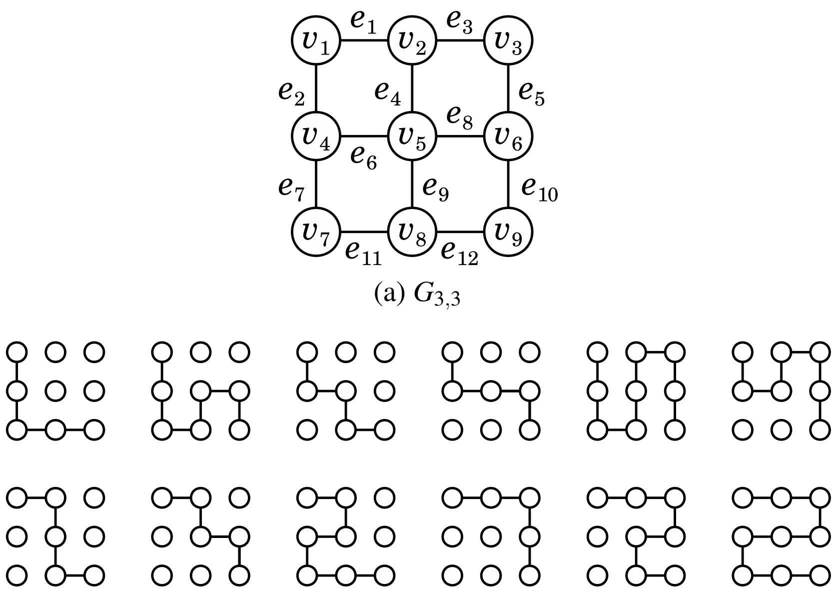
This algorithm constructs a diagram from the top to the bottom. Incomplete nodes are labeled by its index and two operands of the subproblems, in order to share nodes for the same subproblems. The operand information of each node can be removed when its child nodes are fixed. Since the top-down phase does not fully reduce the diagram, the reduction algorithm is applied as a post-process.

2.1.3 Top-Down Construction

Single-pass BDD/ZDD construction from the root to the terminals, which we call *top-down construction*, is another way to build a complex BDD/ZDD structure. It is known that some important graph problems can be solved efficiently using such methods [SIT95, SI97, Knu11, Min13].

Node sharing must be performed on the fly during top-down construction in order to avoid explosion of the diagram. Multiple nonterminal nodes with the same index can be shared if and only if they take the same output values for all combinations of the rest of input values. Since this condition is not always easy to be determined on the fly, it is checked in a false-negative way by comparing the labels generated at some reasonable cost. They are so designed that multiple nonterminal nodes are equivalent if their labels are equivalent; the converse is not necessarily true because unshared nodes can be left for the final reduction phase.

Knuth introduced an interesting algorithm in his book, named SIMPATH, which constructs a ZDD representing a set of paths (ways to go from a point to another point without visiting any point twice) in an undirected graph [Knu11, Knu]. For example, a 3×3 grid graph ($G_{3,3}$) in Figure 2.4a has 12 paths between v_1 and v_9 as shown in Figure 2.4b. The input to the algorithm is an undirected graph $G = (V, E)$ where $V = \{v_1, \ldots, v_m\}$ is a set of vertices



(a) $G_{3,3}$

(b) Paths between v_1 and v_9

Figure 2.4: Path enumeration on $G_{3,3}$

and $E = \{e_1, \ldots, e_n\}$ is a set of edges. The output is a ZDD representing all the set of edges that form paths between v_1 and v_m.

In the SIMPATH algorithm, edge selections from $E = \{e_1, \ldots, e_n\}$ are decided one by one in the order of indices. At each step of the algorithm, a set of selected edges represents path fragments and each vertex has one of the three states:

- not included in any path fragment,
- an endpoint of a path fragment,
- an intermediate point of a path fragment.

The label for a nonterminal node is defined to be $\langle i, mate \rangle$ where $1 \leq i \leq n$ and *mate* is a partial map from V to $V \cup \{0\}$:

$$mate[v] = \begin{cases} v & \text{if vertex } v \text{ is untouched so far,} \\ u & \text{if vertices } u \text{ and } v \text{ are endpoints,} \\ 0 & \text{if vertex } v \text{ is an intermediate point.} \end{cases}$$

For simplicity of the algorithm, *mate* is maintained as if there were a built-in path between v_1 and v_m, and we were enumerating all the virtual cycles that include it. The current set of selected edges is accepted when:

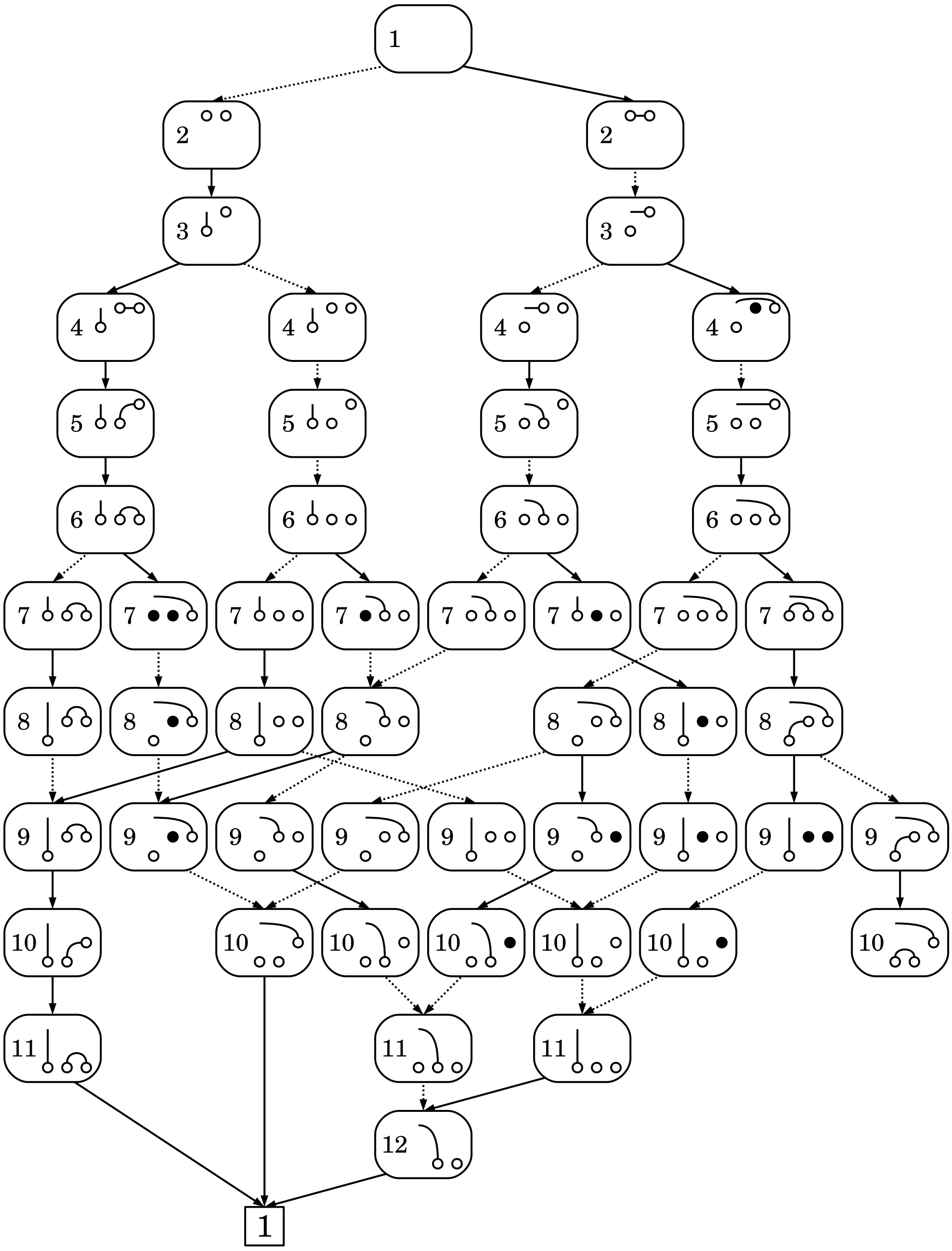


Figure 2.5: ZDD structure constructed by SIMPATH

- a virtual cycle is formed and no other path fragment remains,

and it is rejected when:

- a virtual cycle is formed and some other path fragment remains, or
- an edge to an intermediate point is added, or
- the final chance to attach an edge to some endpoint is not taken.

In order to check the above conditions, we need *mate* entries only for *frontier*, which is a set of vertices contiguous with both decided and undecided edges. When vertex v is entering the frontier, a table entry $mate[v] = v$ is created except for $mate[v_1] = v_m$ and $mate[v_m] = v_1$. The entry for $mate[v]$ is deleted after v has left the frontier.

Figure 2.5 illustrates the result of SIMPATH for $G_{3,3}$, where the 0-terminal node is omitted and *mate* is drawn graphically on each node. Circles and lines represent vertices in the frontier and path fragments among them respectively. An isolated open circle represents a vertex not included in any path fragments ($mate[v] = v$). An isolated filled circle represents an intermediate point of some path fragment ($mate[v] = 0$). Note that the ZDD node deletion rule is only used at edges to the 1-terminal node.

2.2 Symbolic model checking

2.2.1 Finite state machines

Finite state machines (*FSMs*) are widely used for modeling finite state systems. An FSM is a 6-tuple, $(S, I, O, \delta, \lambda, s_0)$, where S is the set of states, I is the set of input values, O is the set of output values, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function, and $s_0 \in S$ is the initial state. In what follows, let $B = \{0, 1\}$. We assume that functions δ and λ are completely specified and the FSM is deterministic. Note that a non-deterministic FSM is easily converted to an equivalent deterministic FSM by adding unconstrained pseudo inputs.

The *transition relation* of an FSM is the function $T : S \times I \times S \to B$; $T(x,i,y) = 1$ if and only if $y = \delta(x,i)$. In the basic symbolic technique, a single BDD is built for representing the transition relation T. For a large FSM, however, we often fail to construct the BDD for T because of a BDD size explosion. *Partitioned transition relations* [BCL91] are popular representation to reduce the BDD size. When the state is expressed by a vector of n Boolean state variables (latches) and the transition function of the k-th latch is given by $\delta_k(\vec{x},\vec{i})$, we can make a conjunctive partitioned transition relation as follows:

$$\begin{aligned} T(\vec{x},\vec{i},\vec{y}) &= T_1(\vec{x},\vec{i},y_1) \wedge \ldots \wedge T_n(\vec{x},\vec{i},y_n)\,, \\ T_k(\vec{x},\vec{i},y_k) &= \Big(y_k \equiv \delta_k(\vec{x},\vec{i})\Big)\,. \end{aligned}$$

Latch transition relations $T_1,\ldots,T_n$ are represented by n BDDs, which are much smaller than the BDD for T in general.

2.2.2 Images and pre-images

Given an FSM $(S,I,O,\delta,\lambda,s_0)$ and a set of states $A \subseteq S$, the *image* of A is defined to be the set of states $\{y \mid \exists x \in A, \exists i \in I, y = \delta(x,i)\}$, and the *pre-image* of A is defined by the set of states $\{x \mid \exists y \in A, \exists i \in I, y = \delta(x,i)\}$.

A set of states $A \subseteq S$ can be represented by a *characteristic function* $\chi_A : S \to B$; $\chi_A(x) = 1$ if and only if $x \in A$. Let $\mathscr{L}(f)$ express the set of states represented by function f. The image and the pre-image of $\mathscr{L}(f)$ are calculated by following symbolic operations:

$$\begin{aligned} &\mathit{Img}(f)(\vec{y}) \\ &\quad= \exists\vec{x}.\exists\vec{i}.\Big[T(\vec{x},\vec{i},\vec{y}) \wedge f(\vec{x})\Big] \\ &\quad= \exists\vec{x}.\exists\vec{i}.\Big[T_1(\vec{x},\vec{i},y_1) \wedge \ldots \wedge T_n(\vec{x},\vec{i},y_n) \wedge f(\vec{x})\Big]\,, \\ &\mathit{Pre}(f)(\vec{x}) \\ &\quad= \exists\vec{i}.\exists\vec{y}.\Big[T(\vec{x},\vec{i},\vec{y}) \wedge f(\vec{y})\Big] \\ &\quad= \exists\vec{i}.\exists\vec{y}.\Big[T_1(\vec{x},\vec{i},y_1) \wedge \ldots \wedge T_n(\vec{x},\vec{i},y_n) \wedge f(\vec{y})\Big]\,. \end{aligned}$$

The image and the pre-image of $\mathscr{L}(f)$ are $\mathscr{L}(\mathit{Img}(f))$ and $\mathscr{L}(\mathit{Pre}(f))$ respectively. Operations $\mathit{Img}(f)$ and $\mathit{Pre}(f)$ are similar if T is given by a single

BDD. When we use conjunctive partitioned transition relations, early existential quantification can be done while calculating the conjunction of all BDDs [TSL+90, BCL91]. Efficiency of the calculation strongly depends on the order in which the BDDs are processed. Strategies to find effective orders for $Img(f)$ and $Pre(f)$ seem to be different, because those expressions have different forms.

2.2.3 State enumeration

State enumeration is the process to compute the set of reachable states from the initial state of an FSM. It is a core procedure for various verification problems: comparing two FSMs, checking reachability to some 'bad' state, and reducing state traversal space in model checking. Given an FSM $(S, I, O, \delta, \lambda, s_0)$, a state enumeration procedure is the least fix-point computation:

$$Reached \;=\; \mathbf{lfp}\, Z\, [s_0 \vee Img(Z)]\,.$$

2.2.4 CTL model checking

Model checking is the process of determining whether a model (FSM) satisfies its requirements (properties). A temporal logic *CTL* [CES86] is commonly used to express properties about an FSM. CTL formulas are composed of atomic propositions with usual logical operators and following temporal operators:

- $\mathbf{EX}\, f$ $(\mathbf{AX}\, f)$ which means that f holds at some (every) successor state of the current state.
- $\mathbf{EF}\, f$ $(\mathbf{AF}\, f)$ which means that for some (every) state transition path, there exists a state on the path at which f holds.
- $\mathbf{EG}\, f$ $(\mathbf{AG}\, f)$ which means that for some (every) state transition path, f keeps holding forever on the path.
- $\mathbf{E}\,[g\,\mathbf{U}\,f]$ $(\mathbf{A}\,[g\,\mathbf{U}\,f])$ which means that for some (every) state transition path, there exists a state on the path at which f holds, and g holds at all the preceding states.

A CTL property is expressed by a notation like "$M, s \models f$." It means that the CTL formula f is true in state s of model M. It is also written simply as "$s \models f$" where the model is not ambiguous.

CTL formula f can be interpreted as a set of states $\mathscr{L}(f) = \{ s \mid s \models f \}$. $\mathbf{EX}\, f$ is then the same operation as computing the pre-image of $\mathscr{L}(f)$:

$$\mathbf{EX}\, f \;=\; \mathit{Pre}(f)\,.$$

$\mathbf{E}\,[g\,\mathbf{U}\,f]$ and $\mathbf{EG}\, f$ can be characterized by the least and greatest fix-point computation as follows:

$$\begin{aligned} \mathbf{E}\,[g\,\mathbf{U}\,f] &= \mathbf{lfp}\, Z\, [f \vee (g \wedge \mathbf{EX}\, Z)]\,, \\ \mathbf{EG}\, f &= \mathbf{gfp}\, Z\, [f \wedge \mathbf{EX}\, Z]\,. \end{aligned}$$

The remaining operators are given by following rules:

$$\begin{aligned} \mathbf{EF}\, f &= \mathbf{E}\,[\mathrm{true}\,\mathbf{U}\,f]\,, \\ \mathbf{AX}\, f &= \neg\,\mathbf{EX}\,\neg f\,, \\ \mathbf{AF}\, f &= \neg\,\mathbf{EG}\,\neg f\,, \\ \mathbf{AG}\, f &= \neg\,\mathbf{EF}\,\neg f\,, \\ \mathbf{A}\,[g\,\mathbf{U}\,f] &= \neg\,(\mathbf{E}\,[\neg f\,\mathbf{U}\,\neg g \wedge \neg f] \vee \mathbf{EG}\,\neg f)\,. \end{aligned}$$

A *fairness constraint* is a condition representing fair state transition paths in which we are interested. CTL model checking under fairness constraints is performed by restricting state transition paths along which each fairness constraint holds infinitely often. Fairness constraints are given by a set of CTL formulas C. CTL formula $\mathbf{EG}\, f$ under fairness constraints in C is computed as follows [McM93]:

$$\mathbf{E}_C\mathbf{G}\, f \;=\; \mathbf{gfp}\, Z \left[f \wedge \mathbf{EX} \bigwedge_{c \in C} \mathbf{E}\,[Z\,\mathbf{U}\,Z \wedge c] \right].$$

The set of states that are the start of some fair path under fairness constraints in C is given by $\mathscr{L}(\mathbf{E}_C\mathbf{G}\,\mathrm{true})$. Once $\mathbf{E}_C\mathbf{G}\,\mathrm{true}$ is evaluated, $\mathbf{EX}\, f$ and $\mathbf{E}\,[g\,\mathbf{U}\,f]$ under fairness constraints in C can be computed simply as follows:

$$\begin{aligned} \mathbf{E}_C\mathbf{X}\, f &= \mathbf{EX}\,(f \wedge \mathbf{E}_C\mathbf{G}\,\mathrm{true})\,, \\ \mathbf{E}_C\,[g\,\mathbf{U}\,f] &= \mathbf{E}\,[g\,\mathbf{U}\,(f \wedge \mathbf{E}_C\mathbf{G}\,\mathrm{true})]\,. \end{aligned}$$

2.2.5 $*$-regular and ω-regular expressions

In what follows, Σ stands for an *alphabet*, which is a finite set, and ω stands for an infinite number ($k < \omega$ for all finite number k). Σ^* is the set of all finite sequences over Σ and Σ^ω is the set of all infinite sequences over Σ. We write $\emptyset$ for an empty set and ε for a set with the unique element that is a null sequence.

Let $X, Y \in \Sigma^*$ and $U, V \subseteq \Sigma^*$. XY is the concatenation of X and Y, i.e., $XY = x_0 \cdots x_{m-1} y_0 \cdots y_{n-1}$ where $X = x_0 \cdots x_{m-1}$ ($x_k \in \Sigma$) and $Y = y_0 \cdots y_{n-1}$ ($y_k \in \Sigma$). The concatenation of U and V is defined as follows:

$$UV = \{XY \mid X \in U,\, Y \in V\}.$$

The *star operation* on $V \subseteq \Sigma^*$, denoted by V^*, is a set of all sequences $X \in \Sigma^*$ which are composed of a finite concatenation of arbitrary sequences in V:

$$\begin{aligned} V^0 &= \varepsilon, \\ V^k &= V^{k-1}V \quad (k \geq 1), \\ V^* &= \bigcup_{0 \leq k < \omega} V^k. \end{aligned}$$

V^ω is defined to be a set of all sequences $X \in \Sigma^\omega$ which are composed of an infinite concatenation of non-null sequences in V. Note that a null sequence in V has no effect on V^ω, e.g., $(V \cup \varepsilon)^\omega = (V - \varepsilon)^\omega$. In this thesis, we consider only about the cases when V does not include a null sequence ($V \cap \varepsilon = \emptyset$), without loss of the expressive power.

The syntax of $*$-*regular expressions* and ω-*regular expressions* are defined inductively as follows:

- ε is a $*$-regular expression.
- If $P \subseteq \Sigma$, then P is a $*$-regular expression.
- If U and V are $*$-regular expressions, then so are $U \cup V$, UV, and V^*.
- If U and V are $*$-regular expressions and $V \cap \varepsilon = \emptyset$, then UV^ω is an ω-regular expression.
- If U and V are ω-regular expressions, then so is $U \cup V$.

The sets of sequences that can be obtained from $*$-regular expressions and ω-regular expressions are called *$*$-regular sets* and *ω-regular sets* respectively.

2.2.6 ω-automata

An ω-automaton is a 5-tuple (Σ, Q, q_0, T, C), where Σ is the alphabet, Q is the set of states, $q_0 \in Q$ is the initial state, $T \subseteq Q \times \Sigma \times Q$ is the transition relation, and C is the acceptance condition of infinite sequences. The set of infinite sequences accepted by ω-automaton A is called the *language* of A, which is denoted by $\mathscr{L}(A)$. The language of an ω-automaton is an ω-regular set.

2.2.7 Language containment check and language emptiness check

Suppose that M is a design model represented as an ω-automaton and A is a property also represented as an ω-automaton, where M and A have the same alphabet. Language containment is a problem of checking $\mathscr{L}(M) \subseteq \mathscr{L}(A)$, which means that every sequence on M satisfies an acceptance condition of A. A conventional way of solving this problem is to check $\mathscr{L}\left(M \times \overline{A}\right) = \emptyset$. $\overline{A}$ is the complement automaton of A, which is the automaton that accepts every infinite sequence that is not accepted by A. $M \times \overline{A}$ is the product machine of M and $\overline{A}$. After this conversion, the problem is often called language emptiness check. It can be also written as $\mathscr{L}(M) \cap \mathscr{L}\left(\overline{A}\right) = \emptyset$.

2.3 Pipelined processors

Pipelined processors have some steps which can usually be executed in one clock cycle. Each of these steps is called a *pipe stage*. During each clock cycle, the hardware executes some parts of different instructions simultaneously. The DLX processor [HP90] has five stages in its pipeline—IF, ID, EX, MEM, and WB—and can potentially execute five overlapped instructions in each clock cycle (Figure 2.6). In the following sections, DLX is used as an example pipelining.

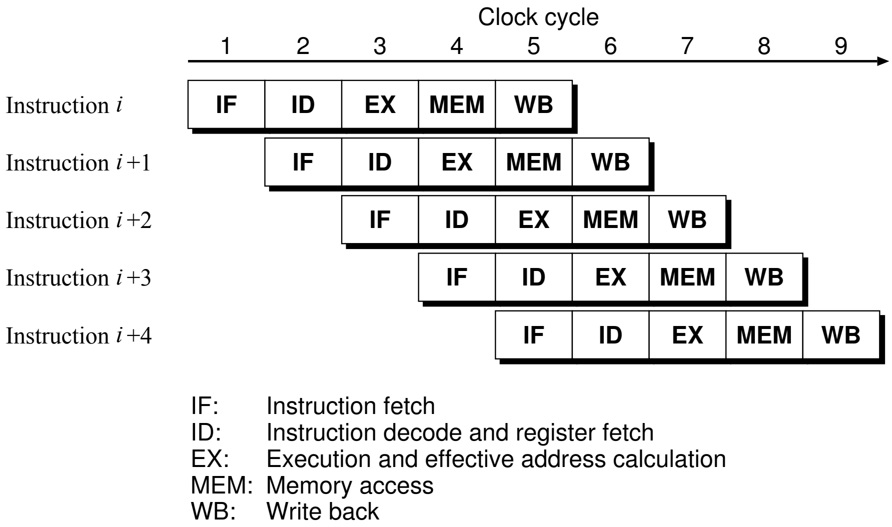


Figure 2.6: Basic DLX pipeline

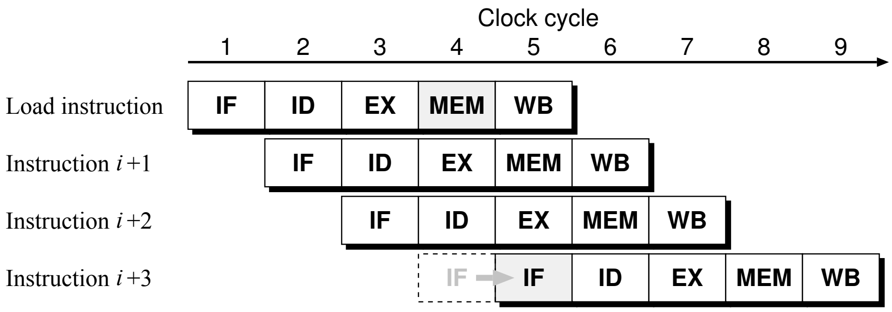


Figure 2.7: Structural hazard

Situations that prevent the next instruction from being executed during its designated clock cycle are called *hazards*. If a hazard occurs, one or more pipe stages stop execution for some clock cycles. These are called *stalls*. Hazards can be classified into three types: *structural hazards*, *data hazards*, and *control hazards* [HP90].

2.3.1 Structural Hazards

Resource conflicts prevent two instructions at different stages from being executed simultaneously. Figure 2.7 shows an example of a structural hazard on a pipelined processor with only one memory port. An instruction fetch cannot be initiated at the IF stage in the same cycle as a data fetch at the MEM stage. In this case, no instruction is initiated when a load instruction is executed in the MEM stage. It is assumed that instructions $i+1$, $i+2$, and

$i+3$ do not have access to memory data and do not cause any other hazards.

2.3.2 Data Hazards

The read or write sequence for the same storage location may be changed with the overlapping execution of instructions. Three types of hazards arise in this situation:

RAW (*read after write*) An instruction attempts to read a result of the previous instruction before it is written.

WAR (*write after read*) An instruction attempts to write a new value when the previous instruction needs the old value and has not yet read it.

WAW (*write after write*) An instruction attempts to write a new value before the previous instruction has written one to the same location.

Figure 2.8 shows examples of RAW hazards between LW (load word) and ADD instructions. It is assumed that hazards do not occur between instruction fetches and data fetches. R0 to R3 denote registers. The LW instruction loads contents of memory location $30+\text{R0}$ at the MEM stage and then writes them into R1 at the WB stage. The ADD instruction reads R1 and R2 at the ID stage, executes the arithmetic operation at the EX stage, and writes the result into R3 at the WB stage. Since the ADD instruction needs the result of the LW instruction through R1, the ID stage of the ADD must stall until the WB stage of the LW is over.

2.3.3 Control Hazards

An instruction that changes the program counter (PC) to something other than the next instruction address, e.g., a jump or branch instruction, may also causes hazards. Control hazards can be treated as special data hazards (RAW hazards on the PC) in some simple pipeline models. Since control hazards degrade performance more than data hazards for most RISC processors, several methods that reduce pipeline branch penalties are widely used.

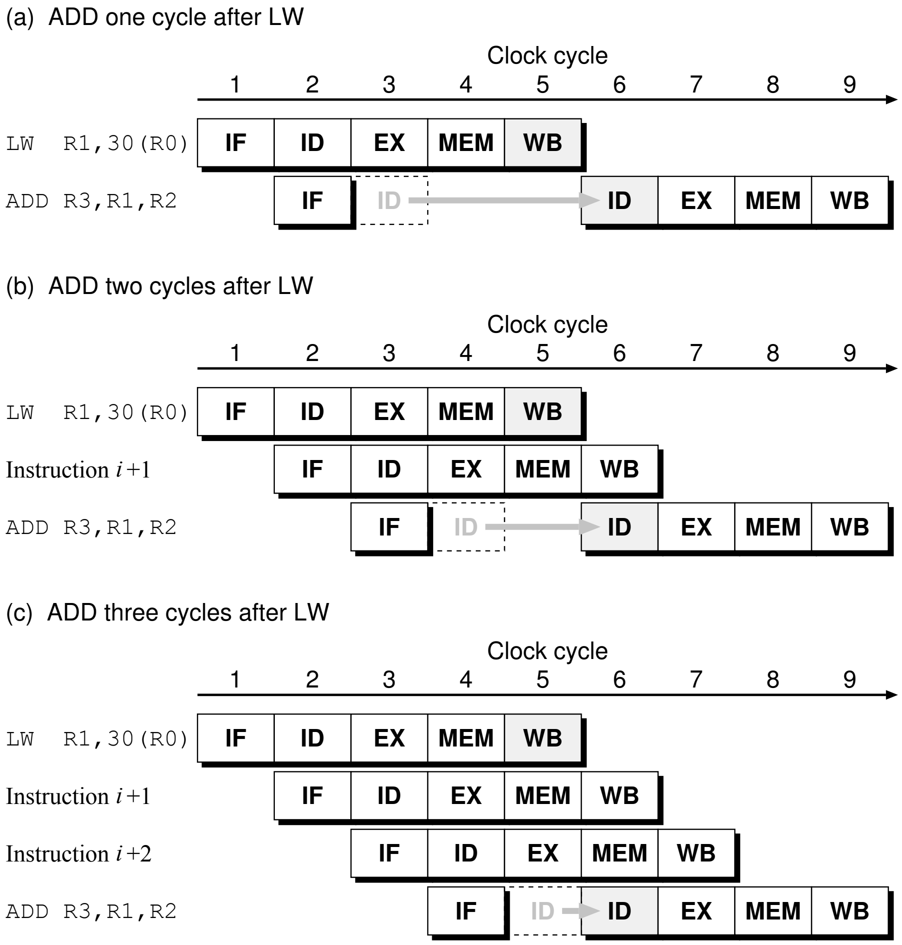


Figure 2.8: Data hazard

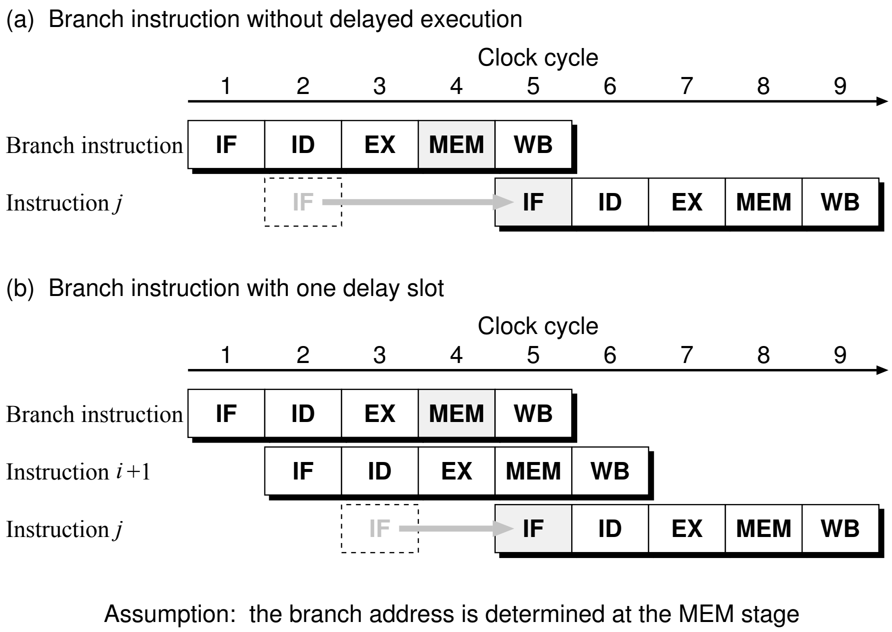


Figure 2.9: Branch instruction with/without delayed execution

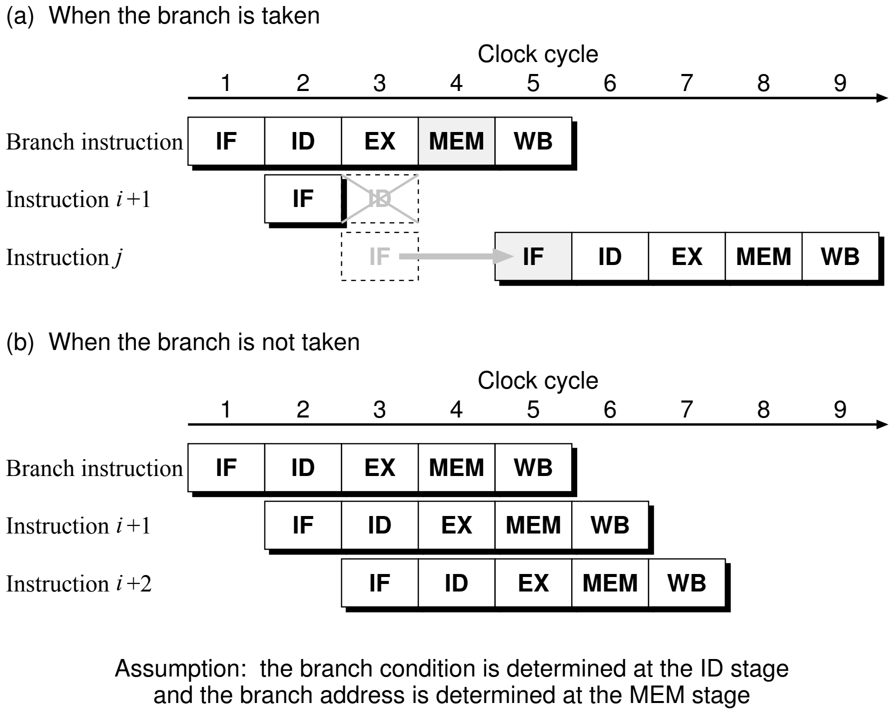


Figure 2.10: Predict-not-taken scheme when the branch is taken or is not taken

Delayed-branch is a technique allowing hardware to execute a fixed number of instructions after a branch but before the actual branch operation is done. Delayed-branch influences pipeline interlocks as some hazardous instruction sequences may no longer cause hazards. Figure 2.9 (a) shows normal execution patterns for a branch instruction, and (b) for a delayed-branch.

Delayed-branch instructions in the SPARC-V9[SPA92] instruction set have *annul* (a) bits, which are used to determine whether their delay instructions are executed or flushed. For example, the delay instruction of a conditional branch instruction with $a = 1$ is *annulled* (not executed) if the branch is taken.

Branch-prediction is a technique allowing hardware to continue execution as if the branch were taken or not taken according to the prediction. When the branch outcome is definitely known and it is not the same as predicted one, we need to flush the pipeline and restart the instruction fetch. In the simple branch-prediction scheme called *predict-not-taken* (Figure 2.10), the branch is always predicted as not taken.

We treat the branch-prediction schemes in RISC processors as a delayed instruction execution combined with an annulling operation. In Figure 2.10, the instruction that follows the branch instruction is fetched, without stalling at cycle 2, as if it were the delay instruction (Figure 2.9(b)), and is annulled at cycle 3 if the branch is taken.

CHAPTER 3

Input Pattern Enumeration for Comprehensive Functional Test

High-speed scalar and superscalar microprocessors use highly sophisticated pipelining [HP90, Joh91]. Pipeline complexity increases the number of design errors, and also makes design verification more difficult. Although formal verification of microprocessors is receiving affection from academic research [BD94b, BB94, BD94a], none of the methods proposed can handle entire designs of today's complex pipelined processors. Simulation-based verification is therefore still indispensable.

Simulation-based verification applies instruction sequences to a logic simulator for a processor design and a reference machine, such as an instruction-level simulator, and compares the results. Instruction sequences for simulation-based verification are called *test programs*. Currently, most test programs are coded by hand or generated randomly. Since processors are becoming more complex, it is harder to write test programs manually. Also, the number of simulation cycles for random instructions must be increased to maintain verification reliability. A systematic way to generate an effective test program is needed.

Some papers have presented test program generation methods for pipelined processor verification [LS91, INH92, INH93]. These methods focus on pipeline hazards [HP90] and can generate effective test programs for target cases automatically. Pipeline behavior when and after a hazard is detected is not considered, so these methods cannot cover cases that are reachable only after hazards. Moreover, they cannot avoid unexpected hazards that prevent satisfying target cases. We present a new approach to generate test programs for any processor state that considers detailed pipeline movements.

3.1 Test programs for pipelined processors

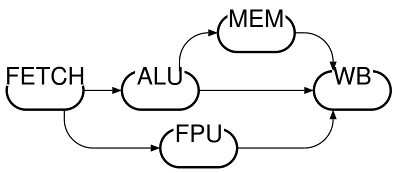
It has not been possible to measure quality of test programs because test program goals have not been formulated. We started by modeling processor pipelines and defined a test program generation goal. This model demonstrates the difficulty in hand-coding test programs. Also, efficiency of simulating random instructions can be measured.

3.1.1 Processor pipeline model

In order to concentrate on pipeline control parts of processors, we only care instruction flow in the processor pipeline, and do not care how operand data and result data are processed in functional modules. Thus, we simplify processor hardware to a set of *pipeline units*. A pipeline unit corresponds to a hardware block which can hold an instruction for one clock cycle. A hardware block that produces results in one cycle, such as an integer ALU, is modeled as a pipeline unit, and a hardware block that produces results in n clock cycles, such as a FPU, is divided into n sub-blocks and modeled as a series of n pipeline units.

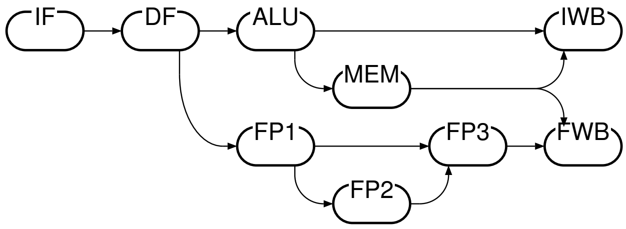
A pipeline unit that can hold k kinds of instructions has $k+1$ states; k states when it holds an instruction, plus one state when it holds nothing. A *pipeline state* of the processor is defined by the states of all pipeline units. If the i-th unit can hold k_i kinds of instructions, the processor has $\prod(k_i+1)$ pipeline states.

Figure 3.1 shows an example of a simple processor pipeline model P1. Pipeline units are FETCH, ALU, FPU, MEM, and WB. P1 has four types of instructions: `NOP`, `INT`, `LD`, and `FP`. FETCH reads a new instruction every clock cycle. `NOP` disappears immediately. `INT` moves to the ALU at the second clock cycle, and then moves to the WB at the third clock cycle. Similarly, `LD` moves to ALU, MEM, and then WB. `FP` moves to FPU and then



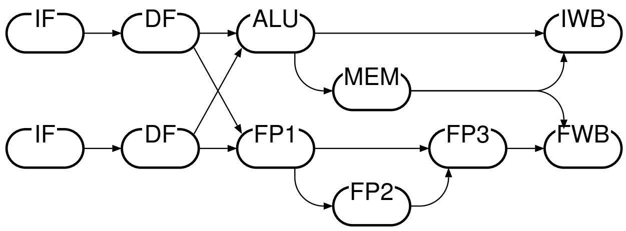
NOP : FETCH
INT : FETCH → ALU → WB
LD : FETCH → ALU → MEM → WB
FP : FETCH → FPU → WB

Figure 3.1: Processor P1



NOP : IF → DF
INT : IF → DF → ALU → IWB
LD : IF → DF → ALU → MEM → IWB
FLD : IF → DF → ALU → MEM → FWB
FADD : IF → DF → FP1 → FP3 → FWB
FMUL : IF → DF → FP1 → FP2 → FP3 → FWB

Figure 3.2: Processor P2



NOP : IF → DF
INT : IF → DF → ALU → IWB
LD : IF → DF → ALU → MEM → IWB
FLD : IF → DF → ALU → MEM → FWB
FADD : IF → DF → FP1 → FP3 → FWB
FMUL : IF → DF → FP1 → FP2 → FP3 → FWB

Figure 3.3: Processor P3

to WB. FETCH can hold four kinds of instructions, ALU can hold two kinds, FPU and MEM can each hold one kind, and WB can hold three kinds. Thus, P1 has $5 \times 3 \times 2 \times 2 \times 4 = 240$ pipeline states. Two more processor pipeline model examples are shown in Figure 3.2 and Figure 3.3. P2 has an FPU with a result latency of 2 (for `FADD`) or 3 (for `FMUL`), which is modeled as three pipeline units — FP1, FP2, and FP3. P3 is a superscalar version of P2, which issues two instructions per clock cycle.

3.1.2 Test case

A *test case* is defined by a set of pipeline states that activate the same class of mechanism. Suppose we want to test all cases that cause structural hazards on pipeline units, and all combinations of two or more of those cases that happen

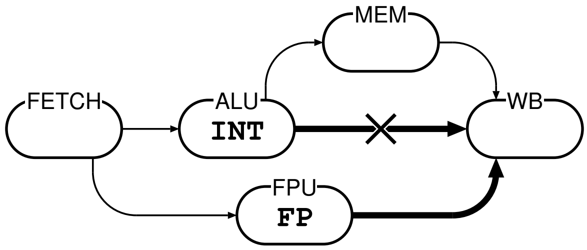


Figure 3.4: Structural hazard on WB

simultaneously. A pipeline control mechanism is activated by instructions in some pipeline units, and we do not have to care about instructions in the rest of pipeline units. A test case can be written as a set of (unit, instruction) pairs.

Figure 3.4 shows a pipeline hazard in P1 when `INT` and `FP` instructions attempt to move to WB simultaneously. This case is written as {(ALU, `INT`), (FPU, `FP`)}, which includes $5 \times 1 \times 1 \times 2 \times 4 = 40$ pipeline states. When a structural hazard occurs between two instructions, the instruction with higher priority is sent to the next pipeline unit and the other is stalled. If we assume a static priority of `LD` > `FP` > `INT` in this example, `INT` in ALU stalls. It causes another hazard on ALU if FETCH has `INT` or `LD`. We count them as individual cases {(FETCH, `INT`), (ALU, `INT`), (FPU, `FP`)} and {(FETCH, `LD`), (ALU, `INT`), (FPU, `FP`)} because different mechanisms may be activated for each case. A combination of test cases {(ALU, `INT`), (FPU, `FP`)} and {(ALU, `INT`), (MEM, `LD`)} also form another test case {(ALU, `INT`), (FPU, `FP`), (MEM, `LD`)}.

The numbers of test cases for processors P1, P2, and P3 (Figures 3.1, 3.2, 3.3) are shown in Table 3.1. These numbers will include many unreachable test cases. However, all cases must be considered to make a complete test program because it is not known whether they are reachable or not before analyzing all cases. The number of test cases grows rapidly as the processor pipeline becomes more complex. It is difficult to cover all test cases for a commercial processor by manual programming.

Table 3.1: Number of test cases

Processor P1 :	12
Processor P2 :	61
Processor P3 :	497

3.1.3 Test sequence for a given test case

Suppose we want to make an instruction sequence that satisfies the test case shown in Figure 3.4. `INT` reaches the ALU one cycle after it is fetched. Also, `FP` reaches the FPU one cycle after it is fetched. Since two instructions cannot be fetched at the same time, it seems that this test case cannot be satisfied. However, another hazard can lead the pipeline to this case. One solution is the sequence `LD–INT–FP–NOP`. `LD` and `INT` cause the primary hazard and `INT` is stalled as shown in Figure 3.5(a). `FP` then overtakes `INT` causing the secondary hazard shown in Figure 3.5(b).

This is an example of a complicated test sequence for a simple pipelined processor. A complex pipelined processor requires a great number of more complicated test sequences. It is difficult to find all sequences, and manual programming is very expensive.

Conventional test program generation methods for pipelined processors [LS91, INH92, INH93] consider only the target hazard. They cannot find complicated test sequences like the one shown above, and cannot avoid unexpected hazards that prevent satisfying test cases.

3.1.4 Efficiency of random tests

We made simulators for three processor pipeline models, P1 (Figure 3.1), P2 (Figure 3.2), and P3 (Figure 3.3), and applied random instruction sequences to them. Figure 3.6 shows the number of test cases covered within each clock cycle. Since P3 normally fetches two instructions per clock cycle, it needs about twice as many instructions as clocks. The clocks include instruction fetch stall cycles.

The random instruction sequence for P1 covered 8 test cases in 390 clock

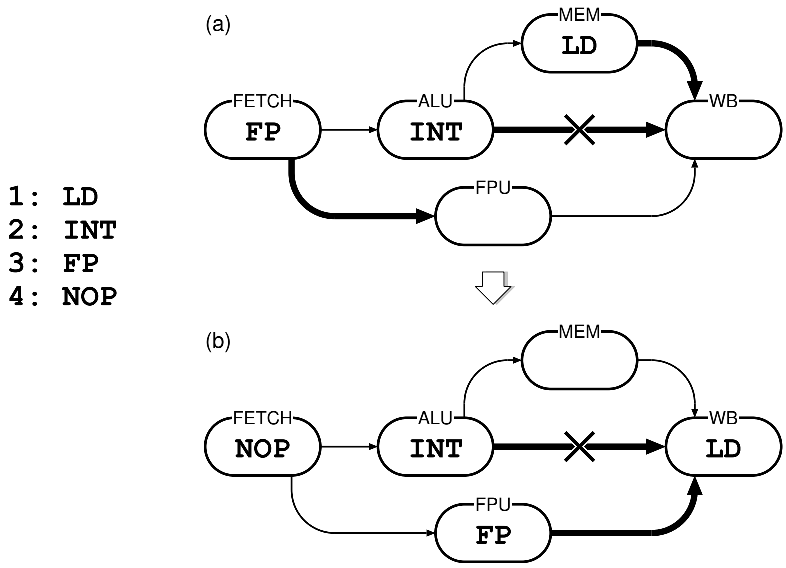


Figure 3.5: An instruction sequence that satisfies the test case

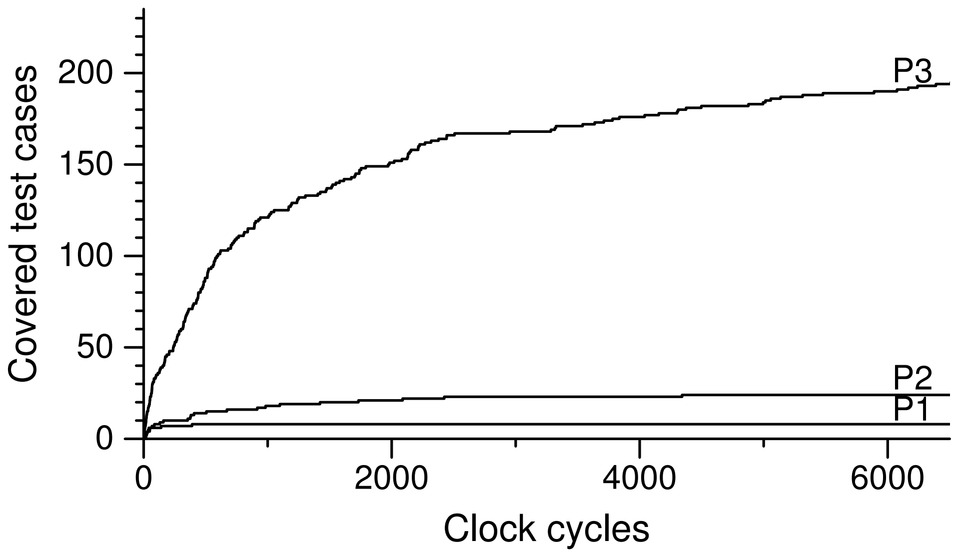


Figure 3.6: Number of test cases covered by random instructions

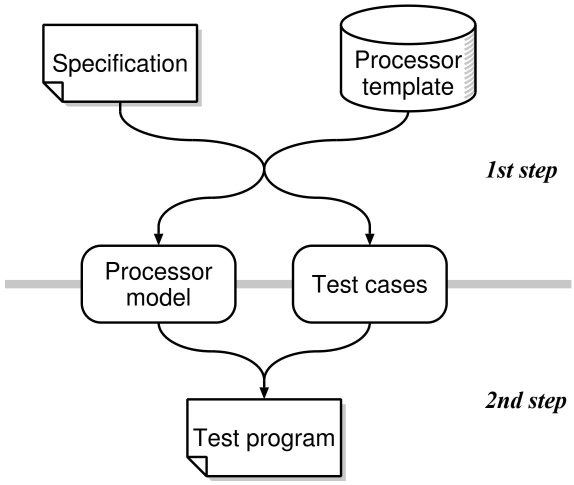


Figure 3.7: Test program generation flow

cycles and reached no other test cases. For P2, the number of covered test cases stabilized after about 3,000 clock cycles and increased slowly from there. For P3, the number of test cases covered went on growing and a saturation point was not reached.

Results show that the number of simulation cycles needed to achieve a reliable test grows rapidly as the processor pipeline becomes more complex. Commercial microprocessors with more pipeline units and more instructions require far more simulation cycles for a random test. It will become impossible for random instructions to cover all test cases if pipeline complexity continues to increase and simulation speed does not greatly improve.

Random tests will always have some value. Random instructions search the entire space equally, and often find unpredictable design errors. However, most of design errors will be in hazardous pipeline states. We have to give a top priority to those places.

3.2 Automatic generation

Our automatic test program generation system generates test programs from processor specifications. Test program generation flow in our system is divided into two steps, as shown in Figure 3.7.

A cycle-accurate processor model is needed to generate a test program.

Also, many test cases are needed to generate a good test program. On the other hand, the specification must be simple enough to be written correctly. From a simple specification, our system generates a processor model and automatically enumerates test cases.

The second step in our system is generating instruction sequences to satisfy the test cases. The difficulty of this problem was discussed in Section 3.1.3. Our system generates instruction sequences to test processor states, even if they can be reached only after pipeline hazards. Instruction sequences generated by the system do not cause unexpected hazards and guarantee satisfying the test cases.

3.2.1 Pipeline specification

Specification must contain the following information to derive all patterns causing hazards and to generate VHDL descriptions:

- Declarations of a set of instruction groups, a set of pipe stages, and input ports of the pipeline controller
- Hardware resources that are used in some pipe stages and may cause hazards
- Actions in each pipe stage to a resource
- Status of a pipe stage when an action is performed

An *instruction group* is a set of instructions that all act the same from the viewpoint of pipeline control. The ADD and SUB instructions, for example, both belong to the ALU instruction group. In each input port declaration, a pipe stage is specified from which input values are taken. These input values become available after execution in the stage. *Status* of a pipe stage contains an instruction group, signal value conditions, and an index signal name for the resource. Signal value conditions consist of expressions comparing signals with constants, which determine instruction attributes including addressing modes and branch conditions. This allows internal signals to be used which are generated in pipe stages. An index for the resource is something like a

(a)

```
ALU
{
  ID:
    occupy GPR;
    read   GPR[RS1];
    read   GPR[RS2];
  WB(RD != 0):
    occupy GPR;
    write  GPR[RD];
}
```

(a′)

```
ALU
{
  ID.1:
    occupy GPR;
    read   GPR[RS1];
  ID.1(imm == 0):
    read   GPR[RS2];
  WB.0(RD != 0):
    occupy GPR;
    write  GPR[RD];
}
```

(b)

```
Branch
{
  EX(cond == 1):
    annul;
  MEM(cond == 1):
    occupy PC;
    write  PC delay 1;
}
```

Figure 3.8: Pipeline specifications for ALU and Branch instruction groups

register address or a memory address. Hazards occur only if two instructions use the same resource with the same indices.

Figure 3.8 (a) shows a simple ALU instruction group specification. The instruction belonging to the group reads from two general purpose registers (GPRs) indexed by two operands (`RS1` and `RS2`) at the ID stage, and writes to a GPR indexed by `RD` at the WB stage unless the index equals zero. When designers like to define a specification in detail, they can add more hardware resources, such as a program counter, a memory, and/or an ALU.

Figure 3.8 (a′) shows a more sophisticated specification for the same group. The suffix after stage indicates a substage. The specification asserts GPR is written to at the first half of the WB stage, while it is read from at the second half of the ID stage. This technique improves the processor performance since GPRs can be read from or written to in the same clock cycle. This example also covers two addressing modes. The actions specified at the ID stage are divided into two. The former is for the common case, and the latter is for a register-register addressing mode. These are distinguished by the control signal `imm`. Typical processors have multiple addressing modes for most instructions. Although multiple addressing modes can be modeled by making different instruction groups for the different addressing modes, the increased number of instruction groups causes processing time to increase rapidly.

Figure 3.8 (b) shows a specification for the branch instruction group with a predict-not-taken scheme. The instruction belonging to the group *annul*s an instruction that follows it at the EX stage and writes to the PC at the MEM stage if `cond` is 1 (taken branch). The keyword *delay* is used to specify the number of instructions to be fetched without waiting for the branch.

3.2.2 Hazard pattern enumeration

Hazards in a basic pipeline are caused when two instructions attempt to use the same hardware resources. A structural hazard occurs if two pipe stages attempt to occupy a common resource in the same clock cycle. A data hazard occurs if they attempt to access a common resource in an illegal order. A control hazard occurs if they attempt to access a program counter in an illegal order. In a general pipeline model, structural hazards may be caused when three or more instructions attempt to use the same kind of hardware resources simultaneously if they exceed the physical number of the resources. We leave this type of pipeline model for future work.

We define a set of hazard patterns $\boldsymbol{H}$ as

$$\boldsymbol{H} = \left\{ (s_1, q_1, s_2, q_2) \;\middle|\; \begin{array}{l} \text{If stage } s_1 \text{ has status } q_1, \\ \text{stage } s_2 \text{ cannot have status } q_2. \end{array} \right\}.$$

Status contains an instruction group, signal value conditions, and an index signal name for the resource.

In the hazard pattern enumeration algorithm (Figure 3.9), $\boldsymbol{C}$ denotes the set of common resources, s a stage, q status, d a number of delay slots, $\boldsymbol{X}_c$ the set of (s,q) when s occupies common resource c, $\boldsymbol{R}_c$ the set of (s,q,d) when s reads data from c, and $\boldsymbol{W}_c$ the set of (s,q,d) when s writes to c. The case where s_2 is a stage before stage s_1 is $s_2 < s_1$. The case where s_2 is a stage before stage s_1, or s_2 and s_1 are the same stage, is $s_2 \leq s_1$. The expression $s+d$ denotes a stage that is d stages after s. The main procedure, *enumerate_hazard_patterns()* makes the hazard pattern set $\boldsymbol{H}$. It enumerates structural hazards, RAW hazards, WAR hazards, and WAW hazards for each common resource. Control hazards are enumerated as special cases of data hazards. The procedure *enumerate_structural_hazards(X)* enumerates all

```
enumerate_hazard_patterns() {
    H ← φ;
    foreach c (c ∈ C) {
        enumerate_structural_hazards(X_c);
        enumerate_data_hazards(W_c, R_c);      /* RAW */
        enumerate_data_hazards(R_c, W_c);      /* WAR */
        enumerate_data_hazards(W_c, W_c);      /* WAW */
    }
}

enumerate_structural_hazards(X) {
    foreach s1,q1 ((s1,q1) ∈ X) {
        foreach s2,q2 ((s2,q2) ∈ X, s2 < s1) {
            H ← H ∪ {(s1,q1,s2,q2)};
        }
    }
}

enumerate_data_hazards(A1, A2) {
    foreach s1,q1,d1 ((s1,q1,d1) ∈ A1) {
        foreach s2,q2,d2 ((s2,q2,d2) ∈ A2, s2 < s1) {
            foreach s (s2 + d1 < s ≤ s1) {
                H ← H ∪ {(s,q1,s2,q2)};
            }
        }
    }
}
```

Figure 3.9: Hazard pattern enumeration algorithm

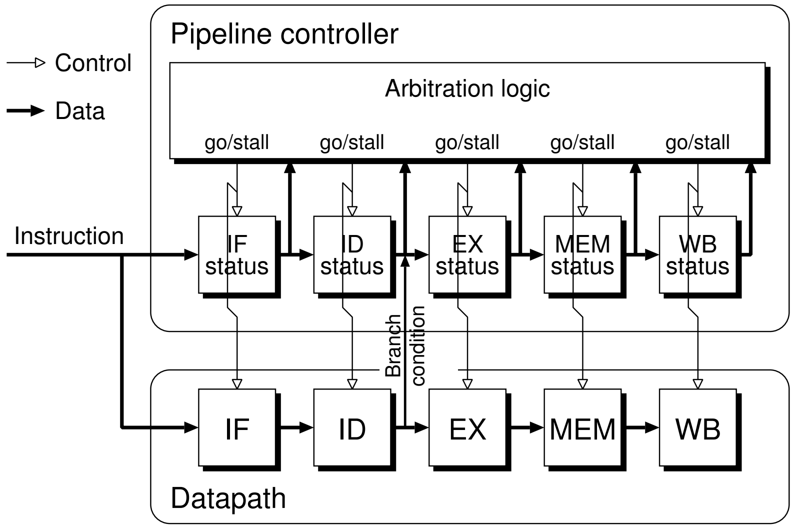


Figure 3.10: Pipeline model

patterns when two pipe stages attempt to occupy the same resource. The procedure *enumerate_data_hazards(*$\boldsymbol{A}_1$*,* $\boldsymbol{A}_2$*)* enumerates all patterns when the order of data access is changed by the pipelined execution, excluding delayed instructions. When an instruction in stage s has d_1 delay slots, instructions in stages $(s - d_1)\ldots(s-1)$ are the delayed instructions.

The hazard patterns corresponding to the hazards shown in Figure 2.8 are as follows:

(a) (EX, (Load; RD$\neq$0; RD), ID, (ALU; – ; RS1))
(b) (MEM, (Load; RD$\neq$0; RD), ID, (ALU; – ; RS1))
(c) (WB, (Load; RD$\neq$0; RD), ID, (ALU; – ; RS1))

This means that a hazard occurs between the (a) EX, (b) MEM, or (c) WB stage of the Load instruction and the ID stage of the ALU when the RD of the Load and the RS1 of the ALU are the same and the RD of the Load is not zero.

3.2.3 Pipeline model generation

The pipeline controller model (Figure 3.10) has status registers for each stage that get input from the previous status register, send output to the next one, and change status every clock cycle unless the corresponding pipe stage stalls. Status includes an opcode, source and destination register addresses, and a branch condition. An arbitration logic gets this information from status registers, checks if a hazard will occur at the next clock cycle, and determines control signals.

The behavioral description of the pipeline controller can be generated from hazard patterns. Each hazard pattern (s_1, q_1, s_2, q_2) corresponds to a condition that stalls stage s_2, which must stall in the next clock cycle when

- The stage before s_1 has status q_1 and the stage before s_2 has status q_2.
- Stage s_1 will run in the next clock cycle if the hazard pattern is for a structural hazard.

$stall_i$ denotes a Boolean variable which is the logical OR of the conditions above and indicates that the ith stage must stall to avoid using common resources in the next clock cycle. $valid_i$ denotes a Boolean variable which shows that the ith stage is processing a valid instruction in the current clock cycle, which is cleared by annulling operations. go_i denotes a Boolean variable showing that the ith stage will run in the next clock cycle. Variable go_i is true when

- Common resources can be used in the next clock cycle.
- The previous stage has a valid instruction.
- If it has a valid instruction, the next stage can get the data in the next clock cycle.

As a result, the following equation is satisfied, where n denotes the number of the stages:

$$\begin{aligned} go_1 &= \overline{stall_1} \cdot (\overline{valid_1} + go_2) \\ go_i &= \overline{stall_i} \cdot valid_{i-1} \cdot (\overline{valid_i} + go_{i+1}) \qquad (2 \le i \le n-1) \\ go_n &= \overline{stall_n} \cdot valid_{n-1} \end{aligned}$$

The pipeline controller consists of combinational logic and n state registers controlled by go_1 to go_n. Signals from input ports are sent to corresponding state registers, and these values flow through the pipeline. Control signals go_1 to go_n are calculated from these values and sent to output ports.

Although the logic may be redundant, its major virtue is that it consists of simple structures. If necessary, conventional logic optimizations and synthesis could be applied.

3.2.4 Test program generation

We introduce a method to generate test sequences for a given processor pipeline model and its test cases. We assume here that the pipeline model is compiled as a state machine represented by Boolean state variables and state transition functions. Each test case is given as a set of machine states corresponding to a hazard pattern. We use reduced ordered binary decision diagrams (*BDD*s) [Bry86] to represent functions and sets.

Pipeline states are encoded as n-dimensional Boolean vectors and instructions are encoded as m-dimensional Boolean vectors. Let $B = \{0, 1\}$ denote a set of Boolean values, $x \in B^n$ a current state, $i \in B^m$ a current input value, and $x' \in B^n$ a next state. The state transition function of the pipeline model is translated to n BDDs representing a n-dimensional Boolean function δ:

$$x' = \delta(x, i)$$

A set of n-dimensional Boolean vectors $A \subseteq B^n$ is represented by a BDD in the form of a *characteristic function* χ_A:

$$\chi_A(x) = 1 \quad \text{iff} \;\; x \in A$$

Union of sets, intersection of sets, and complement of a set can be calculated by logical OR of the BDDs, logical AND of the BDDs, and logical NOT of the BDD.

Basic procedure

A test case is a set of machine states represented by a function $T(x)$. The aim of test program generation is to find input sequences to satisfy that state. Let x_0 be a constant initial state and i_t an input value at clock cycle t. The state at cycle t can be computed by applying the transition function recursively:

$$
\begin{array}{rrcl}
\text{if } t=1, & x_1 & = & \delta(x_0, i_0) \\
 & & \stackrel{\text{def}}{=} & f_1(i_0) \\
\text{if } t \geq 2, & x_t & = & \delta(x_{t-1}, i_{t-1}) \\
 & & = & \delta(f_{t-1}(i_0, \ldots, i_{t-2}), i_{t-1}) \\
 & & \stackrel{\text{def}}{=} & f_t(i_0, \ldots, i_{t-1})
\end{array}
$$

$T(x_t)$ indicates whether state x_t is included in the test case, and $T \circ f_t(i_0, \ldots, i_{t-1})$ — *i.e.* a composite function $T(f_t(i_0, \ldots, i_{t-1}))$ — indicates whether an instruction sequence $i_0, \ldots, i_{t-1}$ satisfies the test case. Thus, $T \circ f_t(i_0, \ldots, i_{t-1})$ is the characteristic function that represents the set of test sequences for case T. A test sequence can be generated by choosing one sequence from $T \circ f_t$. All test sequences that satisfy the case can be enumerated in one pass of the BDD, if necessary.

If $T \circ f_t$ is constant zero, there is no instruction sequence that satisfies the test case at cycle t. We repeat computation for the next cycle until a test sequence is found or the test case is proved to be unreachable.

To check whether the test case is reachable or not, we compute a set of states reachable within each clock cycle. Let A be the input set, C a set of states, and C' the set of states that is reachable from C at the next cycle. C' is the image of the set $C \times A$ after the transition function δ:

$$C' = \{x' \in B^n \,|\, x' = \delta(x, i), x \in C, i \in A\}$$

Efficient image computation algorithms have been proposed for formal verification [TSL+90]. We begin the procedure with $C = \{x_0\}$, and repeat image computations until all reachable states have been enumerated.

Our basic procedure for generating test sequences for a test case T is shown in Figure 3.11. The *output* function generates test sequences by choosing the input values that make $T \circ f_t$ one, and the *image* function computes the image of $C \times A$ using the transition function δ.

$$
\begin{array}{l}
C \leftarrow \{x_0\};\ U \leftarrow C;\ t \leftarrow 0;\ f_0() \leftarrow x_0; \\
\textbf{while}\ (C \neq \phi)\ \{ \\
\quad \textbf{if}\ (C \cap T \neq \phi)\ \{ \\
\quad\quad output(T \circ f_t(i_0, \ldots, i_{t-1})); \\
\quad \} \\
\quad C' \leftarrow image(\delta, C \times A); \\
\quad C \leftarrow C' \cap \overline{U}; \\
\quad U \leftarrow U \cup C'; \\
\quad t \leftarrow t + 1; \\
\quad f_t(i_0, \ldots, i_{t-1}) \leftarrow \delta(f_{t-1}, i_{t-1}); \\
\}
\end{array}
$$

Figure 3.11: Basic procedure

Hazard-free-first procedure

A good test program for a pipelined processor must not cause pipeline hazards that are not related to the test case for two reasons.

- Pipeline hazards make processor behavior more complex and make it more difficult to analyze the cause of the error found by the test.
- Specifications of processor behavior in hazard-free states is simple and reliable, while that in hazard states is complex and prone to errors. Unexpected behavior after a pipeline hazard may prevent the processor from satisfying the test case.

We modified our basic procedure in Figure 3.11 to examine hazard-free state transitions prior to hazardous ones to find hazard-free test sequences. We named this the hazard-free-first procedure. The hazard-free-first procedure removes hazard states from newly reached states and puts them in FIFO storage. Image computations of hazard-free states are repeated until no new hazard-free states are found. The hazard states are then taken from the FIFO and image computations are repeated again (Figure 3.12).

A set of instruction sequences currently being examined, I, is also calculated at each iteration. The procedure checks whether the set of states C

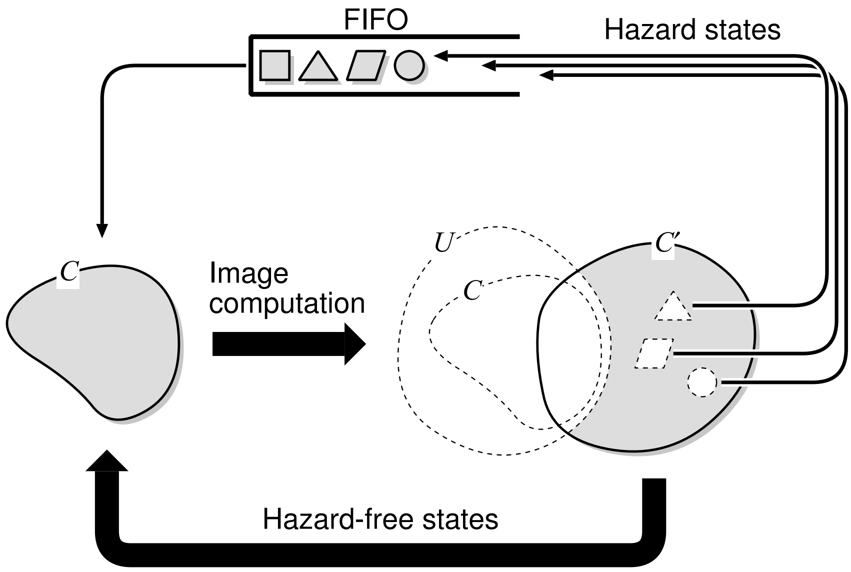


Figure 3.12: Hazard-free-first state enumeration

includes the test case T and generates test sequences for T that are included in I.

3.3 Experimental results

We implemented an experimental system to generate test sequences using the two procedures described in Section 3.2.4. The system is written in Perl and runs on a special Perl interpreter linked with a BDD package written in C.

Execution results of the two procedures for pipeline models P1, P2, and P3 (Figures 3.1, 3.2, 3.3) are summarized in Table 3.2. We generated one test sequence for each reachable test case to construct test programs. We measured CPU times on a SPARCstation2.

The test program generator enumerated all reachable pipeline states and distinguished reachable test cases from unreachable ones. It is difficult to analyze test cases manually, and impossible for conventional test program generation methods to distinguish them. The test programs covered the all test cases that are reachable only after hazards, such as the case shown in Figure 3.5. They are difficult cases to be handled manually, and cannot be covered by conventional test program generation methods.

Results show that computations completed in reasonable CPU/memory

Table 3.2: Execution summary of basic/hazard-free-first procedures

	P1	P2	P3
Pipeline units	5	9	11
Instructions per cycle	1	1	2
FSM states	240	190512	9.335×10^{6}
Test cases	12	61	497
Reachable FSM states	125	16747	1.851×10^{6}
— only after hazards	28	7236	1.244×10^{5}
Reachable test cases	8	25	285
— only after hazards	3	9	0
Test program length	27/ 27	127/ 127	2289/2516
CPU time (seconds)	10/ 13	83/ 90	495/ 579
Max. BDD nodes	2K/599	37K/2658	117K/5797

requirements, and also show that the hazard-free-first procedure is comparable to the basic procedure in CPU time, and superior in memory requirements.

Figure 3.13 shows the percentage of reachable test cases covered by the test programs and random instructions within each clock cycle. The test programs are generated by the hazard-free-first procedure. The random instruction data is the same as that in Figure 3.6. Random instructions need a large number of clock cycles to achieve high coverage, while our test programs can achieve perfect coverage in a small number of clock cycles.

The system can also analyze reachability of test cases. The number of test cases expected to be covered by random instructions in each clock cycle is calculated by the system. Percentages of reachable test cases covered by the test programs and random instructions are plotted in Figure 3.14. About 360 clock cycles of random simulation is needed for P1 to guarantee 99% coverage, 9,600 cycles for P2, and 90,000 cycles for P3. Our test programs are 13 to 76 times smaller than 99% coverage random instructions.

3.4 Chapter summary

We have demonstrated the need for automatic test program generation and shown how we realized it for pipelined processors. We have introduced an

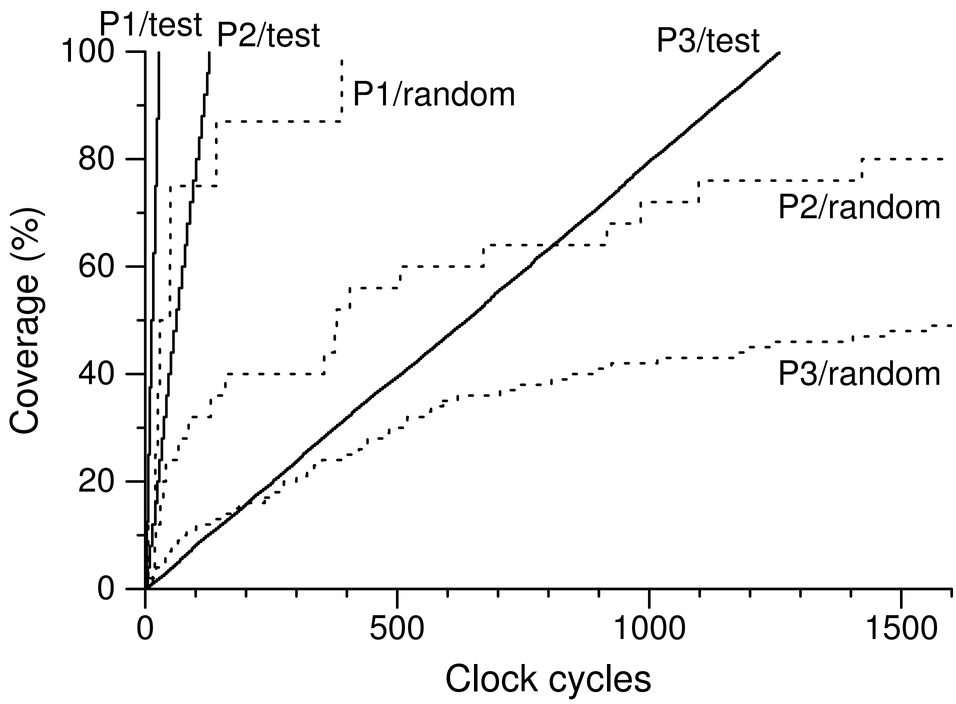


Figure 3.13: Test coverage by test programs and random instructions

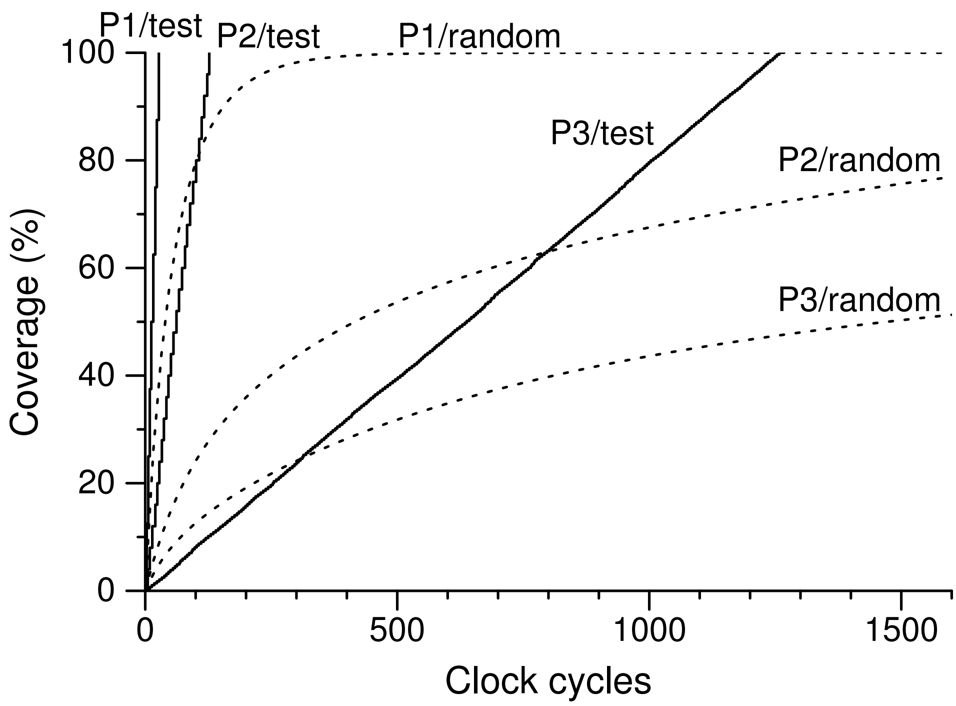


Figure 3.14: Test coverage by test programs and random instructions

automatic test program generator, which is divided into two parts. The first part generates an FSM of the pipeline and enumerates test cases from a simple processor specification. When designers define the hardware resources used at each pipe stage for each instruction group, the system enumerates all hazard patterns and generates a cycle-accurate model of the pipeline.

The second part generates instruction sequences to satisfy the test cases for the pipeline model. It is implemented by utilizing techniques developed for formal verification. We also presented the hazard-free-first state enumeration technique for pipelined processors, which reduces memory requirement. Our method can generate the test programs that are difficult to code manually and that are impossible to generate by conventional methods. Random instructions need a large number of clock cycles to achieve high test coverage, while our test programs can achieve perfect test coverage in a small number of clock cycles.

CHAPTER 4

Practicality-oriented Symbolic Model Checking Methods

Model checking is one of the standard hardware verification methods in many industrial fields. Traditionally, language containment check [HK90, TBK95, HTKB93] and CTL model checking [CES86, McM93, CGMZ95] are two major methods of model checking. Language containment is usually checked after building a product machine, which is an interconnected system of design and property automata. Standard CTL model checking algorithm is based on backward state traversal over the design model, in contrast to the product machine for language containment check.

Symbolic techniques using Binary Decision Diagrams (BDDs) [Bry86] are the important keys of modern model checking algorithms [BCM$^+$92, TBK95, HTKB93, McM93]. Although they have potential power of verifying a finite state system with several hundred Boolean state variables, actual performance heavily depends on the system's structure and state traversal heuristics of the model checker. Even now, model checking of a complicated system cannot be accomplished without human guidance based on their knowledge of both the target system and the model checker. We need a flexible framework of symbolic model checking in order to aid the human efforts and to automate the total process.

Model	Monolithic TR			Partitioned TR		
	#Nodes	Image	Pre-image	#Nodes	Image	Pre-image
atm_sw	1302K	8.9	21.8	1K	10.9	202.5
dh_1	3K	0.0	0.8	1K	0.3	1.3
dh_2	100K	13.8	75.4	35K	46.8	>10000
vpp	N/A	N/A	N/A	60K	3.0	4135.1
pipe_s	322K	10.7	14.8	3K	0.6	387.9
pipe_d	N/A	N/A	N/A	348K	200.4	>20000

Table 4.1: CPU seconds per image or pre-image computation

4.1 Forward versus backward traversal

Conventional CTL model checkers evaluate CTL formulas with repeated pre-image computation, *backward state traversal*. Properties $s_0 \models \mathbf{EF}\, f$ and $s_0 \models \mathbf{AG}\, f$ are also known to be verified by comparing $\mathscr{L}(f)$ and the reachable states. Reachable states are enumerated with repeated image computation, *forward state traversal*.

Performance of the computation is very sensitive to BDD variable ordering. It is difficult to find a good variable order automatically, and ordinary users cannot always find it manually. When we use conjunctive partitioned transition relations, the performance is also sensitive to the order in which the BDDs are processed. In our experience of industrial hardware verification, however, *image* computation with *partitioned* transition relation works relatively fine even if the FSM is very large and the ordering is not tuned so much.

Table 4.1 shows average CPU time per image or pre-image computation during each model checking process. We used both a monolithic transition relation represented by a single BDD and a conjunctive partitioned transition relation represented by a set of BDDs for latch transition relations. Total numbers of BDD nodes for monolithic/partitioned transition relations are also shown in the table. Monolithic transition relations for models `vpp` and `pipe_d` could not be made because of BDD size explosions. Pre-image computation with partitioned transition relation for models `dh_2` and `pipe_d` exceeded the CPU time limit of 24 hours. We should not compare image computation time and pre-image computation time directly, because they are solving different problems. The results, however, show that the time difference of image computation and pre-image computation with a partitioned transition relation is huge, while that with a monolithic transition relation is relatively small.

4.2 Forward model checking of CTL properties

As described in Section 4.1, we often find a large CPU time difference between image computation and pre-image computation especially when a conjunctive partitioned transition relation is used. We should traverse state space forward in such cases. In the symbolic model checking paradigm, CTL formulas have been evaluated with backward state traversal. We describe, in this section, an algorithm to accomplish CTL model checking in the opposite direction. It is effective in many situations where backward state traversal is more expensive than forward state traversal.

4.2.1 Rewriting property notations

A CTL property is given as a notation like "$s_0 \models f$." Conventional model checking procedure matches the notation: it evaluates CTL formula f with backward state traversal, and then checks if it holds at state s_0. We rewrite the notation for the purpose of matching it with our method. We translate the CTL property into a problem of comparing a formula with the constant false. Let s_0 be a state of the FSM, p_0 the characteristic function of $\{s_0\}$, and f an arbitrary CTL formula. Formula p_0 is true only at state s_0, and formula f is true at state s_0 iff $s_0 \models f$ holds. Therefore, the "$\models$" notation can be rewritten as follows:

$$s_0 \models f \iff p_0 \wedge f \neq \text{false}, \tag{4.1}$$

$$s_0 \models f \iff p_0 \wedge \neg f = \text{false}. \tag{4.2}$$

Some model checkers support models with multiple initial states, while "$\models$" represents relation between a single state and a CTL formula. Given a set of initial states S_0, we believe that interpretation of some extended notation like "$S_0 \models f$" is ambiguous. It should be written as "$\exists s \in S_0$, $s \models f$" or "$\forall s \in S_0$, $s \models f$". They can be rewritten as "$p_0 \wedge f \neq \text{false}$" and "$p_0 \wedge \neg f = \text{false}$" respectively where p_0 is the characteristic function of S_0.

4.2.2 Forward EX evaluation

Let p and f be formulas. We can replace an outermost **EX** evaluation with image computation as follows:

$$p \wedge \mathbf{EX}\, f \neq \text{false} \iff \mathit{Img}\,(p) \wedge f \neq \text{false}\,. \tag{4.3}$$

Proof. Assume $p \wedge \mathbf{EX}\, f$ holds at state s. Then p holds at s and f holds at some successor state of s, say t. $\mathit{Img}\,(p)$ holds at t since $\mathit{Img}\,(p)$ holds at any successor state of $s \in \mathscr{L}(p)$. Thus, $\mathit{Img}\,(p) \wedge f$ holds at t. Conversely, assume $\mathit{Img}\,(p) \wedge f$ holds at state t. Then f holds at t and t is a successor state of some state $s \in \mathscr{L}(p)$. $\mathbf{EX}\, f$ holds at s since $t \in \mathscr{L}(f)$ is a successor state of s. Thus, $p \wedge \mathbf{EX}\, f$ holds at state s. □

Notice that we have removed an operator **EX** from f. Using equation (4.3) again or using one of the equations described later, it is possible to continue conversion of a backward traversal operator in f into a forward traversal operator.

4.2.3 Forward EU evaluation

We define a state enumeration procedure under constraints given by two formulas p and q:

$$\mathit{FwdUntil}\,(p,q) \;=\; \mathbf{lfp}\, Z\, [p \vee \mathit{Img}\,(Z \wedge q)]\,.$$

An element of $\mathscr{L}(\mathit{FwdUntil}\,(p,q))$ is a state t such that there exists a path through t from some state at which p holds, and q holds at all states before t on the path.

Using the *FwdUntil*() operator, we can replace an outermost **EU** evaluation as follows:

$$\begin{aligned} & p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f] \neq \text{false} \\ & \qquad \Longleftrightarrow \quad \mathit{FwdUntil}\,(p,q) \wedge f \neq \text{false}\,. \end{aligned} \tag{4.4}$$

Proof. Assume $p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f]$ holds at state s. Then both p and $\mathbf{E}\,[q\,\mathbf{U}\,f]$ hold at s. It means that there exists a path from s through some state $t \in \mathscr{L}(f)$, and q holds at all states before t. Thus, $\mathit{FwdUntil}\,(p,q) \wedge f$ holds at t. Conversely, assume $\mathit{FwdUntil}\,(p,q) \wedge f$ holds at state t. Then both $\mathit{FwdUntil}\,(p,q)$ and f holds at t. There exists a path through t from some state $s \in \mathscr{L}(p)$, and q holds at all states before t. Thus, $p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f]$ holds at s. □

Now the operator **EU** have been removed from f. Thus, we have a chance again to convert a backward traversal operator in f into a forward traversal operator, as in equation (4.3).

4.2.4 Forward EG evaluation

We define an operator like **EG**, except that pre-image computation is replaced by image computation:

$$\mathit{EH}\,(p) \;=\; \mathbf{gfp}\,Z\,[p \wedge \mathit{Img}\,(Z)]\,.$$

$\mathit{EH}\,(p)$ is used to check whether there exists a state transition cycle in $\mathscr{L}(p)$. $\mathscr{L}(\mathit{EH}\,(p))$ is the subset of $\mathscr{L}(p)$ such that every state is reachable from a cycle through states only in $\mathscr{L}(p)$. We also define simple composite operators:

$$\begin{aligned} \mathit{Reachable}\,(p,q) &= \mathit{FwdUntil}\,(p,q) \wedge q\,, \\ \mathit{FwdGlobal}\,(p,q) &= \mathit{EH}\,(\mathit{Reachable}\,(p,q))\,. \end{aligned}$$

$\mathit{Reachable}\,(p,q)$ computes the subset of $\mathscr{L}(q)$ whose elements can be reached from $\mathscr{L}(p \wedge q)$ through states only in $\mathscr{L}(q)$. $\mathit{FwdGlobal}\,(p,q)$ checks whether there exists a state transition cycle in $\mathscr{L}(q)$ that is reachable from $\mathscr{L}(p \wedge q)$ through states only in $\mathscr{L}(q)$.

Using the *FwdGlobal*() operator, we can replace an outermost **EG** evaluation as follows:

$$\begin{aligned} & p \wedge \mathbf{EG}\, q \neq \text{false} \\ & \quad \Longleftrightarrow \quad \mathit{FwdGlobal}(p,q) \neq \text{false}\,. \end{aligned} \tag{4.5}$$

Proof. Assume $p \wedge \mathbf{EG}\, q$ holds at state s. Then both p and $\mathbf{EG}\, q$ holds at s. It means that for some path from s, q keeps holding forever on the path. In other words, there exists a cycle in $\mathscr{L}(q)$ and it is reachable from s through states only in $\mathscr{L}(q)$. $\mathscr{L}(\mathit{Reachable}(p,q))$ includes the cycle, since it includes all the states reachable from $s \in \mathscr{L}(p \wedge q)$ through states only in $\mathscr{L}(q)$. Thus, $\mathit{EH}(\mathit{Reachable}(p,q)) \neq$ false. Conversely, assume $\mathit{EH}(\mathit{Reachable}(p,q)) \neq$ false. There exists a cycle in $\mathscr{L}(\mathit{Reachable}(p,q))$. It means that the cycle is in $\mathscr{L}(q)$ and is reachable from some state $s \in \mathscr{L}(p)$ through states only in $\mathscr{L}(q)$. Thus, $p \wedge \mathbf{EG}\, q$ holds at s. □

4.2.5 Forward fair EG evaluation

We also introduce fairness constraints into forward CTL evaluation. The key is exactly like ordinary fair CTL evaluation, a procedure to find fair cycles. The procedure that compute $\mathit{EH}(p)$ under fairness constraints C is given as follows:

$$\mathit{FairEH}(p) \;=\; \mathbf{gfp}\, Z \left[p \wedge \mathit{Img}\left(\bigwedge_{c \in C} \mathit{Reachable}(c, Z) \right) \right].$$

We then modify the *FwdGlobal*() operator to handle fairness constraints using *FairEH*():

$$\mathit{FwdFairGlobal}(p,q) \;=\; \mathit{FairEH}(\mathit{Reachable}(p,q))\,.$$

Using the *FwdFairGlobal*() operator, we can replace an outermost **EG** evaluation under fairness constraints as follows:

$$\begin{aligned} & p \wedge \mathbf{E}_C\mathbf{G}\, q \neq \text{false} \\ & \quad \Longleftrightarrow \quad \mathit{FwdFairGlobal}(p,q) \neq \text{false}\,. \end{aligned} \tag{4.6}$$

It is clear from the fact that both sides are the modified version of equation (4.5) that restrict paths under the same constraints.

4.2.6 The conversion procedure

Using conversion rules (4.3), (4.4), (4.5), and (4.6), we can replace **EX**, **EU**, **EG**, and $\mathbf{E}_C\mathbf{G}$ with forward traversal operators. An original property notation should be rewritten using either positive form (4.1) or negative form (4.2) so that the formula matches one of the rules. The problem of comparing a disjunctive expression with the constant false, such as "$f \vee g \neq$ false", can be divided into sub-problems, such as "$f \neq$ false" and "$g \neq$ false". We can check each term separately, and if one or more terms are not the constant false, the entire expression is not the constant false. We do not need to convert all CTL temporal operators into forward traversal operators. Remaining operators can be evaluated in usual manner, with backward state traversal. Hence, all CTL formulas can be handled with our method. The conversion procedure is shown below:

1. Rewrite the CTL formula only in temporal operators **EX**, **EU**, **EG**, and $\mathbf{E}_C\mathbf{G}$.
2. Translate "$\models$" notation into an expression comparing a formula with the constant false, using equation (4.1) or (4.2).
3. Arrange outermost logical operations in disjunctive form, and divide the problem into a set of sub-problems comparing each product term with the constant false.
4. For each sub-problem, convert a backward operator to a forward operator using one of equations (4.3), (4.4), (4.5), and (4.6), if applicable.
5. For each newly updated sub-problems, call the procedure recursively from step 3.

Although steps 2 and 4 have choice, it is easy to find good conversion for actual CTL properties. Many properties that we examined can be fully converted to forward state traversal problems, as shown in the next section.

Example Here is an example of converting one of the most common properties, "whenever a request is made, acknowledgment will return in the future," where *req* means the request, *ack* means the acknowledgment, s_0 is the initial state, and p_0 is the characteristic function of $\{s_0\}$:

$$
\begin{aligned}
& s_0 \models \mathbf{AG}\,(req \to \mathbf{AF}\,ack) \\
\iff\quad & s_0 \models \neg\mathbf{E}\,[\text{true}\,\mathbf{U}\,(req \wedge \mathbf{EG}\,\neg ack)] \\
\iff\quad & p_0 \wedge \mathbf{E}\,[\text{true}\,\mathbf{U}\,(req \wedge \mathbf{EG}\,\neg ack)] = \text{false} \\
\iff\quad & \mathit{FwdUntil}\,(p_0, \text{true}) \wedge (req \wedge \mathbf{EG}\,\neg ack) = \text{false} \\
\iff\quad & (\mathit{FwdUntil}\,(p_0, \text{true}) \wedge req) \wedge \mathbf{EG}\,\neg ack = \text{false} \\
\iff\quad & \mathit{FwdGlobal}\,((\mathit{FwdUntil}\,(p_0, \text{true}) \wedge req), \neg ack) = \text{false}\,.
\end{aligned}
$$

4.3 Forward model checking of ω-regular properties

In this section, we present a model checking algorithm based on symbolic forward state traversal over the design model, which can check if there is a possibility that the design generates some trace matched by an ω-regular expression. Some non-symbolic solutions to similar problems have already been introduced [Hir89, HHY89]; however, they are based on explicit state traversal, which is not realistic for large design models. This algorithm is a generalization of forward model checking techniques [INH96, IN97, TSN98].

We also propose an efficient implementation of the algorithm, which makes explicit data structure of a non-deterministic state transition graph for the ω-regular property. State space of the design model is implicitly traversed along the explicit graph. Each node on the graph is used as a working data storage for the computation. This method should become a computational framework with a large amount of flexibility. We can control the state traversal strategies on this framework in order to get the maximum efficiency from BDD-based symbolic techniques. Various improved techniques for reachability analysis [RS95, CCQ96, NIJ$^+$97] should also be applicable on this framework.

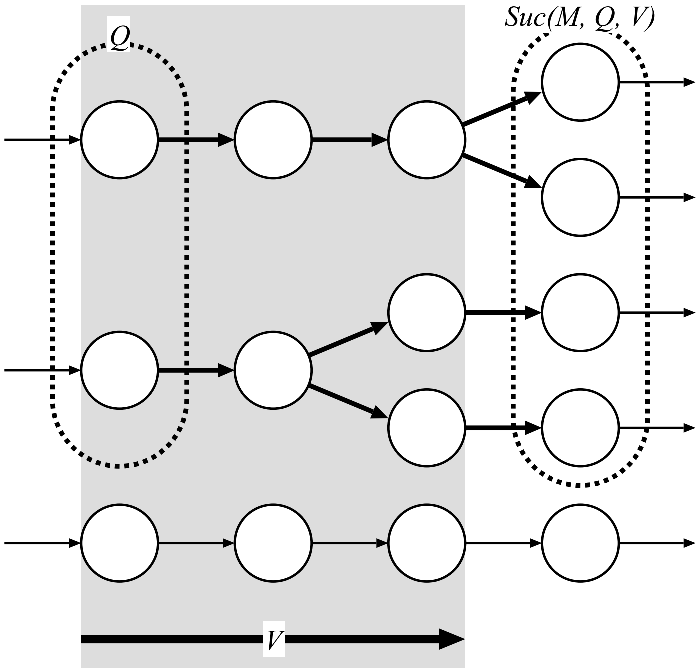


Figure 4.1: An illustration of $Suc(M,Q,V)$

4.3.1 Algorithm for ω-regular properties

Our goal is to check if there is a possibility that a given design may generate one or more instances of a given set of error traces (illegal state transition sequences). The design is modeled by an FSM, $M = (S, I, O, \delta, \lambda, S_0)$. A set of error traces of M is given as an ω-regular expression, considering that the alphabet consists of the states ($\Sigma = S$). This is a kind of language emptiness check.

The key function of the algorithm is $Suc(M,Q,V)$, where M is the FSM, $Q \subseteq S$ is a set of start states, V is a $*$-regular expression for traces of M. The result value is a set of all possible successor states to the traces from Q that are matched by V (Figure 4.1). When M is an FSM, Q and P are sets of states, U and V are $*$-regular expressions, and $Img(M,Q)$ is the image computation function, the following equations hold:

$$
\begin{aligned}
Suc(M,Q,\varepsilon) &= Q\,,\\
Suc(M,Q,P) &= Img(M,Q\cap P)\,,\\
Suc(M,Q,U\cup V) &= Suc(M,Q,U)\cup Suc(M,Q,V)\,,\\
Suc(M,Q,UV) &= Suc(M,Suc(M,Q,U),V)\,,\\
Suc(M,Q,V^*) &= \bigcup_{0\le k<\omega} Suc\left(M,Q,V^k\right).
\end{aligned}
$$

Function $Suc()$ for arbitrary $*$-regular expressions can be computed by recursive application of the above equations.

In order to check if there is a possibility that M may generate one or more instances of UV^ω, we compute a sequence of state sets Q_0, Q_1, $\ldots$ as follows:

$$
Q_i = \begin{cases} Suc(M,S_0,U) & (i=0),\\ Suc(M,Q_{i-1},V) & (i\ge 1). \end{cases}
$$

The Q_i computation is repeated until $i=n$ where n satisfies either case listed below:

Case 1: $Q_n=\emptyset$.

Case 2: $\begin{cases} \exists m,\ \ 0\le m<n,\ \ Q_m\ne\emptyset,\\ Q_m\subseteq \bigcup\limits_{m<k\le n} Q_k. \end{cases}$

No trace of M is matched by UV^ω in the first case and one or more traces of M are matched by UV^ω in the second case. Assuming the first case, no trace from some state in S_0 has a prefix matched by UV^n, therefore no trace from some state in S_0 is matched by UV^ω. Assuming the second case, every state in Q_m is also included in at least one of $Q_{m+1},\ldots,Q_n$. It means that every state in Q_m is a successor state of a trace from some state in Q_m. We can repeat retracing such paths and can visit Q_m infinitely often. Since the state space is finite, we eventually visit some state q in Q_m twice. Thus, there is a cycle from q to q along V, which is reachable from some state in S_0 along U. It causes an infinite trace matched by UV^ω.

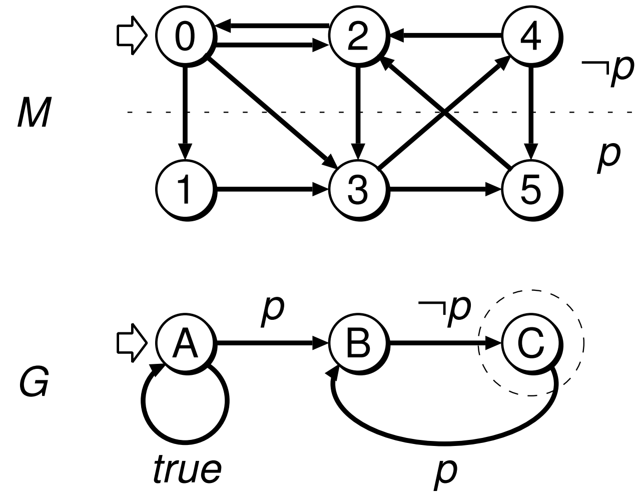


Figure 4.2: Model and property examples

4.3.2 An efficient implementation of the algorithm

In this section, we show an efficient method to implement the forward model checking algorithm. It can be a simple framework with a large amount of flexibility for controlling state traversal on the property space. Breadth-first traversal, depth-first traversal, subsetting, and mixture of them can be characterized as variations in this framework.

Sets of states, functions, and relations of an FSM are represented symbolically using BDDs, in the same way as conventional symbolic model checking tools. A property given in ω-regular expression is translated literally into data structure of a state transition graph, called *property graph*, with labeled edges and an acceptance condition. State space of the FSM is traversed implicitly along the explicit graph. Each node on the graph is used as a working data storage for the computation. First we show an overview using simple examples, then we show the model checking procedure in detail.

Overview of the method

In this section, we use the examples of design model M and property graph G shown in Figure 4.2. M has six states ('0', '1', '2', '3', '4', '5'), and the initial state of M is '0'. A condition p holds at states '1', '3', and '5'. G has three nodes ('A', 'B', 'C'), and the initial node of G is 'A'. G represents a non-

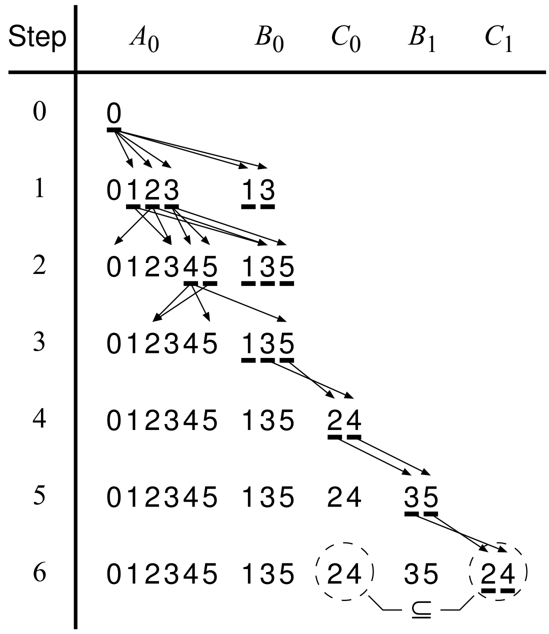


Figure 4.3: Breadth-first checking

deterministic finite automaton for an ω-regular expression $[\text{true}]^*([p][\neg p])^\omega$, where notation $[f]$ stands for a set of states on the design model that satisfies formula f. An infinite sequence is accepted when node 'C' is visited infinitely often.

Figure 4.3 shows one execution trace of the algorithm. A_k, B_k, and C_k are subsets of M's states stored at 'A', 'B', and 'C' respectively, whose index k stands for the number of times passed through the cycle node 'C' for the current computation path on G. Initially, $A_0 = \{0\}$ and others are empty. At step-1, image of state set A_0 on the design model is computed symbolically and we get $\{1,2,3\}$. It is propagated along the edges from node 'A'. One is the self-loop edge labeled true. The result set $\{1,2,3\}$ is directly propagated to node 'A' itself and it is merged into A_0. The other is the edge to 'B' labeled p. The result set is filtered by condition p and then $\{1,3\}$ is merged into B_0. After step-1, we get $A_0 = \{0,1,2,3\}$ and $B_0 = \{1,3\}$. Sets of underlined state numbers in Figure 4.3, i.e. $\{1,2,3\}$ in A_0 and $\{1,3\}$ in B_0 after step-1, represent pending event sets to be processed. Figure 4.3 shows the case when we choose the event set $\{1,2,3\}$ in A_0 for step-2. Image of $\{1,2,3\}$ is propagated similarly to nodes 'A' and 'B'. In general, the procedure terminates either when all events are processed or when some acceptance condition

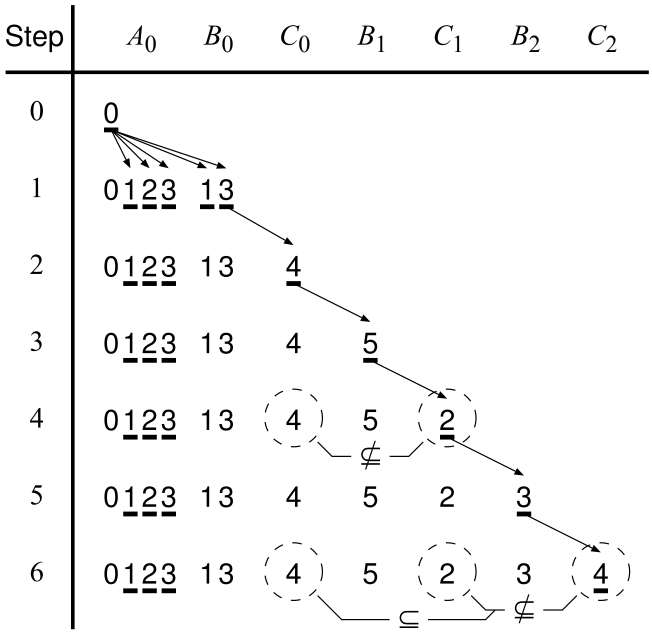


Figure 4.4: Depth-first checking

is satisfied. The emptiness check passes on the former case and it fails on the latter case. In this example, we find after step-6 that every element in C_0 appears again in C_1. It means that node 'C' is visited infinitely often if we repeat the computation steps infinitely. Thus the procedure terminates after step-6 and we find that some sequence on M is accepted by G, i.e. the emptiness check failed.

Figure 4.4 shows another execution trace of the algorithm. In this case, priority is given over the event set on the deepest level. After step-6, we find that every element in C_0 appears again in C_1 or C_2 by checking $C_0 \subseteq (C_1 \cup C_2)$. The procedure terminates after step-6 and we find that the emptiness check failed. Priority of the event processing may affect computational cost of the algorithm; however, it does not affect the result of the algorithm.

A property graph example for an invariant checking problem is shown in Figure 4.5. We can find that there is no essential difference between our method on this graph and the conventional invariant check algorithm based on forward reachability analysis. Various improved techniques of reachability analysis [RS95, CCQ96, NIJ$^+$97] are also applicable to solve this problem. It is not difficult to extend them against general property graphs.

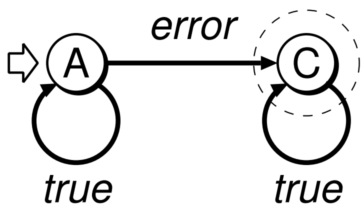


Figure 4.5: Property graph for invariant checking

```
CheckEmptiness (M, G) {
    q0 ← M.initial_condition;
    n0 ← G.initial_node;
    if (Propagate (q0, G, n0, 0)) return "fail";
    while (not QueueIsEmpty ()) {
        (n, i) ← Dequeue ();
        q ← Evaluate (M, n, i);
        if (Propagate (q, G, n, i)) return "fail";
    }
    return "pass";
}
```

Figure 4.6: *CheckEmptiness* function

The detailed procedure

The main function $\mathit{CheckEmptiness}(M,G)$ is shown in Figure 4.6. For design model M, $M.\mathit{initial\_condition}$ is the symbolic representation of the initial state set, i.e. the formula that is true only at the initial states of M. Function $\mathit{Img}(M,q)$ is the only one basic operation on M required for the algorithm, which computes image of p symbolically. For property graph G, $G.\mathit{initial\_node}$ is the initial node, $G.\mathit{cycle\_node}$ is the cycle node representing the acceptance condition, and $G.\mathit{edge\_set}$ is the set of edges $\{(n,p,m) \mid$ There is an edge from node n to node m labeled $p.\}$. Data structure for node n has two lists indexed by integer i, $n.\mathit{total}[i]$ and $n.\mathit{event}[i]$, where a set of propagated states and a set of unprocessed states for each index i are stored respectively.

We use a priority queue of node/index pairs for event management. Func-

```
Evaluate(M,n,i) {
    q ← n.event[i];
    n.event[i] ← false;
    return Img(M,q);
}
```

Figure 4.7: *Evaluate* function

tion *Enqueue* (n,i) inserts a node/index pair into the queue, function *Dequeue* $()$ takes a node/index pair with the highest priority out of the queue, function *Queued* (n,i) checks if the pair (n,i) is already inserted in the queue, and function *QueueIsEmpty* $()$ checks if the queue is empty.

Function *Evaluate* (M,n,i) processes the pending event on (n,i) and returns a set of states to be propagated along the edges from node n (Figure 4.7).

Function *Propagate* (q,G,n,i) propagates state set q along the edges from node n and inserts new events into the queue (Figure 4.8). Index i is incremented if n is the cycle node. When the acceptance condition of G is satisfied, the function terminates and returns 1.

Function *CycleIsFound* (n) checks if the acceptance condition of the cycle node n is satisfied (Figure 4.9). This function corresponds to the second terminal case of the basic algorithm described in Section 4.3.

In this algorithm, event priority can be modified without restriction. Moreover, an event can be partitioned into a set of sub-events, and they can be processed on different schedules. Figure 4.10 gives more generic implementation of function *Evaluate* (M,n,i).

4.4 Experimental results

4.4.1 Applicability to actual CTL properties

Our method becomes effective when many temporal operators in a CTL formula are converted into our forward traversal operators. We investigated

```
Propagate(q, G, n, i) {
    if (n = G.cycle_node)
        i ← i + 1;
    foreach m s.t. (n, p, m) ∈ G.edge_set {
        r ← q ∧ p ∧ ¬m.total[i];
        if (r ≠ false) {
            m.total[i] ← m.total[i] ∨ r;
            m.event[i] ← m.event[i] ∨ r;
            if (m = G.cycle_node and
                CycleIsFound(m)) return 1;
            if (not Queued(m, i))
                Enqueue(m, i);
        }
    }
    return 0;
}
```

Figure 4.8: *Propagate* function

examples in two existing symbolic model checkers SMV [McM93] and VIS [BHSV+96], and also examined our own property examples.

We found that 90% of the properties can be rewritten using only the forward traversal operators: 18 properties out of 20 in the SMV examples, 47 properties out of 55 in the VIS examples, and all of our 13 properties. The rest of the properties are classified into three types:

$$s_0 \models \mathbf{AG}\,\mathbf{EF}\,a$$

$$\iff \quad \mathit{FwdUntil}(p_0, \mathrm{true}) \wedge \neg\mathbf{EF}\,a = \mathrm{false}\,,$$

$$s_0 \models \mathbf{AG}\,(a \rightarrow \mathbf{EG}\,b)$$

$$\iff \quad \mathit{FwdUntil}(p_0, \mathrm{true}) \wedge a \wedge \neg\mathbf{EG}\,b = \mathrm{false}\,,$$

$$s_0 \models \mathbf{AG}\,((a \rightarrow \mathbf{EX}\,b) \wedge (b \rightarrow \mathbf{EX}\,a))$$

$$\iff \quad \begin{cases} \mathit{FwdUntil}(p_0, \mathrm{true}) \wedge a \wedge \neg\mathbf{EX}\,b = \mathrm{false} \\ \mathit{FwdUntil}(p_0, \mathrm{true}) \wedge b \wedge \neg\mathbf{EX}\,a = \mathrm{false}\,, \end{cases}$$

where a and b are atomic propositions. Both forward and backward traversal

CycleIsFound (n) {
    if (n has a self-loop labeled true)
        return 1;
    $r \leftarrow$ false;
    $k \leftarrow$ maximum index of n;
    while $(k \geq 0)$ {
        $q \leftarrow n.total[k]$;
        if ($q \wedge \neg r =$ false) **return** 1;
        $r \leftarrow r \vee q$;
        $k \leftarrow k - 1$;
    }
    return 0;
}

Figure 4.9: *CycleIsFound* function

Evaluate (M, n, i) {
    Choose some condition q such that
        $q \wedge \neg n.total[i] =$ false
        and $q \wedge n.event[i] \neq$ false;
    $n.event[i] \leftarrow n.event[i] \wedge \neg q$;
    if ($n.event[i] \neq$ false) *Enqueue* (n, i);
    return *Img* (M, q);
}

Figure 4.10: Revised *Evaluate* function

are used to check these properties. They are, in fact, the same as conventional methods that can unfold only outermost **AG** operators.

4.4.2 Performance of model checking

We have implemented a model checker, named BINGO, and applied it to our industrial benchmark examples and the VIS [BHSV$^+$96] examples. Table 4.2 summarizes number of Boolean state variables, number of reachable

Table 4.2: Benchmark examples

Model	FFs	States	Depth	Description
atm_sw	54	2.0×10^{5}	126	ATM-switch [CYF94]
dh_1	46	4.0×10^{3}	21	bus protocol
dh_2	66	7.9×10^{6}	17	cache coherence
dh_3	96	3.7×10^{8}	40	cache coherence
vpp	101	2.4×10^{11}	18	VLIW pipeline
pipe_s1	35	4.4×10^{7}	11	superscalar pipeline
pipe_s2	35	4.5×10^{7}	11	superscalar pipeline
pipe_d1	73	5.7×10^{17}	11	superscalar pipeline
pipe_d2	73	5.7×10^{17}	11	superscalar pipeline
elevator	32	6.7×10^{5}	27	VIS example
ether213	118	7.0×10^{4}	81	VIS example

states, and number of image computations to reach the fixed-point on reachability analysis. We checked one property for each model except `dh_2` and `ether213`. For `dh_2` and `ether213`, we checked one failing property and one passing property. Models `pipe_s2` and `pipe_d2` are revised versions of `pipe_s1` and `pipe_d1` respectively. The property passes on the later version, while it fails on the earlier version. All properties are originally written in CTL. We have rewritten it into a complement ω-regular expression by hand for each benchmark. BINGO reads an ω-regular expression and builds the corresponding property graph. A typical CTL property is $\mathbf{AG}(p \to \mathbf{AF}q)$, which is rewritten to $[\text{true}]^*[p \wedge \neg q][\neg q]^{\omega}$. If the property has fairness constraints $C_1, \ldots, C_n$, which must be satisfied infinitely often, the regular expression is modified to $[\text{true}]^*[p \wedge \neg q]([\neg q \wedge \neg C_1]^*[\neg q \wedge C_1] \cdots [\neg q \wedge \neg C_n]^*[\neg q \wedge C_n])^{\omega}$. Model `elevator` has such fairness constraints.

The results are shown in Table 4.3 and Table 4.4. Standard CTL model checking (labeled ‘BW’), forward model checking with breadth-first scheduling (labeled ‘FW-BF’), and forward model checking with depth-first scheduling (labeled ‘FW-DF’) were tested on BINGO. Rows in each box show the total number of image computations (including pre-image computations in CTL model checking), CPU time, and the peak number of

Table 4.3: Results for failing properties

Model	VIS	BINGO		
	BW	BW	FW-BF	FW-DF
atm_sw	213 img 5730 sec 26469k	192 img 287 sec 1359k	37 img 8 sec 249k	51 img 7 sec 220k
dh_1	50 img 1 sec 26k	36 img 3 sec 21k	29 img 1 sec 16k	20 img 1 sec 6k
dh_2	(118 img 28807 sec 11992k)	(96 img 10400 sec 6753k)	78 img 139 sec 1277k	31 img 5 sec 99k
dh_3	(> 1000MB)	(> 1000MB)	77 img 1647 sec 5818k	53 img 128 sec 1166k
vpp	45 img 330 sec 11263k	29 img 113 sec 559k	26 img 21 sec 371k	21 img 12 sec 225k
pipe_s1	46 img 5 sec 360k	27 img 12 sec 223k	26 img 6 sec 383k	20 img 1 sec 54k
pipe_d1	(> 1000MB)	(> 1000MB)	24 img 179 sec 5079k	13 img 14 sec 865k
elevator	900 img 28 sec 441k	1018 img 184 sec 167k	173 img 20 sec 309k	17 img 6 sec 114k
ether213	135 img 2527 sec 16519k	98 img 152 sec 1328k	113 img 88 sec 941k	35 img 5 sec 93k

Table 4.4: Results for passing properties

Model	VIS	BINGO		
	BW	BW	FW-BF	FW-DF
`dh_2`	66 img 24915 sec 23451k	64 img 17193 sec 17837k	63 img 88 sec 1281k	176 img 175 sec 2261k
`pipe_s2`	33 img 2 sec 111k	32 img 19 sec 287k	30 img 5 sec 224k	53 img 7 sec 144k
`pipe_d2`	($>$ 1000MB)	($>$ 1000MB)	25 img 100 sec 2409k	56 img 104 sec 2687k
`ether213`	114 img 88 sec 5469k	113 img 138 sec 873k	111 img 76 sec 910k	1114 img 413 sec 961k

BDD nodes. We also experimented with VIS version 1.3 for reference. Execution parameters for VIS were set based on one of its standard script, '`script_model_check.robust`'. All benchmarks were run on 400MHz Pentium II processors under a memory limit of 1000 megabytes. We provided a good initial BDD variable order for each benchmark, and turned off dynamic variable reordering of BINGO and VIS to measure basic performance of the algorithms. For the benchmarks that could not be completed within the memory limit, we also experimented on it with dynamic reordering. Results with dynamic reordering are shown in parenthesis.

Table 4.3 indicates that depth-first scheduling is very efficient in average for failing properties. We obtained one-figure improvements in CPU time and memory usage for large examples. Although the efficiency of depth-first scheduling would depend on the design error, actual design errors have been found effectively in our all examples.

Table 4.4 indicates that depth-first scheduling is not far inferior to the ordinary breadth-first scheduling for passing properties. It means that we can safely use depth-first scheduling even if we cannot expect whether the property fails. Model `ether213` is the worst case for CPU time of depth-first scheduling compared with breadth-first scheduling. It was caused by the difference of the number of image computations.

It is generally recognized that design errors lead the model to unexpected behavior and often make symbolic state traversal more expensive. The breadth-first results of `pipe_d1` and `pipe_d2` show such a situation. `pipe_d1` has a design error and it is more expensive to check than `pipe_d2`. The problem was avoided by depth-first scheduling because the error was found without searching the entire space.

4.5 Chapter summary

We have presented an efficient symbolic model checking algorithm for CTL properties and ω-regular properties. The algorithm is based mainly on forward state traversal, which often gives better performance than backward traversal for actual verification problems. It can be mixed with conventional backward CTL evaluation techniques, and is applicable to arbitrary CTL properties. An ω-regular property is manipulated explicitly as a non-deterministic state transition graph. It separates the property from the implicit state space, in contrast to the conventional algorithm that traverses an implicit state space of the product automata. The explicit property graph enables us to navigate state traversal to the buggy space in order to find bugs in small CPU time and memory. It should become a good framework of incremental or approximate verification, into which valuable ideas for reachability analysis are imported. Strategy of scheduling and partitioning on property node evaluation is the key points for utilizing this method.

CHAPTER 5

Verification Techniques for Applications beyond LSI Design

Graph enumeration and indexing problems are important applications of BDDs and ZDDs, which include enumeration/indexing of paths, cycles, connected components, trees, forests, cut sets, partitioning, cliques, colorings, tilings, and matching. They are tightly related to various real-life verification problems, such as geographic information systems, dependency analysis, and demarcation problems. Each problem is solved implicitly by construction of a monolithic ZDD representing a family of all instances, where each instance (path, cycle, etc.) is represented by a set of graph edges or vertices. Some of them can be constructed efficiently by conventional bottom-up ZDD operations; others are covered by *frontier-based methods*, which construct result ZDDs directly from the root to the terminal nodes [Min13].

Coudert introduced a ZDD based framework for solving graph and set related optimization problems [Cou97]. It includes a bottom-up construction algorithm of the ZDD that represents all maximal cliques of a given graph. Sekine et al. proposed top-down BDD construction algorithms for computing Tutte polynomial and all spanning trees of a given graph [SIT95]. They also showed that the BDD for all spanning trees can be used to obtain a BDD for all forests and a BDD for all paths between two vertices [SI97]. Knuth's introduced a frontier-based method to construct a ZDD representing all paths between two vertices in a top-down way [Knu11, exercise 225 in 7.1.4]. His algorithm is so efficient that a ZDD representing 227449714676812739631826459327989863387613323440 paths on a 15×15 grid graph is constructed in a few minutes. Cycles, Hamiltonian paths, and path matching of a given graph can also be computed by frontier-based methods [Knu11, YSK$^+$12].

In the first half of this chapter, we present new techniques toward an efficient ZDD framework to deal with frontier-based methods. In the second half, we demonstrate the power of frontier-based methods by computing the number of self-avoiding walks connecting opposite corners of a 26×26 square lattice, which is the current world record registered in the On-Line Encyclopedia of Integer Sequences [OEIa].

5.1 ZDD-based enumeration method using recursive specifications

Our approach applies the ZDD node deletion rule on the fly, while conventional methods do not take it into account. We also introduce top-down ZDD construction algorithms for a combination of multiple properties. They do not construct intermediate ZDDs for all the properties, which may blow up and become bottleneck in conventional methods. Although we describe techniques for ZDDs hereafter, many of them are also applicable to BDDs.

5.1.1 Recursive specifications

The basic idea is to define a common interface to ad hoc parts of the algorithms. We define that a *configuration* is a node label used in a top-down construction algorithm, composed of a pair $\langle i, s \rangle$ of node index i ($1 \le i \le n$) and other information s. We assume that $\langle n+1, 0 \rangle$ and $\langle n+1, 1 \rangle$ are pre-defined configurations for the 0- and 1-terminals respectively.

A *recursive specification* of a ZDD is a definition of the following pair of functions:

- ROOT() takes no argument and returns a *root configuration*, or a configuration of the root node;

- CHILD$(\langle i,s\rangle, b)$ takes configuration $\langle i,s\rangle$ of a node and branch $b \in \{0,1\}$ as arguments, and returns a new configuration for the b-child of the node.

A recursive specification can be viewed as a blueprint of a ZDD since it implicitly represents a unique diagram structure in a compact form. Many interesting top-down ZDD construction algorithms, including SIMPATH, can be adapted in this framework.

For example, let us consider a ZDD representing a family of all combinations of k items out of n items where node index i corresponds to the i-th item for $1 \le i \le n$. We define a set of configurations for nonterminal nodes as

$$\{\, \langle i,s\rangle \mid 1 \le i \le n,\ 0 \le s \le k \,\}$$

where i is the item index for the next decision and s is the number of items included before that node. The current item set is rejected immediately when $s > k$, and accepted when $s = k$ and no more undecided item remains. Its recursive specification $Comb_{n,k}$ is defined as follows:

$Comb_{n,k}$.ROOT()
1: **return** $\langle 1,0\rangle$;

$Comb_{n,k}$.CHILD$(\langle i,s\rangle, b)$
1: $s \leftarrow s+b$;
2: **if** $i=n$ **and** $s=k$, **return** $\langle n+1,1\rangle$; // $\boxed{1}$
3: **if** $i=n$ **or** $s>k$, **return** $\langle n+1,0\rangle$; // $\boxed{0}$
4: **return** $\langle i+1,s\rangle$.

Figure 5.1 shows the ZDD specified by $Comb_{5,2}$ before and after reduction. In this case, it is not very difficult to define the recursive specification that directly represents the reduced ZDD structure:

$Comb'_{n,k}$.ROOT()
1: **return** $\langle 1,0\rangle$;

$Comb'_{n,k}$.CHILD$(\langle i,s\rangle, b)$
1: $s \leftarrow s+b$;
2: **if** $s=k$, **return** $\langle n+1,1\rangle$; // $\boxed{1}$

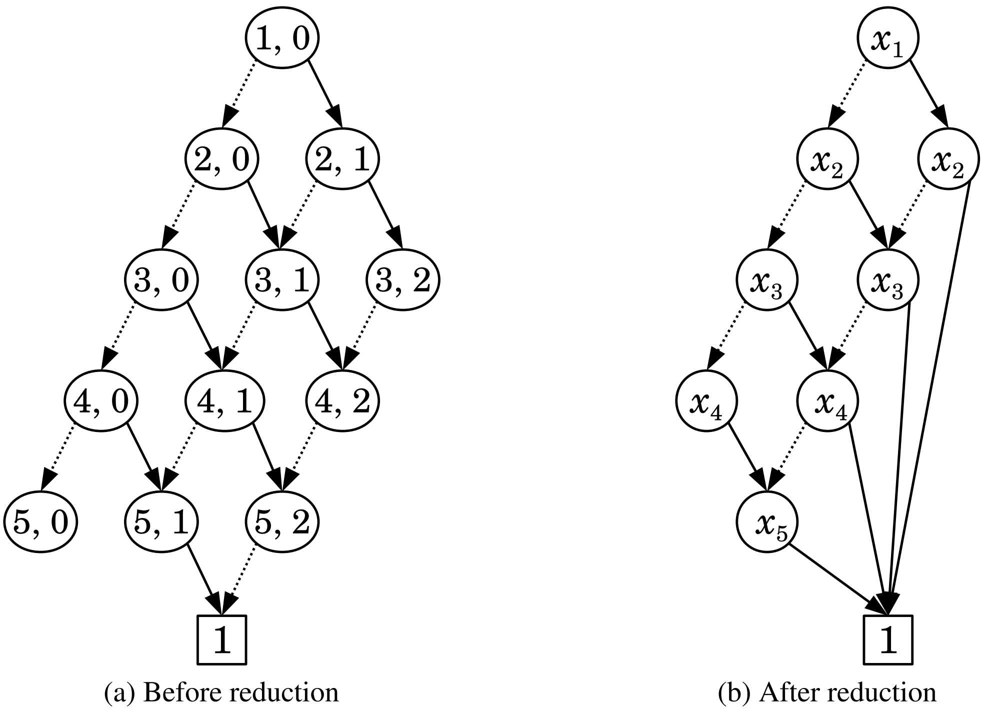


Figure 5.1: ZDD structure for $Comb_{5,2}$

3: **if** $s+n-i<k$, **return** $\langle n+1, 0\rangle$; // $\boxed{0}$
4: **return** $\langle i+1, s\rangle$.

The current item set is rejected as soon as we find that the remaining items are too few to make the k-combination. It is accepted as soon as $s=k$ is satisfied without taking all the remaining items.

5.1.2 General top-down ZDD construction algorithm

Provided that the recursive specifications are given, the top-down ZDD construction can be processed by a common algorithm shown below; let S be a recursive specification and n be the number of its input variables:

CONSTRUCT(S)

1: $\langle i_0, s_0\rangle \leftarrow S$.ROOT();
2: create a new node r and label it as $\langle i_0, s_0\rangle$;
3: **for** $i=i_0$ to n **do**
4:     **for all** node p labeled $\langle i, s\rangle$ for some s **do**
5:         **for all** $b \in \{0, 1\}$ **do**

6: $\langle i', s' \rangle \leftarrow S.$CHILD$(\langle i, s \rangle, b)$;
7: **if** $\langle i', s' \rangle$ corresponds to a terminal node **then**
8: set it to the b-child of p;
9: **else**
10: find or create node p' labeled $\langle i', s' \rangle$;
11: set p' to the b-child of p;
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **return** REDUCE(r).

This algorithm searches all reachable configurations of S from the root to the terminals in a breadth-first manner. Hash tables can be used to ensure one-to-one correspondence between configurations and ZDD nodes. Their entries should be disposed properly because memory size for a configuration might be much larger than that for a ZDD node. Assuming that the hash table operations and evaluations of the recursive specification are constant time operations, this algorithm runs in linear time against the number of reachable configurations.

5.1.3 Parallelizing the construction algorithm

We can parallelize the loop at lines 4–14 of CONSTRUCT based on the fact that the tasks are independent except for the hash table operations at line 10. One can use thread-safe hash table for this purpose; or can divide the hash table into the multiple ones that manage disjoint subsets of possible configurations. The following algorithm makes use of the latter idea:

PARCONSTRUCT(S)

$\langle i_0, s_0 \rangle \leftarrow S.$ROOT$()$;
let d be a dummy node;
insert $\langle s_0, d, 0 \rangle$ to $bucket[i_0][1]$;
for $i = i_0$ to n **do**
for all $k \in \{1, \ldots, m\}$ **do** **// in parallel**

for all $\langle s, \hat{p}, \hat{b} \rangle \in \mathit{bucket}[i][k]$ **do**
find or create node p labeled $\langle i, s \rangle$;
set p to the $\hat{b}$-child of $\hat{p}$;
if p is newly created **then**
for all $b \in \{0, 1\}$ **do**
$\langle i', s' \rangle \leftarrow S.\textsc{Child}(\langle i, s \rangle, b)$;
if $\langle i', s' \rangle$ corresponds to a terminal node **then**
set it to the b-child of p;
else
$k' \leftarrow$ bucket number for $\langle i', s' \rangle$;
insert $\langle s', p, b \rangle$ to $\mathit{bucket}[i'][k']$;
end if
end for
end if
end for
end for
end for
let r be the 0-child of d;
return $\textsc{Reduce}(r)$.

In the above algorithm, configurations are grouped into m buckets; $\mathit{bucket}[i][k]$ works as a task queue for node index $i \in \{1, \ldots, n\}$ and bucket number $k \in \{1, \ldots, m\}$. Since tasks for the same configurations are always stored in the same bucket, different buckets can be processed in parallel without caring about thread-safeness of the hash tables. The number of buckets m should be larger enough than the number of parallel threads for better load balancing, and should be smaller enough than the average number of nodes for each index for less overhead costs. The reduction algorithm in 2.1.1 also can be parallelized in similar ways.

5.1.4 Operations on recursive specifications

Lookahead

In general, any reduced/unreduced ZDD can be represented by a recursive specification; the index might be increased by more than one in the CHILD function when we take the zero-suppress rule aggressively into account. It improves the performance of ZDD construction, while it may worsen simplicity of the recursive specification. It would be pleased if an optimized specification can be generated automatically from an easy-to-understand description for humans.

The *lookahead* operation wraps a given recursive specification and makes the one that represents a smaller and logically equivalent ZDD. It skips redundant configurations of the original specification in terms of the zero-suppress rule.

LOOKAHEAD(S).ROOT()
1: **return** S.ROOT();

LOOKAHEAD(S).CHILD($\langle i,s\rangle, b$)
1: $\langle i',s'\rangle \leftarrow S$.CHILD($\langle i,s\rangle, b$);
2: **while** $i' \leq n$ **and** S.CHILD($\langle i',s'\rangle, 1) = \langle n+1, 0\rangle$ **do**
3: $\quad\langle i',s'\rangle \leftarrow S$.CHILD($\langle i',s'\rangle, 0$);
4: **end while**
5: **return** $\langle i',s'\rangle$.

Figure 5.2 shows the result of SIMPATH for $G_{3,3}$ combined with the lookahead. In comparison with the original result (Figure 2.5), the number of nonterminal nodes is reduced from 52 to 29. This example also shows that the lookahead operation do not always remove all redundant nodes, because they do not care the node sharing and do not backtrack for the node deletion.

Composition

Let us suppose that there are two properties represented by recursive specifications and we want to compute the ZDD that represents the intersection

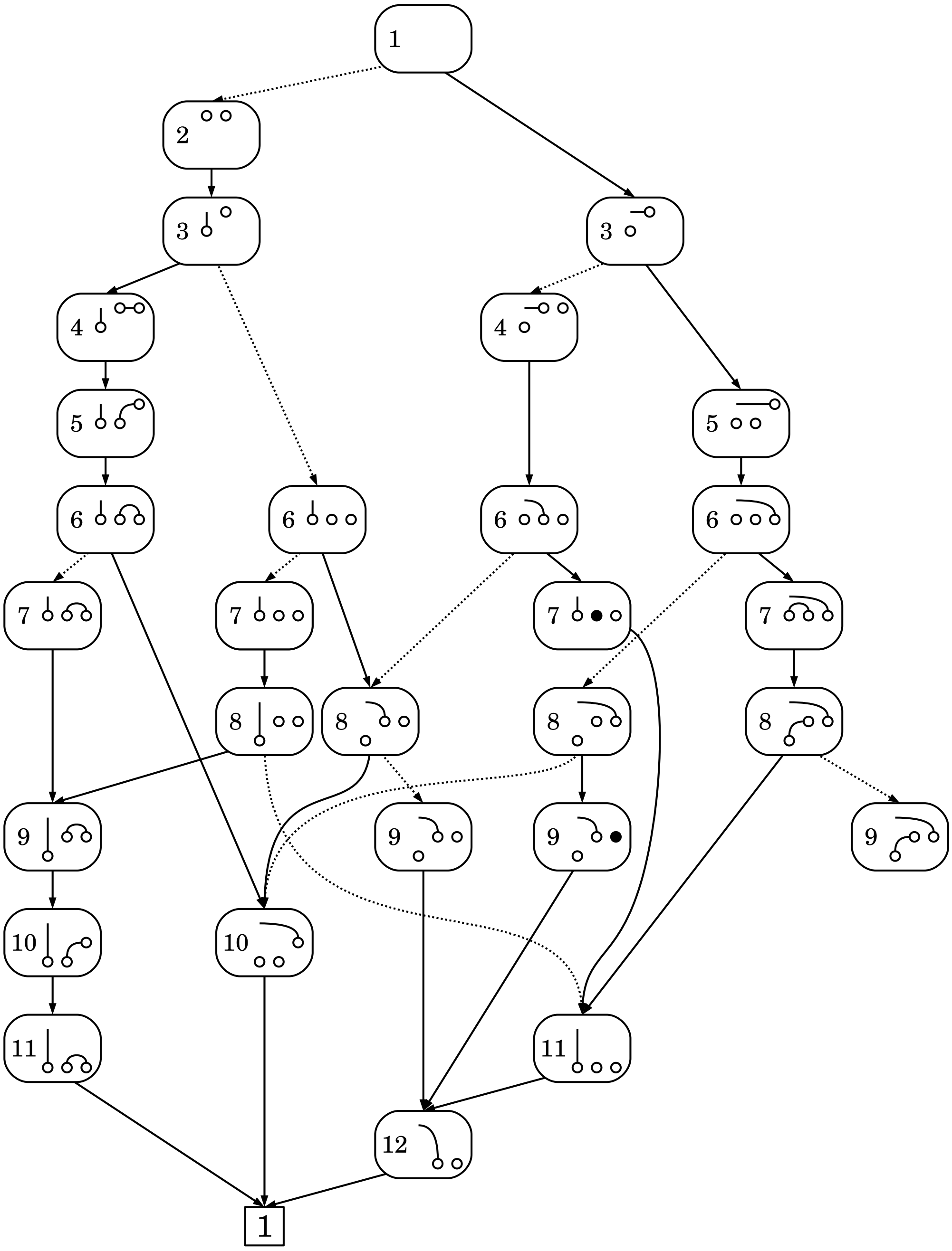


Figure 5.2: ZDD constructed by SIMPATH with the lookahead

of the properties. It is achieved easily by constructing two ZDDs from the specifications and by applying the intersection operation on ZDDs.

In this section, we present an alternative to this approach, in which we first composite the two specifications and then construct a ZDD. It has the advantage of robustness when an intermediate ZDD may blow up while the final ZDD should be compact.

Let S and T be recursive specifications and "$\diamond$" be some binary operator such as "$\vee$" or "$\wedge$". Here we consider a binary operation on S and T, namely $S \diamond T$, such that

$$\textsc{Construct}(S \diamond T) \;=\; \textsc{Construct}(S) \diamond \textsc{Construct}(T)\,.$$

It can be defined as follows:

$(S \diamond T).\textsc{Root}()$

1: $\langle i,s\rangle \leftarrow S.\textsc{Root}()$;

2: $\langle j,t\rangle \leftarrow T.\textsc{Root}()$;

3: **return** $\begin{cases} \langle n+1, s \diamond t\rangle & \text{if } i=j=n+1, \\ \langle \min(i,j), \langle i,s,j,t\rangle\rangle & \text{otherwise.} \end{cases}$

$(S \diamond T).\textsc{Child}(\langle k, \langle i,s,j,t\rangle\rangle\,, b)$ // $k = \min(i,j)$

1: $\langle i,s\rangle \leftarrow \begin{cases} \langle i,s\rangle & \text{if } k<i \text{ and } b=0, \\ \langle n+1, 0\rangle & \text{if } k<i \text{ and } b=1, \\ S.\textsc{Child}(\langle i,s\rangle\,, b) & \text{otherwise;} \end{cases}$

2: $\langle j,t\rangle \leftarrow \begin{cases} \langle j,t\rangle & \text{if } k<j \text{ and } b=0, \\ \langle n+1, 0\rangle & \text{if } k<j \text{ and } b=1, \\ S.\textsc{Child}(\langle j,t\rangle\,, b) & \text{otherwise;} \end{cases}$

3: **return** $\begin{cases} \langle n+1, s \diamond t\rangle & \text{if } i=j=n+1, \\ \langle \min(i,j), \langle i,s,j,t\rangle\rangle & \text{otherwise.} \end{cases}$

In case the operation is set intersection, or logical AND when the ZDDs are interpreted as Boolean functions, we can optimize it by taking more advantage of the zero-suppress rule:

$(S \cap T).\textsc{Root}()$

1: **return** $(S \cap T).\textsc{Sync}(S.\textsc{Root}(), T.\textsc{Root}())$.

$(S \cap T).\textsc{Child}(\langle i, \langle s,t\rangle\rangle\,, b)$

return
$(S \cap T).\textsc{Sync}(S.\textsc{Child}(\langle i,s\rangle, b), T.\textsc{Child}(\langle i,t\rangle, b))$.

$(S \cap T).\textsc{Sync}(\langle i,s\rangle, \langle j,t\rangle)$

1: **while** $i \neq j$ **do**
2:     **if** $i < j$, $\langle i,s\rangle \leftarrow S.\textsc{Child}(\langle i,s\rangle, 0)$;
3:     **if** $j < i$, $\langle j,t\rangle \leftarrow T.\textsc{Child}(\langle j,t\rangle, 0)$;
4: **end while**
5: **if** $i = n+1$, **return** $\langle n+1, s \wedge t\rangle$;
6: **return** $\langle i, \langle s,t\rangle\rangle$.

Lines 1–4 of the SYNC subroutine skips the nodes that would have 1-edges to the 0-terminal. It can be decided easily by checking if configurations derived from S and T have different index numbers. We can go down through 0-edges until the indices are synchronized. It is an interesting property of the combination of intersection operation and zero-suppress rule.

Wrapping and subsetting

We can wrap an existing ZDD structure in a recursive specification. The wrapper of ZDD f is given as follows:

$\textsc{Wrap}(f).\textsc{Root}()$

$i \leftarrow$ the top index of f;
return $\langle i, f\rangle$;

$\textsc{Wrap}(f).\textsc{Child}(\langle i,f\rangle, b)$

$f' \leftarrow$ the b-child of f;
$i' \leftarrow$ the top index of f';
return $\langle i', f'\rangle$.

The same ZDD structure as f can be derived from $\textsc{Wrap}(f)$, that is:

$$\textsc{Construct}(\textsc{Wrap}(f)) = f\,.$$

The wrapping technique extends the usefulness of the operations on recursive specifications. Let us suppose the situation where we have some ZDD f

and want to restrict it by a property represented by specification S. It can be computed as usual by the intersection operation on ZDDs:

$$f \cap \textsc{Construct}(S)\,.$$

Using the wrapping technique, we can also compute it as the intersection on specifications:

$$\textsc{Construct}(\textsc{Wrap}(f) \cap S)\,.$$

We call it *subsetting* technique on top-down ZDD construction, which is an alternative that is worth trying when Construct(S) becomes the bottleneck in the usual method.

5.1.5 Experimental results

Our top-down ZDD construction framework is implemented in C++. We measured single-threaded performance of the algorithms on 2.67GHz Intel Xeon E7-8837 CPU with 1.5TB memory running 64-bit SUSE Linux Enterprise Server 11.

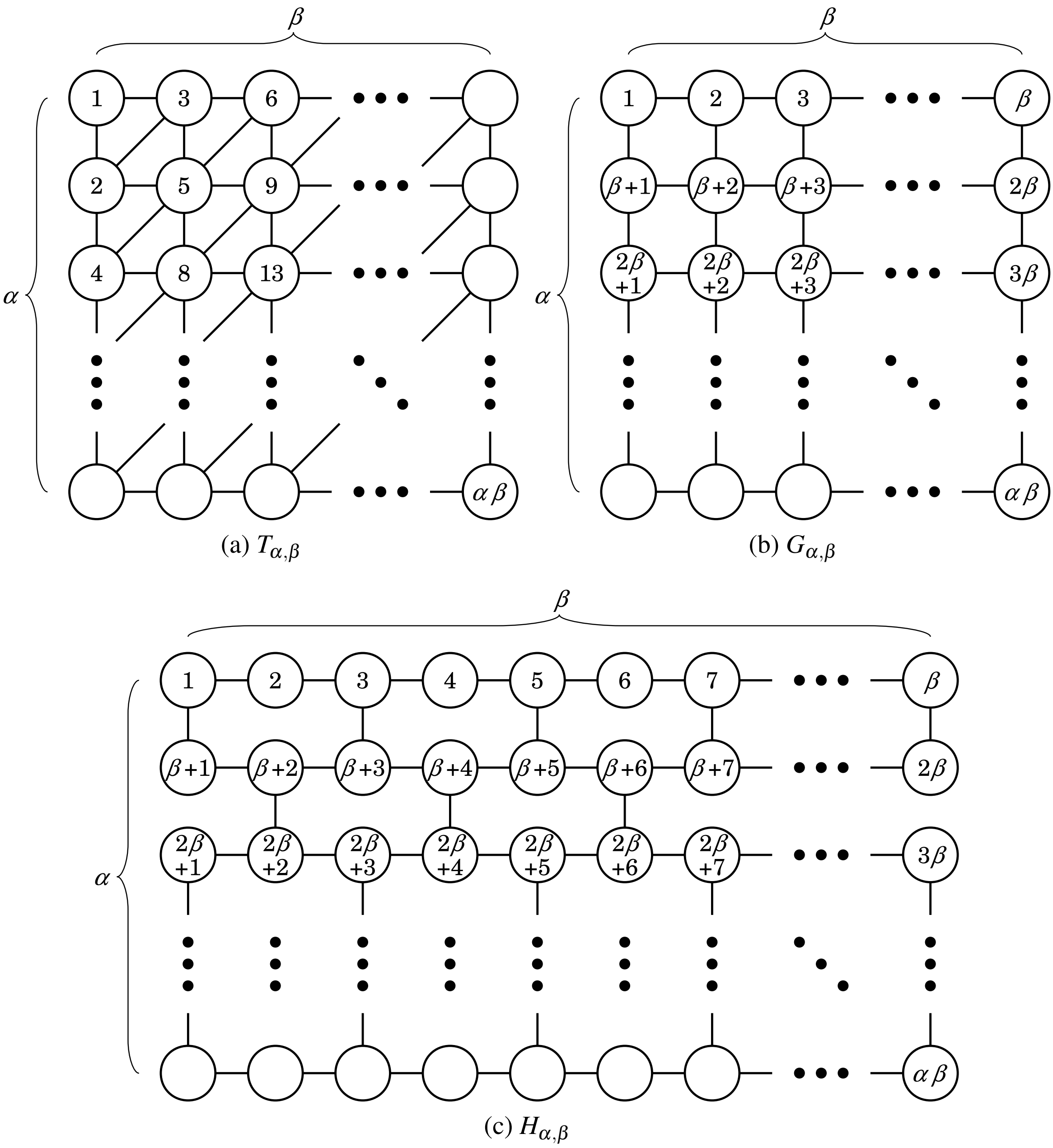
Path enumeration

First, we evaluated the efficiency of our framework in comparison with the original SIMPATH implementation [Knu], and measured improvements achieved by the lookahead and subsetting techniques.

We experimented with graph examples listed in Table 5.1, where m is the number of vertices, n is the number of edges, and #path is the number of paths to be enumerated (paths between v_1 and v_m). We used complete graphs (K_m), triangular grid graphs ($T_{\alpha,\beta}$), square grid graphs ($G_{\alpha,\beta}$), and hexagonal grid graphs ($H_{\alpha,\beta}$) as benchmark examples. The grid graphs and their vertex ordering rules are shown in Figure 5.3. The edge order (ZDD variable order) is defined lexicographically with the vertex order: $\{v_i, v_j\} \leq \{v_{i'}, v_{j'}\}$ if and only if $v_i < v_{i'}$ or $(v_i = v_{i'}$ and $v_j \leq v_{j'})$ where $v_i \leq v_j$ and $v_{i'} \leq v_{j'}$. We have tested some simple vertex ordering rules and have chosen the one that makes final ZDDs compact.

Table 5.1: Characteristics of graph examples

Graph	m	n	#path	SIMPATH (sec.) Main	SIMPATH (sec.) Reduce
K_{17}	17	136	3.55×10^{12}	16.45	18.27
K_{18}	18	153	5.69×10^{13}	57.45	59.70
K_{19}	19	171	9.67×10^{14}	183.98	194.77
K_{20}	20	190	1.74×10^{16}	641.52	662.34
K_{21}	21	210	3.31×10^{17}	2106.68	2344.67
K_{22}	22	231	6.61×10^{18}	7201.05	7939.68
$T_{11,11}$	121	320	4.35×10^{39}	5.58	7.20
$T_{12,12}$	144	385	6.81×10^{47}	27.39	30.48
$T_{13,13}$	169	456	6.33×10^{56}	115.19	124.89
$T_{14,14}$	196	533	3.50×10^{66}	504.95	522.68
$T_{15,15}$	225	616	1.15×10^{77}	2168.47	2260.38
$T_{16,16}$	256	705	2.24×10^{88}	8868.41	9218.46
$G_{13,13}$	169	312	6.45×10^{34}	12.42	14.94
$G_{14,14}$	196	364	6.95×10^{40}	46.23	50.40
$G_{15,15}$	225	420	2.27×10^{47}	152.77	161.76
$G_{16,16}$	256	480	2.27×10^{54}	503.63	534.77
$G_{17,17}$	289	544	6.87×10^{61}	1644.86	1826.86
$G_{18,18}$	324	612	6.34×10^{69}	5598.11	5912.85
$H_{22,23}$	506	726	2.20×10^{61}	10.91	11.90
$H_{24,25}$	600	864	4.90×10^{73}	37.77	39.70
$H_{26,27}$	702	1014	1.50×10^{87}	136.18	129.32
$H_{28,29}$	812	1176	6.28×10^{101}	452.25	434.77
$H_{30,31}$	930	1350	3.61×10^{117}	1531.62	1486.30
$H_{32,33}$	1056	1536	2.85×10^{134}	4935.35	4864.38



(a) $T_{\alpha,\beta}$ (b) $G_{\alpha,\beta}$

(c) $H_{\alpha,\beta}$

Figure 5.3: Grid graphs and their vertex order

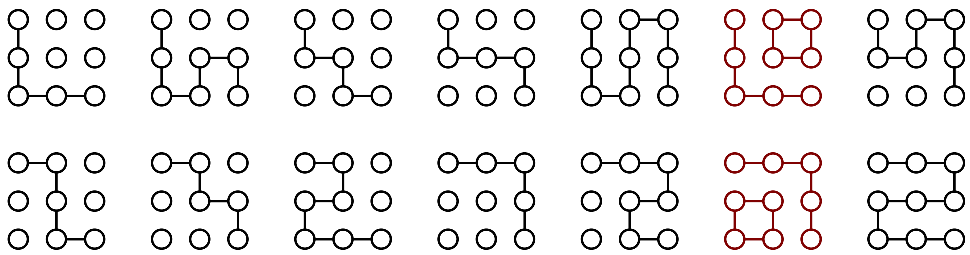


Figure 5.4: Sets of graph edges represented by $Degree(G_{3,3}, 1, 9)$

Table 5.1 also includes CPU time for the original SIMPATH implementation,[1] which is composed of the main program and the ZDD reduction program. The columns "Main" and "Reduce" show their CPU time in seconds. Note that they hand over an unreduced ZDD via text file, while our implementation performs both ZDD construction and reduction on memory.

We wrote the recursive specification $Path(G, v_1, v_m)$ that corresponds to the SIMPATH algorithm. The parameter $G = (V, E)$ is a target graph where $V = \{v_1, \ldots, v_m\}$ is a set of vertices and $E = \{e_1, \ldots, e_n\}$ is a set of edges. We also wrote the recursive specification $Degree(G, v_1, v_m)$ that represents constraints on vertex degrees (number of edges incident to a vertex). If $E' \subseteq E$ forms a path between vertices v_1 and v_m, vertices in graph $G' = (V, E')$ must have a degree of 0 or 2 except that v_1 and v_m must have a degree of 1. That condition is necessary but not sufficient for the set of edges to form a path. For example, Figure 5.4 shows the 14 instances represented by $Degree(G_{3,3}, 1, 9)$, which include the 2 instances that do not actually form paths.

We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), and that with lookahead (2P+L). The four methods are defined as follows:

1P $f \leftarrow \textsc{Construct}(Path(G, v_1, v_m));$

1P+L $f \leftarrow \textsc{Construct}(\textsc{Lookahead}(Path(G, v_1, v_m)));$

2P $g \leftarrow \textsc{Construct}(Degree(G, v_1, v_m));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap Path(G, v_1, v_m));$

[1] We have slightly modified the programs in order to process larger input graphs on a 64-bit machine.

Table 5.2: CPU time for path enumeration (sec.)

Graph	1P	1P+L	2P	2P+L
K_{17}	18.68	13.86	11.06	11.09
K_{18}	62.50	47.69	33.10	33.93
K_{19}	210.82	169.54	103.84	109.43
K_{20}	776.18	586.84	318.73	322.96
K_{21}	2573.39	2083.32	1096.70	1024.71
K_{22}	9683.19	7068.47	3839.62	3856.21
$T_{11,11}$	5.32	3.44	4.66	4.86
$T_{12,12}$	26.78	17.11	21.04	21.27
$T_{13,13}$	115.23	78.26	84.69	85.66
$T_{14,14}$	555.83	340.44	372.78	334.79
$T_{15,15}$	2121.35	1610.42	1355.35	1357.32
$T_{16,16}$	11185.38	6587.02	5499.10	5644.78
$G_{13,13}$	10.53	6.53	6.13	5.71
$G_{14,14}$	41.09	25.31	20.08	18.76
$G_{15,15}$	159.97	91.54	64.80	60.34
$G_{16,16}$	501.28	320.46	211.42	193.67
$G_{17,17}$	1693.46	1067.76	681.77	636.42
$G_{18,18}$	6398.75	3664.18	2296.25	2274.09
$H_{22,23}$	6.07	3.63	3.94	3.47
$H_{24,25}$	32.98	18.29	16.37	14.34
$H_{26,27}$	124.76	70.37	55.55	47.81
$H_{28,29}$	454.18	268.33	183.16	165.88
$H_{30,31}$	1542.16	953.21	610.25	571.67
$H_{32,33}$	5680.29	3275.21	2078.65	1934.34

2P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Degree}(G, v_1, v_m)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Path}(G, v_1, v_m))).$

Table 5.2 shows CPU time of the four methods. In comparison with the original SIMPATH implementation (Table 5.1), our implementation 1P looks reasonably fast. It shows that there is no noticeable overhead in our top-down construction framework. The lookahead and subsetting techniques accelerated top-down ZDD construction by a factor of 1.1 to 2.9. Figure 5.5

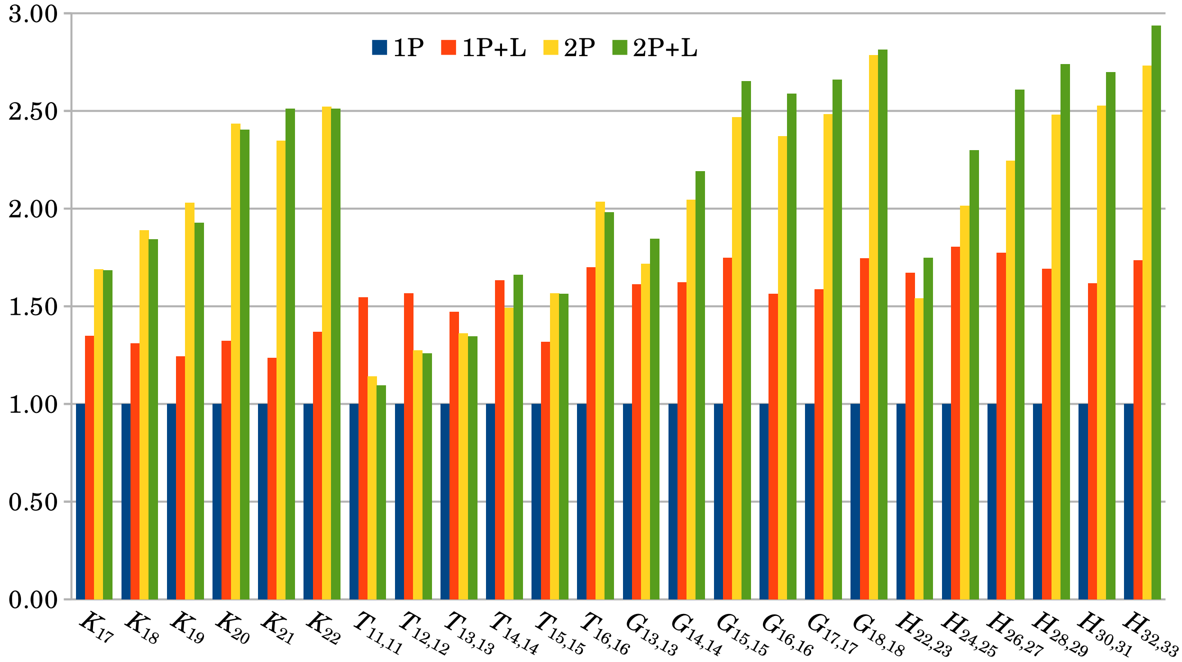


Figure 5.5: Speed ratio for path enumeration

summarizes computation speed of methods 1P+L, 2P, and 2P+L relative to the basic method 1P. While the fastest method is dependent on the example, we can read that the subsetting technique is relatively effective in large examples.

Construction of ZDD g dominates CPU time in 2P and 2P+L because g is much smaller than f. Table 5.3 compares ZDD size (the number of nonterminal nodes) of g and f. The larger the graph is, the larger the size difference between g and f becomes. "Peak size" is the ZDD size just before the reduction phase, which indicates the number of iterations run in both construction and reduction phases. The lookahead and subsetting techniques have effect to reduce 40 to 50 percent of the peak size. It is interesting that methods 1P+L and 2P does not show much difference in the peak size of f, even though the node deletion rule is not checked explicitly in 2P. It means that the zero-suppress information was effectively inherited from g in the subsetting method.

The total memory usage in megabytes is shown in Table 5.4. It is confirmed that the lookahead and subsetting techniques are also effective to reduce memory usage.

Table 5.3: ZDD size in path enumeration

	g			f				
	Peak size		Final size	Peak size				Final size
Graph	2P	2P+L		1P	1P+L	2P	2P+L	
K_{17}	3,383,182	1,446,371	1,415,798	31,687,586	16,776,941	17,913,664	16,685,440	15,469,186
K_{18}	7,803,921	3,331,561	3,266,139	98,595,128	52,894,116	56,302,697	52,644,447	48,935,273
K_{19}	17,916,021	7,638,950	7,498,891	309,329,033	168,011,957	178,326,054	167,320,943	155,922,881
K_{20}	40,960,634	17,445,357	17,144,913	978,702,177	537,803,391	569,319,129	535,863,645	500,559,700
K_{21}	93,303,788	39,699,692	39,055,059	3,122,714,316	1,734,809,580	1,832,020,786	1,729,287,871	1,619,050,484
K_{22}	211,843,795	90,058,256	88,674,234	10,047,097,379	5,639,089,096	5,941,791,248	5,623,154,813	5,276,150,643
$T_{11,11}$	3,454,359	1,678,273	1,534,383	13,365,043	6,909,231	7,648,691	6,909,231	6,432,417
$T_{12,12}$	11,445,656	5,558,016	5,075,298	53,816,252	27,977,245	30,916,232	27,977,245	26,076,799
$T_{13,13}$	37,584,679	18,243,715	16,642,341	215,875,876	112,786,898	124,439,778	112,786,898	105,238,888
$T_{14,14}$	122,497,140	59,440,240	54,176,364	863,508,297	453,159,395	499,288,002	453,159,395	423,254,393
$T_{15,15}$	396,720,695	192,448,119	175,277,115	3,446,706,536	1,816,007,604	1,998,415,097	1,816,007,604	1,697,726,218
$T_{16,16}$	1,277,849,872	619,726,690	564,075,414	13,735,340,349	7,262,868,868	7,983,662,545	7,262,868,868	6,795,583,172
$G_{13,13}$	1,952,762	983,037	971,773	26,894,640	15,032,057	16,178,631	15,032,057	13,803,430
$G_{14,14}$	4,599,802	2,314,237	2,289,661	86,698,791	48,641,299	52,307,691	48,641,299	44,871,856
$G_{15,15}$	10,702,842	5,382,141	5,328,893	277,581,568	156,253,978	167,908,407	156,253,978	144,759,636
$G_{16,16}$	24,641,530	12,386,301	12,271,613	883,640,711	498,888,415	535,749,877	498,888,415	464,004,180
$G_{17,17}$	56,213,498	28,246,013	28,000,253	2,799,256,918	1,584,605,112	1,700,699,101	1,584,605,112	1,479,128,501
$G_{18,18}$	127,205,370	63,897,597	63,373,309	8,830,604,856	5,010,748,938	5,375,051,545	5,010,748,938	4,692,765,814
$H_{22,23}$	1,895,414	1,004,539	897,019	20,985,221	12,431,317	13,117,268	12,431,317	10,686,910
$H_{24,25}$	4,548,598	2,410,491	2,151,419	68,690,969	40,853,448	43,081,787	40,853,448	35,229,328
$H_{26,27}$	10,751,990	5,697,531	5,083,131	222,730,862	132,929,717	140,105,957	132,929,717	114,956,610
$H_{28,29}$	25,092,086	13,295,611	11,857,915	716,615,275	429,006,718	451,953,721	429,006,718	371,973,561
$H_{30,31}$	57,917,430	30,687,227	27,361,275	2,290,741,210	1,375,126,756	1,448,070,796	1,375,126,756	1,195,179,926
$H_{32,33}$	132,415,478	70,156,283	62,537,723	7,282,606,658	4,382,454,784	4,613,178,936	4,382,454,784	3,817,373,513

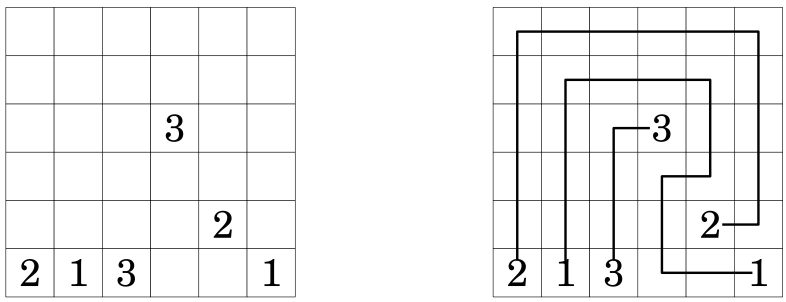


Figure 5.6: Numberlink problem and its solution

Numberlink and Slitherlink puzzles

Secondly, we have improved the ZDD-based solvers of Numberlink and Slitherlink introduced in [YSK+12] using the lookahead and subsetting techniques. Examples of Numberlink and Slitherlink are shown in Figure 5.6 and Figure 5.7 respectively. They are logic puzzles that involve finding the paths or the cycle that satisfy given local and global properties [Nikc]. These experiments show case studies of designing efficient ZDD construction procedures on our framework.

Table 5.4: Memory usage for path enumeration (MB)

Graph	1P	1P+L	2P	2P+L
K_{17}	891	806	986	951
K_{18}	2,701	2,717	3,447	3,187
K_{19}	8,204	8,293	10,831	10,250
K_{20}	26,173	26,275	33,392	32,031
K_{21}	82,395	84,297	112,346	108,139
K_{22}	254,359	272,999	367,226	357,666
$T_{11,11}$	327	194	249	231
$T_{12,12}$	1,273	713	905	820
$T_{13,13}$	5,059	2,770	3,616	3,121
$T_{14,14}$	20,075	10,855	13,202	12,086
$T_{15,15}$	79,987	43,862	53,101	49,570
$T_{16,16}$	318,664	184,504	224,682	205,607
$G_{13,13}$	786	402	412	388
$G_{14,14}$	2,277	1,212	1,287	1,202
$G_{15,15}$	6,532	3,758	4,079	3,802
$G_{16,16}$	20,721	11,927	13,017	12,125
$G_{17,17}$	65,952	37,686	41,235	38,419
$G_{18,18}$	206,430	118,760	129,699	120,904
$H_{22,23}$	505	322	333	319
$H_{24,25}$	1,652	990	1,045	986
$H_{26,27}$	5,464	3,155	3,318	3,151
$H_{28,29}$	16,639	10,422	10,640	10,081
$H_{30,31}$	53,028	32,016	33,912	32,215
$H_{32,33}$	168,351	101,819	107,812	102,273

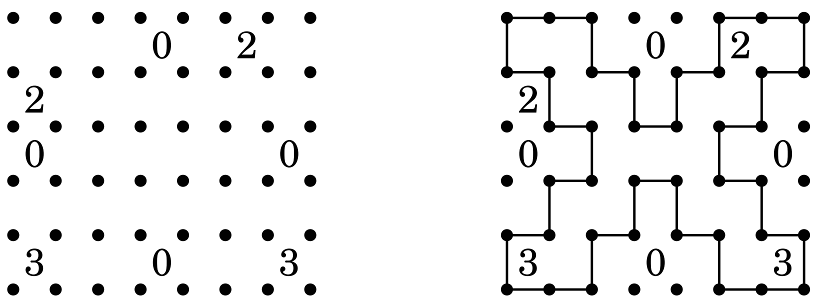


Figure 5.7: Slitherlink problem and its solution

Table 5.5: Characteristics of Numberlink problems

Name	Graph	m	n	Description
BN64	$G_{10,10}$	100	180	64th problem in [Nik89]
BN79	$G_{10,10}$	100	180	79th problem in [Nik89]
BN85	$G_{20,15}$	300	565	85th problem in [Nik89]
BN99	$G_{20,15}$	300	565	99th problem in [Nik89]
C108	$G_{36,20}$	720	1384	Vol. 108 in [Nika]

Numberlink solver Numberlink is played on a grid with the following rules.

1. Connect pairs of the same hint numbers with a continuous line.
2. Lines go through the center of the cells, horizontally, vertically, or changing direction, and never twice through the same cell.
3. Lines cannot cross, branch off, or go through the cells with hint numbers.
4. Lines must cover all the cells.

It can be viewed as a problem of finding a path matching on grid graph G under hint h where each cell corresponds to a vertex of G. It can be solved by an algorithm derived from SIMPATH [YSK+12]. The output is the ZDD that represents the set of all solutions, which must be a singleton if the problem is designed correctly.

We have experimented on Numberlink problems listed in Table 5.5. In the same way as experiments in the previous section, we wrote the main algorithm as recursive specification $\mathit{Numlin}(G,h)$, which gives the exact solutions, and also wrote the constraints on vertex degrees (1 for vertices with hint numbers and 2 for others) as another recursive specification $\mathit{Degree}(G,h)$, which gives a superset of the solutions. We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), and that with lookahead (2P+L). The four methods are defined as follows:

1P $f \leftarrow \textsc{Construct}(\mathit{Numlin}(G,h))$;

1P+L $f \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Numlin}(G,h)))$;

Table 5.6: CPU time for solving Numberlink puzzles (sec.)

Name	1P	1P+L	2P	2P+L
BN64	0.02	0.01	0.02	0.02
BN79	0.03	0.02	0.03	0.03
BN85	72.69	42.19	32.93	27.14
BN99	79.95	42.47	38.72	28.97
C108	10092.48	5546.26	4653.71	3309.26

Table 5.7: Memory usage for solving Numberlink puzzles (MB)

Name	1P	1P+L	2P	2P+L
BN64	4	3	10	9
BN79	9	6	20	14
BN85	8,010	4,741	6,118	4,561
BN99	8,907	5,073	6,965	5,013
C108	817,969	467,968	644,967	474,646

2P $g \leftarrow \textsc{Construct}(\mathit{Degree}(G,h));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Numlin}(G,h));$

2P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Degree}(G,h)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Numlin}(G,h))).$

In the methods 2P and 2P+L, g is used as a search space for the solutions.

Table 5.6 and Table 5.7 show CPU time in seconds and memory usage in megabytes respectively. The lookahead and subsetting techniques were effective for the Numberlink solvers to reduce CPU time and memory usage. Method 2P+L was the fastest and was about three times as fast as the basic method 1L.

Slitherlink solver Slitherlink is played on a grid of dots with the following rules.

1. Connect adjacent dots with vertical or horizontal lines.

2. A single loop is formed with no crossing or branch.

Table 5.8: Characteristics of Slitherlink problems

Name	Graph	m	n	Description
BS68	$G_{25,15}$	375	710	68th problem in [Nik92]
BS77	$G_{25,15}$	375	710	77th problem in [Nik92]
BS89	$G_{37,21}$	777	1496	89th problem in [Nik92]
BS96	$G_{37,21}$	777	1496	96th problem in [Nik92]
S10	$G_{37,21}$	777	1496	10th problem in [Nikb]
C95	$G_{46,32}$	1472	2866	Vol. 95 in [Nika]
C103	$G_{46,32}$	1472	2866	Vol. 103 in [Nika]
C113	$G_{46,32}$	1472	2866	Vol. 113 in [Nika]

3. Each hint cell indicates the number of lines surrounding it, while empty cells may be surrounded by any number of lines.

It can be viewed as a problem of finding a cycle on grid graph G under hint h where each dot corresponds to a vertex of G. It can be solved by an algorithm also derived from SIMPATH [YSK+12]. The output is the ZDD that represents the set of all solutions, which must be a singleton if the problem is designed correctly.

We have experimented on Slitherlink problems listed in Table 5.8. We made three recursive specifications: $\mathit{Cycle}(G)$ for enumerating all cycles in G, $\mathit{Hint}(G,h)$ for the constraints defined by the hints, and $\mathit{Degree}(G)$ for the constraints on vertex degrees (0 or 2 for all vertices). Intersection of $\mathit{Cycle}(G)$ and $\mathit{Hint}(G,h)$ gives the solution, while $\mathit{Degree}(G)$ is expected to be an extra guide to get the solution. ZDD for $A_{\mathit{Cycle}}(G)$ could not be constructed because G is fairly large in the Slitherlink problems. We compared a basic one-pass method (1P), that with lookahead (1P+L), a two-pass subsetting method (2P), that with lookahead (2P+L), a three-pass subsetting method (3P), and that with lookahead (3P+L), using $\mathit{Hint}(G,h)$ as the start points of the subsetting methods. The six methods are defined as follows:

1P $\quad f \leftarrow \textsc{Construct}(\mathit{Hint}(G,h) \cap \mathit{Cycle}(G));$

1P+L $\quad f \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h) \cap \mathit{Cycle}(G)));$

Table 5.9: CPU time for solving Slitherlink puzzles (sec.)

Name	1P	1P+L	2P	2P+L	3P	3P+L
BS68	0.02	0.02	0.02	0.02	0.02	0.02
BS77	0.04	0.04	0.02	0.02	0.02	0.01
BS89	1.92	1.11	0.85	0.52	1.79	1.16
BS96	6.60	3.43	2.34	1.43	12.21	7.59
S10	25.92	12.75	7.94	4.19	10.15	6.06
C95	12095.40	5209.64	2394.51	1284.40	1304.61	740.11
C103	6456.14	2400.81	1434.15	808.77	7830.46	4385.51
C113	713.80	341.19	285.49	155.98	393.05	243.67

Table 5.10: Memory usage for solving Slitherlink puzzles (MB)

Name	1P	1P+L	2P	2P+L	3P	3P+L
BS68	7	7	17	12	19	14
BS77	12	8	24	17	18	13
BS89	340	173	430	301	689	540
BS96	817	426	761	552	2,517	1,570
S10	2,247	1,113	1,382	936	2,025	1,273
C95	624,616	318,325	211,630	124,468	149,335	89,381
C103	364,887	178,929	139,410	84,036	739,841	441,777
C113	44,478	21,776	32,714	19,231	56,104	33,072

2P $g \leftarrow \textsc{Construct}(\mathit{Hint}(G,h));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Cycle}(G));$

2P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Cycle}(G)));$

3P $g \leftarrow \textsc{Construct}(\mathit{Hint}(G,h));$
$g' \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \mathit{Degree}(G));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g') \cap \mathit{Cycle}(G));$

3P+L $g \leftarrow \textsc{Construct}(\textsc{Lookahead}(\mathit{Hint}(G,h)));$
$g' \leftarrow \textsc{Construct}(\textsc{Wrap}(g) \cap \textsc{Lookahead}(\mathit{Degree}(G)));$
$f \leftarrow \textsc{Construct}(\textsc{Wrap}(g') \cap \textsc{Lookahead}(\mathit{Cycle}(G)));$

Table 5.9 and Table 5.10 describe CPU time in seconds and memory usage in megabytes respectively. Method 2P+L was the fastest for most examples, while 3P+L was very efficient for C95. The results show that the lookahead and subsetting techniques were effective also in Slitherlink solvers and that it is not easy to find the best subsetting strategy before trial.

5.2 Fast computation of self-avoiding walks crossing a square lattice

A *path* in a graph is a way to go from a vertex to another vertex without visiting any vertex twice. It is also referred to as a *self-avoiding walk (SAW)*, which is known to be introduced by the chemist Flory as a model of polymer chains [Flo49]. In spite of its simple definition, many difficult mathematical problems are hidden behind SAWs [MS93, Wei]. They include a problem of counting the number of SAWs from $(0,0)$ to (n,n) in a grid graph, which has become popular in Japan through a YouTube-animation demonstrating importance of algorithms against combinatorial explosion [Dea]. The answer is known to grow as $\lambda^{n^2+o(n^2)}$ and $\lambda \simeq 1.7$ [BMGJ05]. When $n = 10$, it is about 10^{24} and cannot be counted one by one in a realistic time even if we could find trillions of paths in a second.

According to the On-Line Encyclopedia of Integer Sequences (OEIS) A007764 [OEIa], Rosendale computed the answers up to $n = 11$ in 1981 and Knuth computed the answer to $n = 12$ in 1995. Bousquet-Mélou et al. presented the answers up to $n = 19$ in their paper [BMGJ05] in 2005, using an algorithm based on the transfer-matrix method of Conway et al. [CEG93], which makes good use of the fact that the target is a grid graph. On the other hand, Knuth introduced a new algorithm for general graphs called SIMPATH in 2008 [Knu][Knu11, exercise 225 in 7.1.4], which constructs a zero-suppressed binary decision diagram (ZDD) [Min93] representing a set of all paths between two vertices in a graph. We have extended the answers up to $n = 21$ in 2012 by reimplementing SIMPATH to directly count the number of paths instead of building a diagram. The record has been extended recently to $n = 24$ by Spaans, using an efficient and parallel implementation of the transfer-matrix

Algorithm 1 Computation of the number of paths in a graph

```
1: let count be an empty hash table and mate_r represent an empty mapping;
2: count[mate_r] ← 1;
3: for i = 1 to n do
4:    let tmp be an empty hash table;
5:    for all keys mate in count do
6:       if exclusion of e_i makes no unwanted endpoint then
7:          mate_0 ← the next state of mate when e_i is excluded;
8:          tmp[mate_0] ← tmp[mate_0] + count[mate];
9:       end if
10:      if inclusion of e_i makes no unwanted endpoint, cycle, or branch then
11:         mate_1 ← the next state of mate when e_i is included;
12:         tmp[mate_1] ← tmp[mate_1] + count[mate];
13:      end if
14:   end for
15:   count ← tmp;
16: end for
17: let mate_t be a mapping with a single entry mate_t[v_m] = v_1;
18: return count[mate_t].
```

method.

We introduce a SIMPATH-based algorithm optimized for grid graphs. On the premise that the graph is a square grid, we can use a simple array with a minimal perfect hash function instead of an ordinary hash table. We improve the performance still more by various techniques such as in-place update and parallel processing.

5.2.1 Algorithm for general graphs

Knuth's SIMPATH algorithm introduced in 2.1.3 constructs a ZDD structure for a set of paths in a graph. If we just want to know the number of paths, it can be computed without constructing the ZDD structure as shown in Algorithm 1. We use two hash tables from mates to integers, named *count* and *tmp*. Each entry $count[mate]$ represents the number of cases reaching *mate*. In lines 8 and 12 of Algorithm 1, we assume $tmp[mate] = 0$ when *mate* is not defined as a key of *tmp*. At the end of the algorithm, *count* will have a single entry, which maps the terminal state to the answer. Memory usage of this algorithm is dominated by the size of hash tables. It depends on the number of unique mates appearing at each level, which cannot be foretold unless we know characteristics of the input graph.

5.2.2 Basic idea for grid graphs

Let $v_{(i,j)}$ be the vertex at row i and column j of an $n \times n$ grid graph ($1 \leq i \leq n$, $1 \leq j \leq n$) and we compute the number of paths between $v_{(1,1)}$ and $v_{(n,n)}$. We visit the vertices in the order of $v_{(1,1)}, \ldots, v_{(1,n)}, v_{(2,1)}, \ldots, v_{(2,n)}, \ldots$ and make decisions on vertical line $\{v_{(i-1,j)}, v_{(i,j)}\}$ and horizontal line $\{v_{(i,j-1)}, v_{(i,j)}\}$. The frontier at the step visiting $v_{(i,j)}$ is $\{v_{(i,1)}, \ldots, v_{(i,j)}, v_{(i-1,j+1)}, \ldots, v_{(i-1,n)}\}$.

The algorithm for general graphs keeps vertex identifiers of path endpoints in a mate while we can compress the information efficiently on the premise that the graph is embedded in a plane. Pairs of path endpoints always form nested structure on the frontier because no path fragment can intersect. We do not need to record all vertex identifiers of endpoint pairs but record only whether they are left or right endpoints. Let $s = c_1 c_2 \cdots c_n$ be a string, called *frontier state*, at a step visiting $v_{(i,j)}$ where $\Sigma = \{\boxed{(}, \boxed{)}, \boxed{\circ}, \boxed{\bullet}\}$ and $c_k \in \Sigma$ is a character representing a state of the k-th vertex in the frontier:

$$c_k = \begin{cases} \boxed{(} & \text{if the } k\text{-th vertex is a left endpoint;} \\ \boxed{)} & \text{if the } k\text{-th vertex is a right endpoint;} \\ \boxed{\bullet} & \text{if } k = j \text{ and the } k\text{-th vertex is an intermediate point;} \\ \boxed{\circ} & \text{otherwise.} \end{cases}$$

We consider the endpoint connected to $v_{(1,1)}$ is always $\boxed{)}$, a right endpoint . $\boxed{\bullet}$ is a special character only for c_j, meaning that it is different from $\boxed{\circ}$ as we cannot include the next horizontal line $\{v_{(i,j)}, v_{(i,j+1)}\}$.

Examples of state transitions are shown in Figure 5.8. We have no alternatives of vertical line selections in every step, because $\{v_{(i-1,j)}, v_{(i,j)}\}$ must be excluded if $v_{(i-1,j)}$ is not an endpoint and it must be included if $v_{(i-1,j)}$ is an endpoint. Branches are always made for horizontal lines; a frontier state changes only when $\{v_{(i,j-1)}, v_{(i,j)}\}$ is included except for $\boxed{\bullet}$ to $\boxed{\circ}$ changes. The state transition is summarized in Figure 5.9.

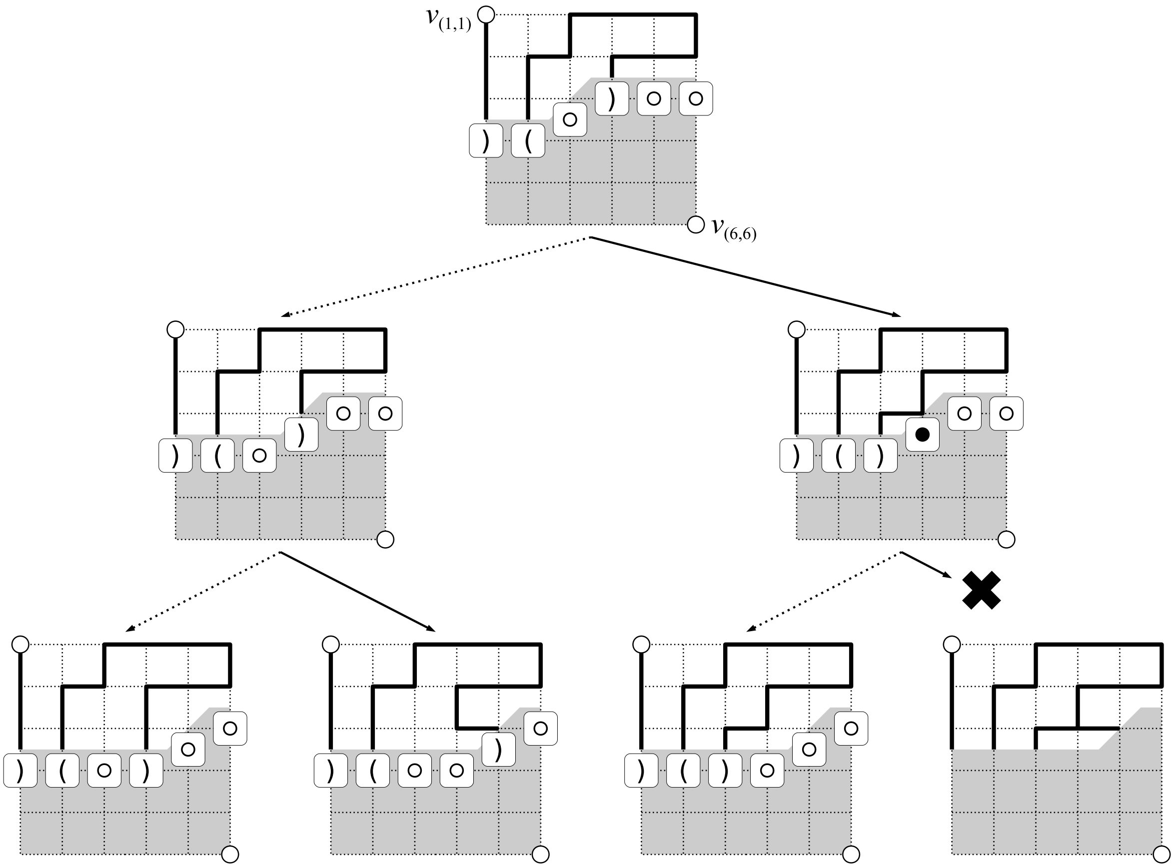
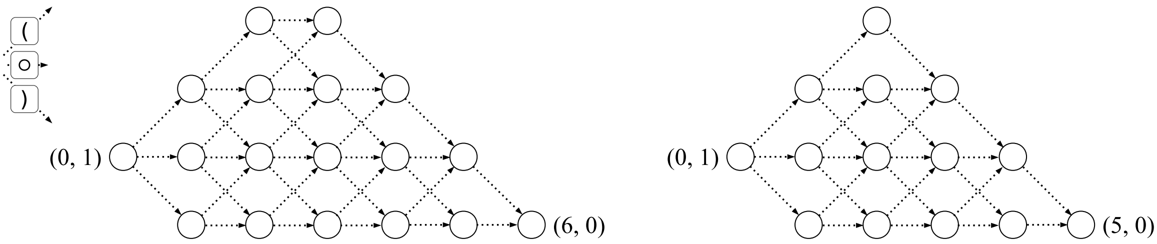


Figure 5.8: Examples of state transitions on a 6×6 grid graph

Current state	Next state: Line excluded	Next state: Line included
⋯ ○ ○ ⋯	⋯ ○ ○ ⋯	⋯ ( ) ⋯
⋯ ○) ⋯	⋯ ○) ⋯	⋯) ● ⋯
⋯ ○ (⋯	⋯ ○ (⋯	⋯ (● ⋯
⋯) ○ ⋯	⋯) ○ ⋯	⋯ ○) ⋯
⋯ (⋯)) ⋯	⋯ (⋯)) ⋯	⋯) ⋯ ○ ● ⋯
⋯) (⋯	⋯) (⋯	⋯ ○ ● ⋯
⋯ (○ ⋯	⋯ (○ ⋯	⋯ ○ (⋯
⋯ ((⋯) ⋯	⋯ ((⋯) ⋯	⋯ ○ ● ⋯ (⋯
⋯ ( ) ⋯	⋯ ( ) ⋯	N/A (cycle)
⋯ ● ○ ⋯	⋯ ○ ○ ⋯	N/A (branch)
⋯ ●) ⋯	⋯ ○) ⋯	N/A (branch)
⋯ ● (⋯	⋯ ○ (⋯	N/A (branch)

Figure 5.9: State transition table



(a) Main states: $\mathscr{M}_{(0,1)\to(6,0)}$ (b) Blocked states: $\mathscr{M}_{(0,1)\to(5,0)}$

Figure 5.10: Possible frontier states for a 6×6 grid graph

Since $\boxed{(}\,\boxed{)}$ pairs are nested, the number of valid frontier states is closely related to Motzkin numbers [DS77]. The n-th Motzkin number is given by $M_n = \left|\mathscr{M}_{(0,0)\to(n,0)}\right|$ where $\mathscr{M}_{(x_1,y_1)\to(x_2,y_2)}$ represents a set of routes from coordinates (x_1,y_1) to coordinates (x_2,y_2) on $x_2 - x_1$ steps of moves $(1,1)$, $(1,-1)$, or $(1,0)$ without visiting negative y-coordinates. They are given by recurrence relation:

$$M_0 = M_1 = 1\;;\quad M_n = \frac{3(n-1)M_{n-2} + (2n+1)M_{n-1}}{n+2}\;. \tag{5.1}$$

We divide frontier states into two classes: *main states* and *blocked states*. A main state does not have $\boxed{\bullet}$ and a blocked state has $\boxed{\bullet}$ at the j-th position. Let $\Sigma' = \{\boxed{(},\boxed{)},\boxed{\circ}\}$and $S' \subset \Sigma'^n$ be a set of main states. S' corresponds to $\mathscr{M}_{(0,1)\to(n,0)}$ (Figure 5.10a) where characters $\boxed{(}$, $\boxed{)}$, and $\boxed{\circ}$ are associated with moves $(1,1)$, $(1,-1)$, and $(1,0)$ respectively. The number of main states is given by:

$$N_n \;=\; \left|\mathscr{M}_{(0,1)\to(n,0)}\right| \;=\; M_{n+1} - M_n\;. \tag{5.2}$$

Let $S''_j \subset \Sigma'^{j-1} \times \{\boxed{\bullet}\} \times \Sigma'^{n-j}$ be a set of blocked states after visiting j-th column. As all blocked states have $\boxed{\bullet}$ at the j-th position, it can be ignored in enumerating S''_j. That corresponds to $\mathscr{M}_{(0,1)\to(n-1,0)}$ (Figure 5.10b) and the number of blocked states is given by N_{n-1}. Therefore, the total number of frontier states is:

$$N_n + N_{n-1} \;=\; M_{n+1} - M_{n-1}\;. \tag{5.3}$$

Since the domain of frontier states has become clear, we can define a minimal perfect hash function $\varphi_j : S_j \to \{1,\ldots,N_n + N_{n-1}\}$ for each column position $1 \le j \le n$. It allows us to use a simple array instead of an ordinary hash table to keep intermediate results. A simple implementation of φ_j would

Algorithm 2 Computation of the number of paths in a grid graph

$count[k] \leftarrow 0$ for all $1 \le k \le N_n + N_{n-1}$;
$count[\varphi_1(\overbrace{\boxed{)}\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}}^{n})] \leftarrow 1$;
for $i = 1$ **to** n **do**
  for $j = 1$ **to** $n-1$ **do**
    $tmp[k] \leftarrow 0$ for all $1 \le k \le N_n + N_{n-1}$;
    for all $s \in S_j$ **do**
      $t \leftarrow$ the next state of s when $\{v_{(i,j)}, v_{(i,j+1)}\}$ is excluded;
      $tmp[\varphi_{j+1}(t)] \leftarrow tmp[\varphi_{j+1}(t)] + count[\varphi_j(s)]$;
      $u \leftarrow$ the next state of s when $\{v_{(i,j)}, v_{(i,j+1)}\}$ is included;
      if u is defined **then**
        $tmp[\varphi_{j+1}(u)] \leftarrow tmp[\varphi_{j+1}(u)] + count[\varphi_j(s)]$;
      end if
    end for
    $count[k] \leftarrow tmp[k]$ for all $1 \le k \le N_n + N_{n-1}$;
  end for
  $tmp[k] \leftarrow 0$ for all $1 \le k \le N_n + N_{n-1}$;
  for all $s \in S_n$ **do**
    $t \leftarrow$ the string made from s by replacing its $\boxed{\bullet}$ with $\boxed{\circ}$;
    $tmp[\varphi_1(t)] \leftarrow tmp[\varphi_1(t)] + count[\varphi_n(s)]$;
  end for
  $count[k] \leftarrow tmp[k]$ for all $1 \le k \le N_n + N_{n-1}$;
end for
return $count[\varphi_1(\overbrace{\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}\boxed{)}}^{n})]$;

scan characters in a given state string one by one and would calculate the serial number using some function or lookup table. Our improved implementation is described later in 5.2.3.

Algorithm 2 shows the basic method for computing the number of paths in a grid graph. The *count* array keeps the integers that represent the numbers of cases for all frontier states at the current step. For example, if we apply this algorithm to a 24×24 grid graph using 56-byte integers, *count* requires $(M_{25} - M_{23}) \times 56$ bytes $=$ 413 gigabytes of memory. The *tmp* array is a temporary storage, which must be the same size as *count*.

It is not difficult to modify Algorithm 2 for computing the number of cycles. The numbers of main and blocked states for cycle enumeration are

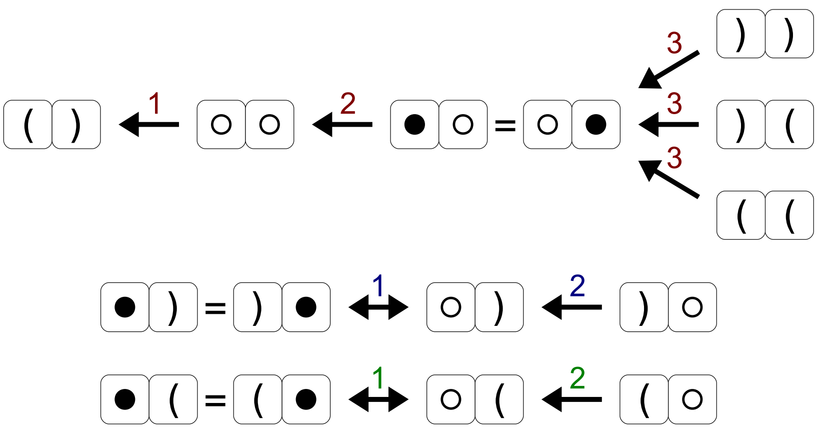


Figure 5.11: Data dependency

M_n and M_{n-1} respectively; *count* array is initialized to be 1 for $\boxed{\circ}\boxed{\circ}\cdots\boxed{\circ}$; the answer is the total number of cycles found during the algorithm.

5.2.3 Techniques to improve the efficiency

In-place update of the array

If we eliminate the *tmp* array, the memory usage of Algorithm 2 can be reduced by half. We achieve it by finding a good access order to the *count* array elements. Let $1 \le j \le n-1$, $\alpha \in \Sigma'^{j-1}$, $\beta \in \Sigma'^{n-j-1}$, and $c \in \Sigma'$. We choose the minimal perfect hash functions that satisfy the following relations:

$$\begin{aligned} \varphi_j(s) &= \varphi_{j+1}(s) && \text{for all } s \in S'; \\ \varphi_j(\alpha\boxed{\bullet}c\beta) &= \varphi_{j+1}(\alpha c\boxed{\bullet}\beta) && \text{for all } \alpha\boxed{\bullet}c\beta \in S''_j. \end{aligned} \tag{5.4}$$

Hereafter, we write as $count(s)$ to denote $count[\varphi_j(s)]$ because position j is not important anymore.

Data dependency in the *count* array is illustrated in Figure 5.11. For instance, $\boxed{(}\boxed{)}$ represents a set of the frontier states that have $\boxed{(}$ at position j and $\boxed{)}$ at position $j+1$ when we are processing a horizontal line between columns j and $j+1$. An arrow from $\boxed{\circ}\boxed{\circ}$ to $\boxed{(}\boxed{)}$ shows that a value read from $count(\alpha\boxed{\circ}\boxed{\circ}\beta)$ is added into $count(\alpha\boxed{(}\boxed{)}\beta)$ for all $\alpha\boxed{\circ}\boxed{\circ}\beta \in S'$. Since $count(\alpha\boxed{\circ}\boxed{\circ}\beta)$ is used to calculate the new value of $count(\alpha\boxed{(}\boxed{)}\beta)$, we update $count(\alpha\boxed{(}\boxed{)}\beta)$ before adding the value of $count(\alpha\boxed{\bullet}\boxed{\circ}\beta)$ into $count(\alpha\boxed{\circ}\boxed{\circ}\beta)$. We update $count(\alpha\boxed{)}\boxed{\bullet}\beta)$ and $count(\alpha\boxed{\circ}\boxed{)}\beta)$ simultaneously because they depend on each other.

Fast mapping from frontier states to serial numbers

Hash functions are evaluated frequently and have a large impact to the total speedup. In this section, we describe a minimal perfect hash function for main states. One for blocked states can be made in the same way by ignoring $\boxed{\bullet}$ and considering that the string size is $n-1$.

The minimal perfect hash function for main states $\varphi' : S' \to \{1, \dots, |S'|\}$ is implemented as a sum of two subfunctions:

$$\varphi'(s) = \varphi'_L(l) + \varphi'_R(r) \tag{5.5}$$

where l is the left half of string s and r is the right half of s. The sizes of s, l, and r are n, $m = \lfloor n/2 \rfloor$, and $n-m$, respectively. This composition works because the main states correspond to $\mathscr{M}_{(0,1)\to(n,0)}$. The left half must be in $\mathscr{M}_{(0,1)\to(m,h)}$ and the right half must be in $\mathscr{M}_{(m,h)\to(n,0)}$ where $0 \le h \le m$. Let $base_h + i$ denote the i-th serial number of the main states passing through (m,h). They are given by:

$$base_0 \;=\; 0\,; \quad base_{h+1} \;=\; base_h + \left|\mathscr{M}_{(0,1)\to(m,h)}\right| \cdot \left|\mathscr{M}_{(m,h)\to(n,0)}\right| \,. \tag{5.6}$$

When $1 \le i_l \le \left|\mathscr{M}_{(0,1)\to(m,h)}\right|$ is a serial number of string l in $\mathscr{M}_{(0,1)\to(m,h)}$ and $1 \le i_r \le \left|\mathscr{M}_{(m,h)\to(n,0)}\right|$ is a serial number of string r in $\mathscr{M}_{(m,h)\to(n,0)}$, the subfunctions can be defined as:

$$\varphi'_L(l) \;=\; base_h + (i_l - 1) \cdot \left|\mathscr{M}_{(m,h)\to(n,0)}\right| \;; \quad \varphi'_R(r) \;=\; i_r \,. \tag{5.7}$$

Main states are implemented as $2n$-bit codes under such 2-bit code assignments as $\boxed{\circ} = 00$, $\boxed{)} = 01$, and $\boxed{(} = 10$. We divide a $2n$-bit code into the left $2m$-bit subcode and the right $2(n-m)$-bit subcode. The two subfunctions can be implemented by the simple arrays that are indexed directly by the subcodes. The total size of the two arrays is about 2^{n+1} and is small enough in comparison with the size of the *count* array.

Fast enumeration of all main states

As shown in 5.2.3, we want to enumerate all main states in some specific order for in-place update of *count* entries. A simple implementation would be a single array of states sorted in the desired order. Although it is very fast and accepts any state order, it consumes memory as much as the *count* array. We introduce a method with much better memory efficiency for visiting main states in lexicographic order.

We divide main states again into the left and right halves. A left state array *lstate* has ordered collection of $\langle l, h \rangle$ values where l is a left state subcode and l corresponds to $\mathscr{M}_{(0,1)\to(m,h)}$. A right state array $rstate_h$ ($0 \le h \le m$) has ordered collection of right state subcodes in $\mathscr{M}_{(m,h)\to(n,0)}$. For each $\langle l, h \rangle$ in *lstate* and for each r in $rstate_h$, main state lr is visited. Running time of this double loop would be almost the same as the single array implementation because most of the time is consumed in the inner loop and it is just scanning $rstate_h$ in the same way as the single array implementation.

Shared memory parallel processing

Let $s = c_1 c_2 \cdots c_n$ be a state string and the current step is for the horizontal line connecting columns j and $j+1$. As shown in Figure 5.9, the transition patterns of state strings have much locality. Except for modification of c_j and c_{j+1}, at most one position can be changed from $\boxed{(}$ to $\boxed{)}$ or from $\boxed{)}$ to $\boxed{(}$.

We can partition the states into 2^{n-2} groups across which no transition occurs, based on an $(n-2)$-bit binary code "$g(c_1) \cdots g(c_{j-1})\, g(c_{j+2}) \cdots g(c_n)$" where $g(c) = 1$ if $c \in \{\boxed{(}, \boxed{)}\}$, otherwise $g(c) = 0$. It is suitable for parallel processing since data update within a group is independent of other groups. We use the leftmost m-bit of the binary code for task allocation, where m is decided to make enough number of tasks and not to make each task too small; actually, $m \simeq n/2$ would not be a bad choice.

5.2.4 Experimental results

We have compared the original program for general graphs and three new programs specialized for grid graphs:

Original a SIMPATH-based sequential program described in 5.2.1;

Table 5.11: Memory usage (MB)

n	Original	Grid-BS	Grid-FS	Grid-FP
11	7	2	3	3
12	18	6	6	6
13	44	15	15	15
14	145	40	41	41
15	432	110	111	111
16	1290	304	306	306
17	3676	847	849	849
18	9993	2367	2376	2376
19	34298	6641	6663	6663
20	95329	18688	18729	18729
21	297260	52723	52791	52791
22	$\sim$700000	149108	149254	149254
23	$>$1.5TB	422634	422861	422861

Grid-BS a basic sequential program for grid graphs based on the techniques in 5.2.2 and 5.2.3 with *count* array of 56-byte integers;

Grid-FS a faster version of Grid-BS using the techniques in 5.2.3 and 5.2.3;

Grid-FP a parallelized version of Grid-FS using the technique in 5.2.3.

All those programs are written in C++. CPU time was measured on a machine with four Xeon E7-8837 (2.67GHz, 8 cores) processors and 1.5TB of memory, running 64-bit SUSE Linux Enterprise Server 11. We have assigned 12 CPU cores for Grid-FP. Results for $(n+1)\times(n+1)$ grid graphs are shown in Table 5.11 and Table 5.12. The original program could not finish $n = 22$ within a week and could not compute $n = 23$ because of insufficient memory. In comparison with the original program, space and time efficiency has been improved five times and ten times respectively in the sequential processing (Grid-FS), and time improvement of another digit is achieved by the parallel processing on 12 CPU cores (Grid-FP). Grid-FS used slightly more memory than Grid-BS, while it achieved three times speedup.

Table 5.12: Computation time (sec.)

n	Original	Grid-BS	Grid-FS	Grid-FP
11	0.7	0.5	0.1	0.0
12	2.4	1.6	0.3	0.1
13	8.6	5.1	1.0	0.2
14	38.0	16.7	4.7	0.5
15	140.1	53.7	14.0	1.9
16	508.1	172.5	45.8	7.9
17	1763.7	554.1	146.4	20.9
18	6003.0	1755.0	459.3	67.6
19	17961.9	5687.2	1759.6	212.4
20	61570.0	18121.4	4616.1	652.0
21	208001.7	56263.6	17917.2	3244.3
22	>1 week	178439.6	53671.1	5695.0
23	N/A	554159.8	170475.7	21313.0

Table 5.13: The number of paths between opposite corners of $G_{n+1,n+1}$

n	#path
1	2
2	12
3	184
4	8512
5	1262816
6	575780564
7	789360053252
8	3266598486981642
9	41044208702632496804
10	1568758030464750013214100
11	182413291514248049241470885236
12	64528039343270018963357185158482118
13	69450664761521361664274701548907358996488
14	227449714676812739631826459327989863387613323440
15	2266745568862672746374567396713098934866324885408319028
16	68745445609149931587631563132489232824587945968099457285419306
17	6344814611237963971310297540795524400449443986866480693646369387855336
18	1782112840842065129893384946652325275167838065704767655931452474605826692782532
19	1523344971704879993080742810319229690899454255323294555776029866737355060592877569255844
20	3962892199823037560207299517133362502106339705739463771515237113377010682364035706704472064940398
21	31374751050137102720420538137382214513103312193698723653061351991346433379389385793965576992246021316463868
22	755970286667345339661519123315222619353103732072409481167391410479517925792743631234987038883317634987271171404439792
23	55435429355237477009914318489061437930690379970964331332556958646484008407334885544566386924020875711242060085408513482933945720
24	12371712231207064758338744862673570832373041989012943539678727080484951695515930485641394550792153037191858028212512280926600304581386791094
25	8402974857881133471007083745436809127296054293775383549824742623937028497898215256929178577083970960121625602506027316549718402106494049978375604247408
26	17369931586279272931175440421236498900372229588288140604663703720910342413276134762789218193498006107082296223143380491348290026721931129627708738890853908108906396

Table 5.14: The number of cycles in $G_{n+1,n+1}$

n	#cycle
1	1
2	13
3	213
4	9349
5	1222363
6	487150371
7	603841648931
8	2318527339461265
9	27359264067916806101
10	988808811046283595068099
11	109331355810135629946698361371
12	36954917962039884953387868334644457
13	38157703688577845304445530851242055267353
14	1202857893405338595558405124213592877516931371715
15	1157093265735985301622023713535739570361128480949044165
16	33953844023386621558992302713698922032806419330748312822875435
17	3038442783379807358500340876029076156976090161888565971315511241074745
18	828994780940754546935155193723999456397444612351458013833196066822882956458645
19	689454183113508056830743257694610073918027584884435945831932331941835810292154647051579
20	1747591025316879549979786952655665410563226741505288698037078203555463416502484945892166703679171
21	13498757056229389624710286314663261122166783333470228470330291400847321675907180262434015144922092043438151
22	317698324732748859895908541034955194127994098630324812636624826258997224150342908049795056399595180923955046092422345
23	22780223929670574691223507874671963939543945238061927803443588158966878341201898099174425109669737384772981749340922723497405087
24	4976037374055300327445222400443546830498798429935446621053707681453137859077914419511395195468579271956320503531047087399178114550443364089
25	3310986696180301144320032978096652329596523503435882543114890295466038827631907053110552071003521069706315474677769157738758382711047460804662161917199
26	6710410656552530384649365616353468091258297745658800698270685658306730312047500343532306146532051342931562598353663208665087446435996989284936423311881624210082193

As the parallel program runs fast enough, it is worth exchanging space costs for time costs using the Chinese Remainder Theorem in the same way as [BMGJ05]. Using a 64-bit modular arithmetic version of Grid-FP, we have succeeded in discovering the answers to $n = 25$ and $n = 26$, which are 151-digit and 164-digit numbers respectively, as well as confirming the all past answers ($n \leq 24$) in OEIS A007764 [OEIa]. The numbers are listed in Table 5.13. We performed 10 runs with different moduli for $n = 26$; each run took 2 days on 40 CPU cores and used 1400 gigabytes of memory. For the number of cycles, our program also have confirmed the all past answers to $n \leq 19$ in OEIS A140517 [OEIb] and newly found the answers to $20 \leq n \leq 26$ as listed in Table 5.14.

5.2.5 Additional techniques

Using line symmetry

The problem is line symmetrical with respect to a line passing through $v_{(1,1)}$ and $v_{(n,n)}$; the number of paths starting from $\{v_{(1,1)}, v_{(1,2)}\}$ is exactly the same as the number of paths starting from $\{v_{(1,1)}, v_{(2,1)}\}$. It means that the final answer can be computed by doubling either answer. This method, however, would not contribute much computational reduction because both cases can reach soon onto the entire set of frontier states.

One could make some algorithm utilizing line symmetry by introducing diagonal frontiers instead of horizontal frontiers. That approach seems difficult because it have to overcome faster growth of the number of frontier states than using horizontal frontiers. Figure 5.12 compares the peak numbers of frontier states measured in the program for general graphs.

Using point symmetry

When we use the point symmetry of the problem, Algorithm 2 can be stopped at the middle of computation. We can compute the answer by enumerating every pair of main states that matches. Figure 5.13 shows that paths are completed by $A = \boxed{(}\boxed{(}\boxed{)}\boxed{)}\boxed{\circ}\boxed{)}$ and $B = \boxed{(}\boxed{\circ}\boxed{)}\boxed{)}\boxed{(}\boxed{)}$ ($C = \boxed{(}\boxed{\circ}\boxed{(}\boxed{)}\boxed{)}\boxed{)}$)

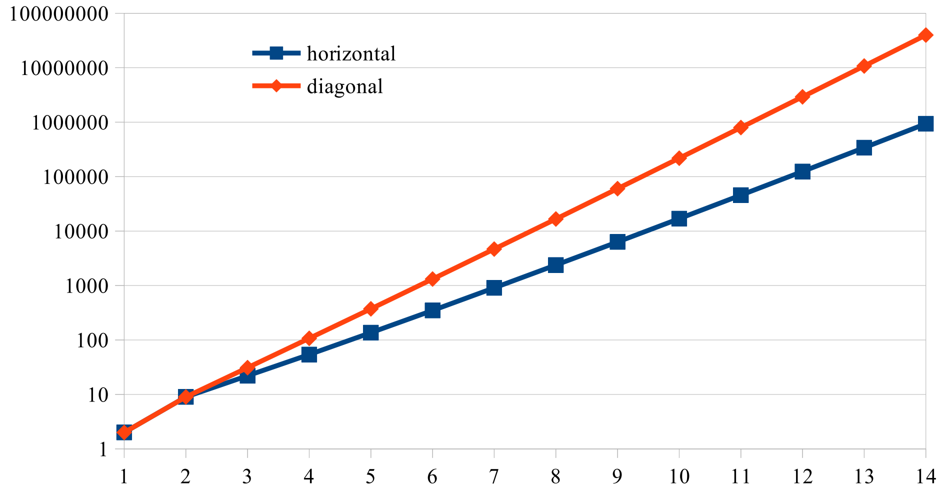


Figure 5.12: The number of frontier states against n

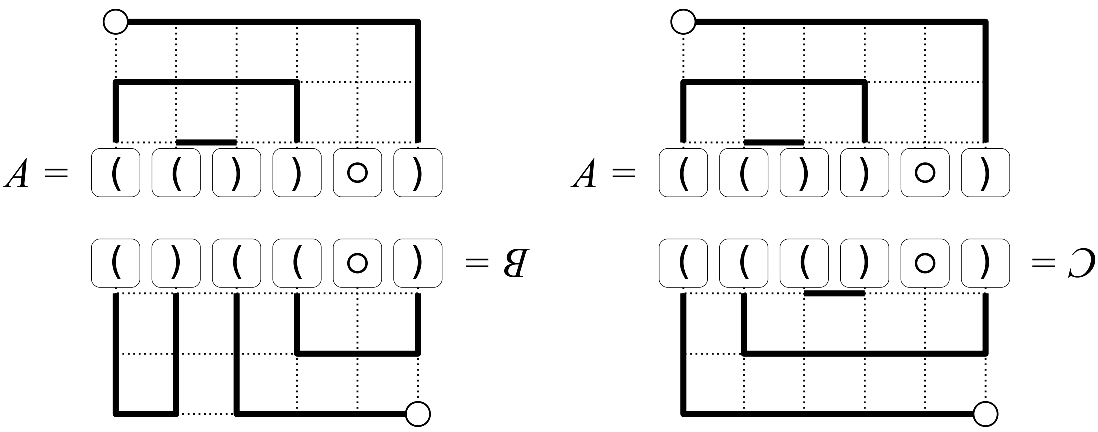


Figure 5.13: Example of matching

where B (C) is turned over and they are combined. The total number of paths found here is $count(A) \times (count(B) + count(C))$.

Numbers stored in *count* grows exponentially against the number of loop iterations in the algorithm. This technique reduces the memory usage by half because integer size can be halved when we only compute numbers for the upper half of the graph. As for the time cost, however, this technique might be at a disadvantage. Time growth of the matching process against n was larger than that of the basic algorithm in our preliminary experiments.

5.3 Chapter summary

First, we proposed an efficient top-down ZDD construction framework for solving graph enumeration and indexing problems, in which the property is specified as a recursive specification.

The lookahead technique applies the zero-suppress rule on the fly, while conventional methods do not take it into account during the top-down construction phase. Binary operations on recursive specifications, which allow us to combine multiple properties without constructing ZDD structure for each property, can be a strong alternative to the conventional procedures that use operations on ZDDs. An improved algorithm for set intersection on recursive specifications also reduces CPU time and memory usage for constructing ZDDs by skipping redundant states in the top-down construction phase. The subsetting technique enables us to use an existing ZDD for restricting search space of top-down ZDD construction.

The experimental results confirmed that our techniques actually improve time and space for the top-down ZDD construction algorithms. The lookahead technique easily fits for any application and produces good results in most cases. We can improve it further by adding the subsetting technique especially for large examples, though it is not always easy to find the best strategy of subsetting.

Secondly, we have maximized space and time efficiency of the path enumeration algorithm by focusing on grid graphs. We have succeeded in clarifying the exact domain of state codes and analyzing their transition patterns. They allowed us to use simple arrays instead of ordinary hash tables and to integrate various techniques into the algorithm, namely, in-place update, fast hash functions, fast state enumeration, and parallel processing.

The new algorithm for grid graphs have achieved double-digit performance improvement over the original algorithm for general graphs. It have extended the numbers recorded in OEIS [OEIa, OEIb] and have verified all those past results correctly.

CHAPTER 6

Conclusions

We saw in previous chapters that binary decision diagram representation is a core technology against property verification problems. It is very useful and effective in enumerating all possible cases for the property. We also learned that the applications are supported with two layers of BDD-related technologies: the lower layer manipulates BDDs under such mathematical abstraction as Boolean functions and families of sets; the higher layer models individual problems mathematically and solves them using BDD operations provided by the lower layer. As they have clean boundary, excellent implementations of the lower layer have been reused as BDD packages. Researchers of both layers have been able to focus on their own area, which greatly accelerated the spread of BDD applications.

Chapters 3 and 4 belong to the higher layer; we used matured BDD packages as they were and built our own algorithms on them. The most important task in the simulation-based validation problem was the formulation. Initially, a target of verification is often ambiguous; for instance, people just want to make their system bug-free. We first have to "design" input and output of the problem so that all information can be extracted from the human input and computers can make the output at a reasonable cost. In the model checking problem, we somewhat changed input and output from the conventional method in those days. We focused on ω-regular language emptiness check, which does not cover all traditional CTL properties. It is, however, practical in terms of human input and computational cost; it is easy to understand for humans and is covered by the efficient model checking algorithms.

Chapter 5 described recent work of extending the lower layer aiming for novel applications using frontier-based methods. We have seen possibility of the frontier-based methods through world records of computing the numbers of self-avoiding walks. Our framework encapsulates the ad hoc part of the frontier-based methods using a recursive specification and leaves it as an object to be implemented in the higher layer. In the same way as the success story of traditional BDD packages, improvement of the lower layer benefits any higher layer application. This framework is realized as a top-down ZDD construction library in C++, which includes our new techniques, such as lookahead and subsetting.

Wide variety of techniques for property verification have been developed through this work. The next step should be more studies of the new higher layer; applications based on frontier-based methods. They would be built on our top-down ZDD construction library. As already mentioned above, a key to practical application is problem formulation; extensive techniques and know-hows will be required for it. I am hopeful that the experiences in this work can contribute to plenty of future applications.

Bibliography

[AC94] Pranav Ashar and Matthew Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '94, pages 622–627, 1994.

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[BB94] Derek L. Beatty and Randal E. Bryant. Formally verifying a microprocessor using a simulation methodology. In *Proceedings of the 31st Design Automation Conference*, DAC '94, pages 596–602, 1994.

[BCL91] Jerry R. Burch, Edmund M. Clarke, and David E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th Design Automation Conference*, DAC '91, pages 403–407, 1991.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BD94a] Vishal Bhagwati and Srinivas Devadas. Automatic verification of pipelined microprocessors. In *Proceedings of the 31st Design Automation Conference*, DAC '94, pages 603–608, 1994.

[BD94b] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *Proceedings of the 6th International Conference on Computer Aided Verification*, CAV '94, pages 68–80, 1994.

[BHSV+96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano Villa. VIS: A system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 428–432, 1996.

[BMGJ05] Mireille Bousquet-Mélou, Anthony J. Guttmann, and Iwan Jensen. Self-avoiding walks crossing a square. *Journal of Physics A: Mathematical and General*, 38:9159–9181, 2005.

[BRB90] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th Design Automation Conference*, DAC '90, pages 40–45, 1990.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986.

[CCQ96] Gianpiero Cabodi, Paolo Camurati, and Stefano Quer. Improved reachability analysis of large finite state machines. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, pages 354–360, 1996.

[CEG93] A. R. Conway, I. G. Enting, and A. J. Guttmann. Algebraic techniques for enumerating self-avoiding walks on the square lattice. *Journal of Physics A: Mathematical and General*, 26:1519–1534, 1993.

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transitions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGMZ95] Edmund M. Clarke, Orna Grumberg, Kenneth L. McMillan, and Xudong Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd Design Automation Conference*, DAC '95, pages 427–432, 1995.

[Cou97] Olivier Coudert. Solving graph optimization problems with ZBDDs. In *Proceedings of the European Design and Test Conference*, ED&TC '97, pages 224–228, 1997.

[CYB97] Yirng-An Chen, Bwolen Yang, and Randal E. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach. Technical Report CMU-CS-97-120, School of Computer Science, Carnegie Mellon University, 1997.

[CYF94] Ben Chen, Michihiro Yamazaki, and Masahiro Fujita. Bug identification of a real chip design by symbolic model checking. In *Proceedings of European Design and Test Conference*, ED&TC '94, pages 132–136. IEEE, 1994.

[Dea] Seiji Doi and et al. Time with class! let's count! `http://www.youtube.com/watch?v=Q4gTV4r0zRs`.

[DS77] Robert Donaghey and Louis W. Shapiro. Motzkin numbers. *Journal of Combinatorial Theory, Seires A*, 23(3):291–301, 1977.

[Flo49] Paul J. Flory. The configuration of real polymer chains. *Journal of Chemical Physics*, 17:303–310, 1949.

[HHY89] Kiyoharu Hamaguchi, Hiroki Hiraishi, and Shuzo Yajima. Formal verification of sequential machines using an omega model checking algorithm of ∞ regular temporal logic. Technical Report COMP89-24, IEICE, 1989.

[Hir89] Hiromi Hiraishi. Design verification of sequential machines based on ε-free regular temporal logic. In *Proceedings of IFIP Symposium on Computer Hardware Description Languages and Their Applications*, pages 249–263, 1989.

[HK90] Zvi Har'El and Robert P. Kurshan. Software for analytical development of communications protocols. *AT&T Technical Journal*, 69(1):45–59, 1990.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Kaufmann, 1990.

[HTKB93] Ramin Hojati, Hervé J. Touati, Robert P. Kurshan, and Robert K. Brayton. Efficient ω-regular language containment. In *Proceedings of the Fourth International Workshop on Computer Aided Verification*, CAV '92, pages 396–409, 1993.

[IKM12] Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. ZDD-based computation of the number of paths in a graph. Technical Report TCS-TR-A-12-60, Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, 2012.

[IKNH94a] Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata, and Fumiyasu Hirose. Automatic program generator for simulation-based processor verification. In *Proceedings of the Third Asian Test Symposium*, ATS '94, pages 298–303, 1994.

[IKNH94b] Hiroaki Iwashita, Satoshi Kowatari, Tsuneo Nakata, and Fumiyasu Hirose. Automatic test program generation for pipelined processors. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '94, pages 580–583, 1994.

[IN97] Hiroaki Iwashita and Tsuneo Nakata. Forward model checking techniques oriented to buggy designs. In *Proceedings of the*

1997 IEEE/ACM International Conference on Computer-Aided Design, ICCAD '97, pages 400–404, 1997.

[IN99] Hiroaki Iwashita and Tsuneo Nakata. Efficient forward model checking algorithm for ω-regular properties. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E82-A(11):2448–2454, 1999.

[INH92] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. Behavioral design and test assistance for pipelined processors. In *Proceedings of the First Asian Test Symposium*, ATS '92, pages 8–13, 1992.

[INH93] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. Integrated design and test assistance for pipeline controllers. *IEICE Transactions on Information and Systems*, E76-D(7):747–754, 1993.

[INH96] Hiroaki Iwashita, Tsuneo Nakata, and Fumiyasu Hirose. CTL model checking based on forward state traversal. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, pages 82–87, 1996.

[INK$^+$13a] Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Efficient computation of the number of paths in a grid graph with minimal perfect hash functions. Technical Report TCS-TR-A-13-64, Division of Computer Science, Graduate School of Information Science and Technology, Hokkaido University, 2013.

[INK$^+$13b] Hiroaki Iwashita, Yoshio Nakazawa, Jun Kawahara, Takeaki Uno, and Shin-ichi Minato. Fast computation of the number of paths in a grid graph. In *The 16th Japan Conference on Discrete and Computational Geometry and Graphs*, JCDCG^2, 2013.

[Joh91] Mike Johnson. *Super-Scalar Processor Design*. Prentice Hall, 1991.

[Knu] Donald E. Knuth. Don Knuth's home page. `http://www-cs-staff.stanford.edu/~uno/`.

[Knu11] Donald E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, 1st edition, 2011.

[LS91] David C. Lee and Daniel P. Siewiorek. Functional test generation for pipelined computer implementations. In *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, FTCS-21, pages 60–67, 1991.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th Design Automation Conference*, DAC '93, pages 272–277, 1993.

[Min13] Shin-ichi Minato. Techniques of BDD/ZDD: Brief history and recent activity. *IEICE Transactions on Information and Systems*, E96-D(7), 2013.

[MIY90] Shin-ichi Minato, Nagisa Ishiura, and Shuzo Yajima. Shared binary decision diagram with attribute edges for efficient boolean function manipulation. In *Proceedings of the 27th Design Automation Conference*, DAC '90, pages 52–57, 1990.

[MS93] Neal Madras and Gordon Slade. *The Self-Avoiding Walk*. Birkhäuser, 1993.

[NIJ+97] Amit Narayan, Adrian J. Isles, Jawahar Jain, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Reachability analysis using partitioned-ROBDDs. In *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '97, pages 388–393, 1997.

[Nika] Nikoli. Nikoli.com puzzle championship. `http://www.nikoli.com/en/event/puzzle_hayatoki.html`.

[Nikb] Nikoli. Sample problems of slitherlink puzzle. `http://www.nikoli.com/en/puzzles/slitherlink/`.

[Nikc] Nikoli. Web nikoli. `http://www.nikoli.co.jp/`.

[Nik89] Nikoli. *Numberlink 1*. Nikoli, Tokyo, Japan, 1989.

[Nik92] Nikoli. *Slitherlink 1*. Nikoli, Tokyo, Japan, 1992.

[OEIa] OEIS Foundation. A007764: Number of nonintersecting (or self-avoiding) rook paths joining opposite corners of an $n \times n$ grid. The On-Line Encyclopedia of Integer Sequences, `http://oeis.org/A007764`.

[OEIb] OEIS Foundation. A140517: Number of cycles in an $n \times n$ grid. The On-Line Encyclopedia of Integer Sequences, `http://oeis.org/A140517`.

[OYY93] Hiroyuki Ochi, Koichi Yasuoka, and Shuzo Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '93, pages 48–55, 1993.

[RS95] Kavita Ravi and Fabio Somenzi. High-density reachability analysis. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '95, pages 154–158, 1995.

[SI97] Kyoko Sekine and Hiroshi Imai. Counting the number of paths in a graph via BDDs. In *IEICE Transactions on Fundamentals*, volume E80-A, pages 682–688, 1997.

[SIT95] Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proceedings of the 6th International Symposium on Algorithms and Computation*, ISAAC '95, pages 224–233, 1995.

[SPA92] SPARC International. Sparc version 9 draft 1.0.3, 1992.

[SRBSV96] Jagesh V. Sanghavi, Rajeev K. Ranjan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *Proceedings of the 33rd Design Automation Conference*, DAC '96, pages 635–640, 1996.

[TBK95] Hervé J. Touati, Robert K. Brayton, and Robert Kurshan. Testing language containment for ω-automata using BDDs. *Information and Computation*, 118(1):101–109, 1995.

[TSL$^+$90] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. In *Proceedings of the 1990 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '90, pages 130–133. IEEE, 1990.

[TSN98] Koichiro Takayama, Taizo Satoh, and Tsuneo Nakata. An approach to verify a large scale system-on-a-chip using symbolic model checking. In *Proceedings of the 1998 International Conference on Computer Design*, ICCD '98, page 308, 1998.

[Wei] Eric W. Weisstein. Self-avoiding walk. From MathWorld—A Wolfram Web Resource, `http://mathworld.wolfram.com/Self-AvoidingWalk.html`.

[YSK$^+$12] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.