学位請求論文

Logic Design Assistance System based on Temporal Logic

(時相論理に基づく論理設計支援システム)

1989年12月22日

指導教官　田中英彦 教授

東京大学大学院工学系研究科電気工学専攻

博士課程3年　77078　中村 宏

Logic Design Assistance System based on Temporal Logic

Hiroshi Nakamura

Department of Electrical Engineering
The University of Tokyo
7-3-1 Hongou, Bunkyo-ku
Tokyo 113, Japan

December 22,1989

Abstract

In this thesis, a logic design assistance system based on temporal logic is presented. There are three key points in this system: how to specify the behavior, how to verify the control part, and how to verify the data path.

A new behavioral description language named Tokio is proposed. Tokio is based on interval temporal logic. Tokio has the following three characteristics.

- Sequentiality and concurrency can be specified accurately and simply.
- Specified behaviors can be simulated.
- The behavioral descriptions at both the algorithmic level and the register transfer level are given in the same language.

A verifier for control part is presented in this thesis. In this verifier, a structure is verified formally whether it satisfies a given specification. This verifier is applied to two examples.

A data path verifier is described next. This is the central part of this assistance system. The inputs are the behavior and the structure at the register transfer level. The following items are realized by this verifier.

- Link informations between the behavior and the structure are derived automatically.
- Consistency between the behavior and the structure is verified automatically.
- The logic of the control part is derived automatically.

This data path verifier is also successfully applied to three examples.

Performance of this system is discussed and the proposed system is concluded to have enough power to assist practical hardware design.

Acknowledgement

My deepest gratitude goes to my supervisor, Hidehiko Tanaka. His great feeling for what research areas are important and intersting was a good guide during my research.

I am very grateful to the late Tohru Moto-oka. He led me to investigate CAD.

I wish to thank Masahiro Fujita for valuable discussions and guidance.

I also wish to thank Shinji Kono for comments on this work.

I thank Masaya Nakai and Yuji Kukimoto for their deep contributions to implement the assistance system. Takeshi Shimizu agreed to use the data of the network interface processor.

The students in Tanaka Laboratory have made my stay here a pleasant one, thanks to Tadashi Saito, Fumiko Kouda, Tadashi Omori, Kouichi Doi, Minoru Yoshida, Eiichi Takahashi, Yasuo Hidaka, Hirohiko Sagawa, Toshihiro Nemoto, and all other members in Tanaka Lab.

Many thanks go to the following people for happy student lives: Takahiko Yamazaki, Yoshitaka Okada, Chinae Yonezawa, and Hiroki Konaka.

Finally, my most sincere thanks go to my parents for their devoted supports.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, computer systems have become larger and more complicated. They have also spread over and have got the most essential part in our lives. The popularization of ASIC(Application Specific Integrated Circuit)s typically represents this matter. Along with this popularization, there arise much more requirements to develop digital systems in short turn-around and without errors. In order to meet these requests, CAD (Computer Aided Design) tools are indispensable.

The process of designing digital systems consists of several hierarchical stages, that is, system level design, logic design, and implementation design. Although CAD systems for implementation design are quite practical and utilized in actual designs, upper levels of the design hierarchy are not assisted enough. Therefore, a logic design assistance system is proposed in this thesis.

There has not been an excellent assistance method yet with which everyone agrees, though there have been many researches in this area. The strong motivation in this thesis is to establish a *formal* assistance method. Formal reasonings are very important and helpful for us. It is tried to formalize the assistance with temporal logic.

Temporal logic has recently attracted researchers attention because it can specify

concurrency and sequentiality accurately and it is suitable for describing behaviors of hardware.

In this thesis, a logic design assistance system based on temporal logic is presented and the power of the system is evaluated.

1.2 Organization of Thesis

This thesis has the organization as follows.

- Chapter 2: Design flow of hardware design is overviewed and some attempts to assist a logic design are introduced. Hardware description languages are also summaried.
- Chapter 3: The proposed assistance system is described.
- Chapter 4: A new hardware description language Tokio is explained. This language is based on temporal logic.
- Chapter 5: A control part verifier is presented. Experimental results of the verifier are also shown in this chapter.
- Chapter 6: A data path verifier and its experimental results are presented. This is the central part of the proposed assistance system.
- Chapter 7: Performance and contribution of the proposed system are discussed.
- Chapter 8: The thesis is concluded.

Chapter 2

Survey on Logic Design

2.1 Design Flow of Circuit Design

In this section, the design flow of circuits design is reviewed briefly. The process of designing a very large scale intergration (VLSI) chip is usually divided into several stages, and the abstract specification is refined in each stage. The way of separating the process into stages depends on whether the designer has already designed other similar hardware or not, whether the designed hardware is large or small, and so on. In most cases, the process is divided as shown in Figure 2.1. Each design stage is explained in the followings.

2.1.1 System Design

First of all, designers must specify what to be designed. This process is called 'requirement specifications'. In this stage, the border line between the hardware and the software is not clear.

Next, designers decide how to process the given specification in this system design process. They select an algorithm which realizes the specification and separate the algorithm into two parts which are realized by hardware and software respectively. Which algorithm they select and how to separate the algorithm depends on both the required performance and the permitted costs. Then, the specifications of both

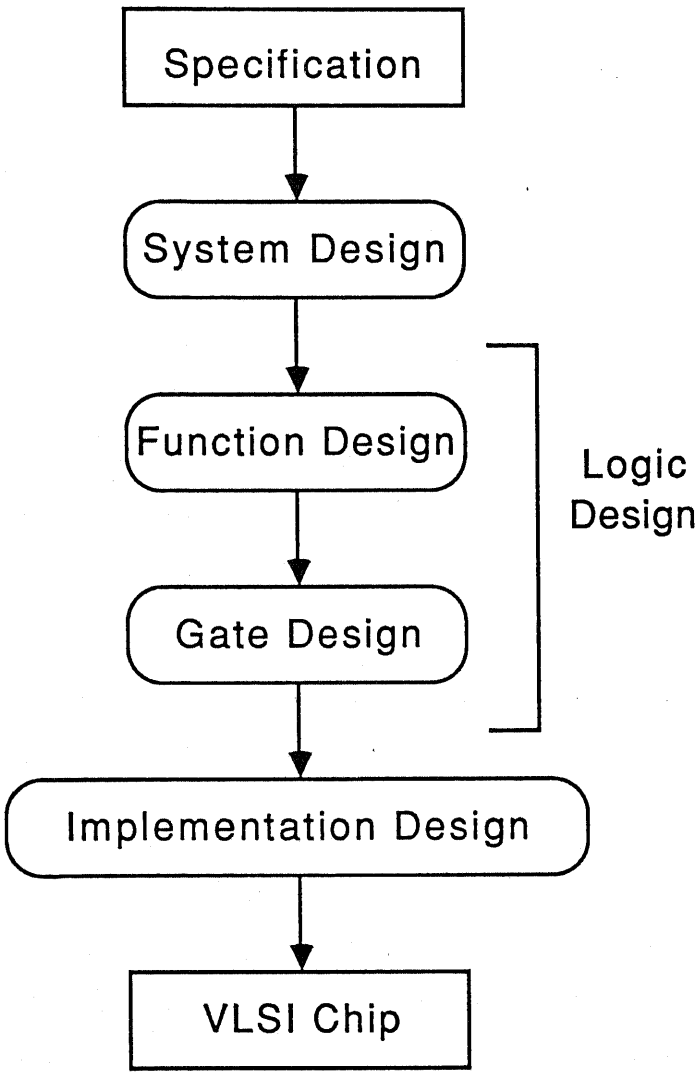


Figure 2.1: Design Flow of Circuit Design

hardware part and software part is derived. In the case of designing a processor, designers also decide instruction sets, rough hardware architecture (such as pipeline), and the way of executing the instructions on the architecture. The behavior of the hardware part is usually described in common languages, such as Ada, Pascal, C, etc, or in original language they use. This specification is usually verified through simulations.

The process is followed by the logic design stage. The logic design stage accepts the specification at the system level and produces networks of logical gates. The stage is divided into the two stages: function design stage and gate design stage.

2.1.2 Function Design

In this stage, behaviors at the system level are refined into other ones at the *register transfer level* which represents data transfers between registers and memories. This refinement is done with constructing a data path structure which consists of registers, multiplexers, busses, operating modules, and so on. The logic of control part is also designed. This logic indicates how to control the constructed data path in order to realize the behavior at the register transfer level.

This process is heavily affected by the constraints of required performance and permitted cost. Using more resources generally leads to hardware of high speed and high performance. Using too many resources, however, degrades the performance of the hardware, because designed chip becomes too large and delay of the hardware becomes too long. In this stage, several candidates of behaviors and structure are designed, and designers choose one of them. This is the key point for effective hardware design, but it is very difficult.

Consequently, this stage is almost done manually by designers. There have been many researches which aim at automatical function design, called *high level synthesis*. These researches are explained later in the section 2.2. Hardwares which

are designed using the techniques of high level synthesis are not satisfactory yet. Therefore, how to assist function design is still one of the most important topics in computer aided design.

Derived behaviors and structures at the register transfer level are described in hardware description languages (HDLs), such as DDL [DD68], ISPS [Bar81], HSL [SKS80], SDL [Van77]. Currently, in the most industries and academias, they have their original HDLs. On the other hand, the effort of standardizing many HDLs is active in many countries. For example, VHDL [Sha89] [Wax86] [Ayl86] is the standard in United States, UDL/I [Kar89] is going to be a Japanese standard, and so on.

2.1.3 Gate Design

In the previous function design stage, all the functions needed for executing the behavior are decided. In the gate design stage, it is decided how the functions are implemented with logical expressions. Functions are expanded into Boolean expressions or state transition tables, and they are simplified and transformed into networks of logical gates. The main works in this stage are the assignments of the states into flip-flops [DBS85] and simplification of the Boolean expressions [BMHS84][Sas84].

Some CAD (Computer Aided Design) tools of automatical gate design have already been produced on a commercial basis. In recent years, ASIC (Application Specific Integrated Circuit)s have been spread widely over in many fields of productions. Consequently, there arise many requirements to design LSI chips which do not require very high performance but entail shorter design cycle. The commercial CAD tools are useful and suitable for such kinds of LSI chips.

There also exist other demands on LSI chips of very high performance. As for such LSIs, gate design is still done manually. Many verification techniques are researched now in order to assist manual design. Boolean comparison (or tautology

check of boolean expressions) is the main topic in the verification of combinational circuit [HM88][WS86][Bry86][FMF89][MIY89]. Verification of sequential circuits are more difficult than that of combinational circuits because sequentiality should be considered. Usually, sequential circuits are divided into two parts: latches and combinational part. In addition to the techniques of handling combinational logic, the method of dealing with sequentiality is required in this verification.

Down to this stage, designs are independent of technologies. That is, designers need not alter the design process no matter which technology (for example, TTL, CMOS, etc.) is adopted.

2.1.4 Implementation Design

Results in the logic design stage are divided into several LSI chips and converted to fit the adopted technology. The division is decided by the constraints of the number of gates and pins allowed for one chip. Then, layouts and routings are processed. Much efforts to automate this stage have been done. As a result, CAD systems for implementation design seem to be very practical. However, it is required to design much larger and much more highly complexed systems, because the progress of device technology is drastic. Therefore, it is still necessary to develop implementation CAD systems of higher performance.

The design flow of hardware design has been reviewed. Design assistance at each stage is not enough yet. Particularly, both the system level design and the function design among the hierarchical hardware design.

Recently, researches on the so called "silicon compiler" become one of the most major topics in computer aided design. This "silicon compiler" aims to generate mask patterns directly from function designs. Key points in silicon compiler are also how to assist function design.

In the next section, researches which aim to automate function design is introduced. These researches are called "high level synthesis".

2.2 High Level Synthesis

In recent years there has been a strong motivation for automating synthesis at higher levels of the design hierarchy. There are a number of reasons for this:

- Shorter design cycle:
- Fewer errors
- The ability to search the design space
- The design process is obvious
- Availability of IC technologies to more people

In the 26th Design Automation Conference (held in 1989, sponsored by ACM and IEEE), there were several sessions concerning high level synthesis. Moreover, the fourth High-Level Synthesis Workshop (sponsored by ACM) was held in October, 1989.

A current status of 'high-level synthesis' is presented in section 2.2.1, synthesis task is surveyed in section 2.2.2, a research of high level synthesis is presented as an example in section 2.2.3, and open problems are described in section 2.2.4,

2.2.1 Current Status

High-level synthesis [Par84][MPC88][BD88] is the design and implementation of a digital circuit from a behavioral description. A behavioral description does not include structural information (for example, registers, busses, multiplexers, etc.) but describes the functions to be performed by the circuit in an algorithmic form. In

essence, a high-level specification is a program describing the behavior of the circuit to be designed. High-level synthesis should be distinguished from other types of synthesis, such as logic synthesis or system level synthesis which operate at different levels of the design hierarchy,

In the third 'High-Level Synthesis Workshop' (held in January 1988), four benchmarks were chosen. Out of the 18 presentations, 6 were from industry and 12 were from university researchers. The reason to select the benchmarks was due to the frustration felt by many of the participants at the previous workshop (held in May 1986 in Santa Barbara, CA) with the difficulty in comparing the quality, applicability, practicality, and originality.

The selected four examples are:

- Intel8251: a serial-line controller
- MCS6502: a small microprocessor
- DSP: a digital signal processing filter
- MC68000: a large microprocessor

Three examples of Intel8251, MCS6502, and MC68000 were available in ISPS [Bar81] format. A one-page data-flow graph was provided for the filter example.

Among the 18 presentations, 11 were related to these benchmarks. The filter example was the commonest (7), followed by the Intel8251 (5), and the MCS6502 (4). Only two presentations included results for the largest example, the MC68000.

2.2.2 Synthesis Task

The system to be designed is usually represented at the algorithmic level by a programming language such as Ada or Pascal [Tri87] or by a hardware description language such as ISPS [Bar81] or VHDL [Wax86]. Since hardware description languages

are the central part of CAD tools, ISPS and VHDL are mentioned in section 2.3.

The first step in high-level synthesis is usually the compilation of the formal language into an internal representation. Two types of internal representations are generally used: parse trees and graphs.

The next two steps in synthesis are the core of transforming behavior into structure, that is, scheduling and allocation. They are closely interrelated and depend on each other. Scheduling represents to assign the operations to so-called control steps. A control step refers to a machine cycle at the register transfer level. Allocation consists in assigning the operations to hardware. These assignments are allocation of functional units, storage and communication paths.

Scheduling aims to minimize the amount of the required number of control steps for executing behaviors within the limits of the available hardware resources.

In *allocation*, the problem is to minimize the amount of the hardware. The hardware consists of functional units, memory elements and communication paths. Since it is too complex to minimize them together, they are minimized separately in most proposed systems. For example, in the minimization of functional units, mutually exclusive operations clearly can share functional units. Then, the problem is then replaced by another problem how to gather those operations which are mutually exclusive. The problem of minimizing the amount of storage and the complexity of the communication paths for a given schedule can be formulated similarly. After both the scheduling and the allocation have finished, it is necessary to synthesize a controller which will drive the data paths.

The most difficult problem throughout all these tasks is how to narrow down the large "design space" into the real design. In order to select the optimal design which meets the constraints, extremely large number of design candidates should be examined.

2.2.3 Yorktown Silicon Compiler

In this section, a system named Yorktown Silicon Compiler [Cam87][Cam88] is presented as a typical example of high level synthesis system.

In the paper [Cam88], the automatic synthesis of an IBM 801 processing unit using the YSC system is presented. This YSC was developed at the IBM Thomas J. Watson Research Center. The IBM 801 architecture is a 32-bit architecture in the sense that most instructions, data, registers and addresses are 32-bit wide. In the following, the design process model of the YSC and the results of the automatic synthesis is presented. All compilation and synthesis times are given in CPU seconds on an IBM 3090-200 computer.

Design Process

The design process in the YSC is depicted in Figure 2.2. The compiler is coded in PL/1 and the structural synthesis and logic synthesis are coded in APL(interpreted).

Compiler The behavior is given in V form. The V specification is transformed into an internal format (called YIF: Yorktown Internal Format) consisting of a data flow graph and a control graph. An example of YIF is shown in Figure 2.3. The main tasks of this compilation include the decomposition of expressions into single operations and the mapping of complex data types into simple ones. The complete 801 design at this stage contains 1851 operation nodes and 8192 variables.

Structural Synthesis **1** The next task is merging nodes representing variables. Initially each bit is represented by a node. In the 801, for example, 32 bit arithmetic variables are compressed into two nodes. The 8192 variables in the YIF are reduced to around 1000 nodes.

2 The following task is to identify the cycles in the control graph and to eliminate

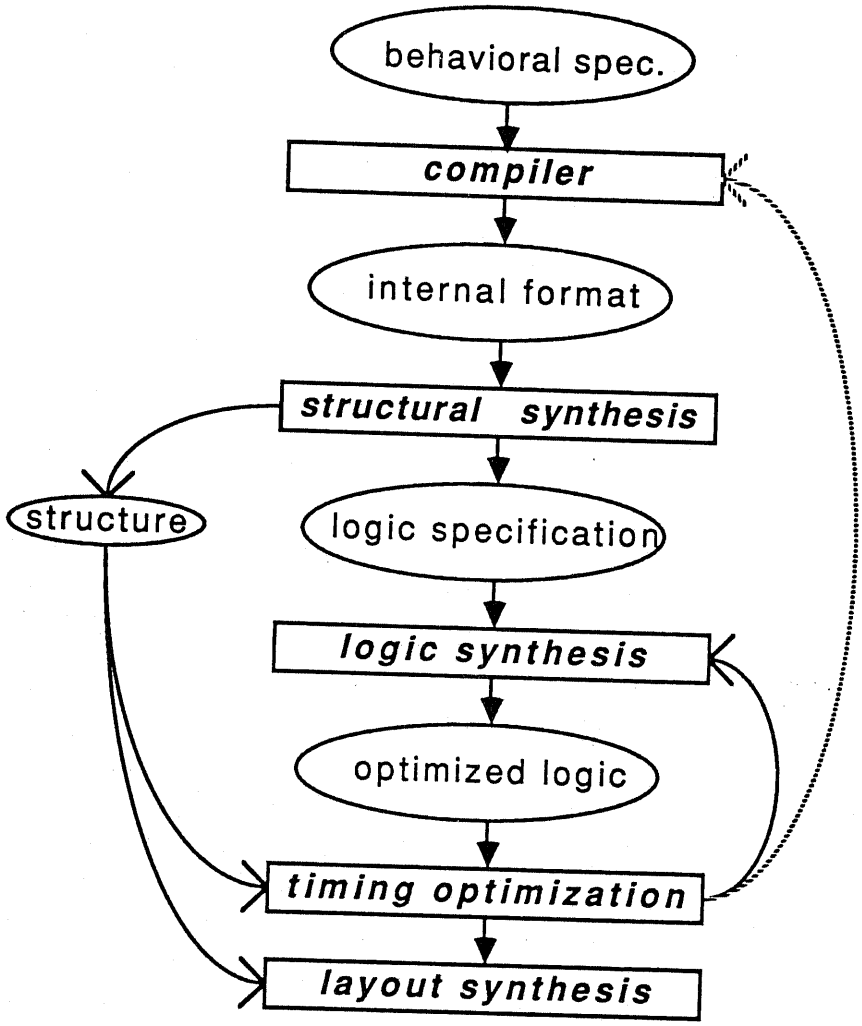
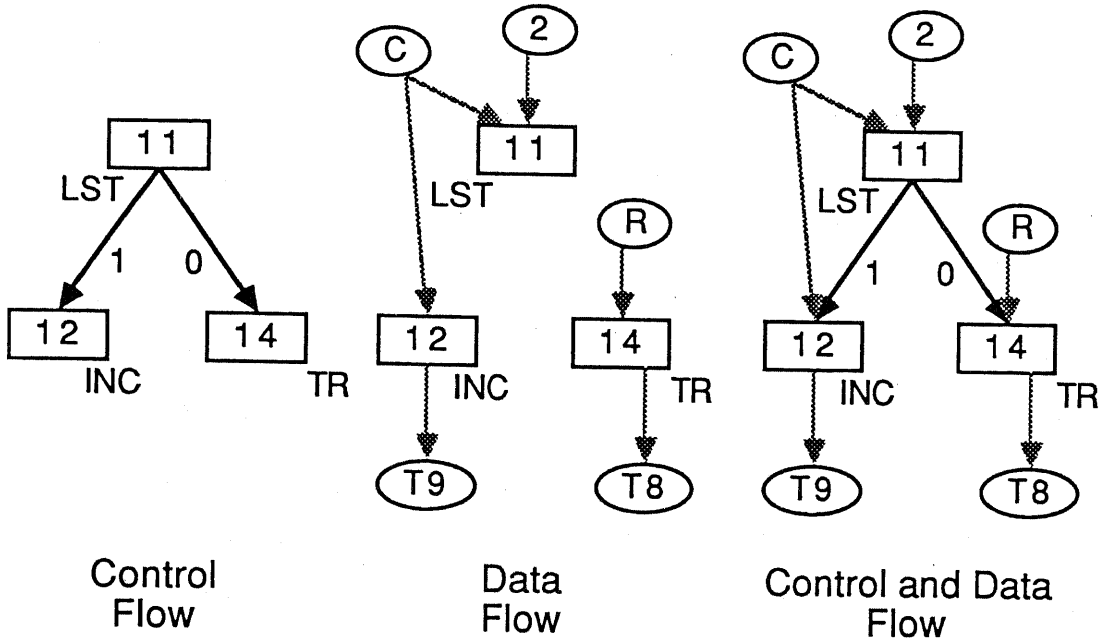


Figure 2.2: Design Process in the YSC




```
INDEX 11 TAG CBR OPERATION LST LINE 678;
INPUTS C(1..2)[0..0] 2;
OUTPUTS *;
PREDECESSORS 10;
SUCCESSORS 11 CONDITIONS 1;
SUCCESSORS 14 CONDITIONS 0;

INDEX 12 TAG S0 OPERATION INC LINE 702;
INPUTS C(1..2)[0..0];
OUTPUTS T9(1..2)[0..0];
PREDECESSORS 11;
SUCCESSORS 13 CONDITIONS *;

INDEX 14 TAG S0 OPERATION TR LINE 898;
INPUTS R(2..8)[0..0];
OUTPUTS T8(1..7)[0..0];
PREDECESSORS 11;
SUCCESSORS 15 CONDITIONS *;
```

Figure 2.3: YIF Example

them by removing edges. At this point, the concept of control steps is introduced. In general, finding the minimal set of edges to break all cycles in a graph is NP-hard. In the YSC, the operations involved in loops are tagged by the V compiler; this is easily done by identifying syntactically all possible loops. Thus, the stage of eliminating cycles is very fast.

3 This stage is the core of structural synthesis. First, all variables holding values which must cross a control step boundary are marked as registers. Second, data flow constraints like writing a register twice during one control step are detected. Then, new control steps are introduced carefully to solve the data flow constraints. Introducing new registers may create new data flow constraints. Consequently, these steps are repeated.

4 The next stage is called *variable unfolding.* Each variable is duplicated as many times as necessary to achieve single assignment.

5 By this stage, the YIF has been converted into a structure. A possible implementation consists of implementing each operation by combinational logic and providing a net for each unfolded variable. In order to reduce costs, the problem of allocating the minimum number of registers, of required operators, and of multiplexers is solved. This is called *folding.*

6 Finally, the structure is generated. The structure consists of a netlist connecting latches and blocks of combinational logic. Since the combinational logic of each module is minimized by logic synthesis, there is a limit to its size. If this limit is exceeded, the combinational logic is partitioned into several smaller blocks. Then, the structure is given in HND (Hierarchical Network Definition), and the logic function is given in YLL (Yorktown Logic Language).

Logic Synthesis Each combinational logic block is minimized separately during logic synthesis by the YLE(Yorktown Logic Editor). The YLE initially minimizes

the size of the combinational logic and produces a multi-level implementation, both for SCVC (Single Cascode Voltage Switch) and CMOS.

Timing Optimization Timing optimization then finds the critical path with respect to the delay in the complete design and reduces delay by different measures such as transistor resizing, logic resynthesis, etc. The design is finally passed into layout synthesis. After timing optimization, logic synthesis is usually invoked again to resynthesize for smaller delays. This loop may be repeated several times. In case the timing obtained is not satisfactory, it may be necessary to return to the original V specification and change it manually.

Result Evaluation

Table 2.1 gives a comparison between a manual design and the automatic design. Here, the term "manual" refers to the fact that the register transfer level structural specification and the combinational logic were designed by human designers. Both used the YLE for logic synthesis. The complete design was done only in SCVS. In this example, structural synthesis yields results which are equal in performance to those of the manual design at the register transfer level. The automatic version was not optimized for timing, but the critical path in combinational logic has a delay of 72.9ns, which is almost equal to the delay in that of manual design. Timing optimization for the manual version yields a maximum delay of 49.9ns. The size regarding the number of transistors is 26% larger for the automatic design. This is caused by 45% more transistors in combinational logic and 11% more latches. However, the required CPU time for the automatic synthesis is much larger than for the manual design.

The differences between the manual design and automatic design are explained as follows.

Structural Design	SCVC Manual	SCVC Auto	CMOS Manual	CMOS Auto
Total number of latches	534	593	534	593
Comb. logic transistors	11164	16259	9896	15321
Total number of transistors	55066	69524	-	-
Execution cycles/instructions	1	1	1	1
Unoptimized comb. delay	72.7 ns	72.9 ns	-	-
Structural synthesis time	-	13303 sec	-	13303 sec
Logic synthesis time	1680 sec	13820 sec	3039 sec	14255 sec

Table 2.1: Comparison of Manual and Automatic Structural Design

- Structural synthesis does not yet minimize the number of interconnections. This not only leads to a large number of wires but may also hide some logic minimization potential.

- Automatic partitioning of pipeline stage P1 and P2 may lead to a lower optimization possibility during logic synthesis.

- In SCVS, the automatic design does not take advantage of distributing combinational logic among two clock phases, as was done in the manual version.

- In the case of large unoptimal logic design, such as generated by structural synthesis, logic synthesis may not reveal all the optimization potential.

2.2.4 Open Problem

There remain a number of open problems yet to be solved in the high-level synthesis area.

- *Human Factor*: This involves the role of the designer in the design process, such as, how the designer inputs design specifications and constraints, how the system outputs results, what decisions the designer should make and what information the designer needs in order to make them, and how the system explains to the user what is going during the design process.

- *Design Verification*: This refers to verify whether the design produced by the synthesis system satisfies the initial specification or not. In order to design an effective hardware, the process of manual improvement will remain. Formal verification is very important in such situations.

- *Integrating Levels of Design*: In order to make realistic evaluations of design tradeoffs at the algorithmic and the register transfer level, it is necessary to

be able to anticipate what the lower level tools will do. In the usual manual design process, many times of feedback over different design levels occur. This is the key point for an effective design. In contrast, the automatic synthesis is processed in one straight way, and this may degrade the performance of synthesized design.

In summary, high-level synthesis has been researched actively and many difficult problems have been solved. Much work, however, needs to be done before synthesis becomes really practical.

2.3 Hardware Description Language

In this section, ISPS and VHDL are introduced among a number of hardware description languages. These languages can specify the behaviors at the register transfer level and are used as an input to the high-level synthesis system as mentioned in section 2.2.

2.3.1 ISPS

ISPS is an abbreviation of Instruction Set Processor Specification and was developed at Carnegie-Mellon University under the sponsorship of DARPA (the Defense Advanced Research Projects Agency). ISPS is a register transfer language designed to support a wide range of application rather than a wide range of design levels. The language has many features for the behavioral modeling of hardware, but its capabilities for gate-level modeling are rather limited. The design philosophy of ISPS was guided by two principles: flexibility and simplicity. Specifically, it was desired, as mentioned in [Bar81], to design a computer description language that would be appropriate for many kinds of applications: automated design, simulation (for both software development and hardware debugging), and automatic genera-

tion of machine relative software (in particular, compiler-compilers). Thus, although ISPS can be viewed as a programming language, the aim of the notation is to describe computers and other digital systems, not necessarily general computational algorithms.

ISPS describes the *interface* (i.e. external structure) and the behavior of hardware units. The interface describes the number and the types of carriers (registers and memories) which are used to store and transmit information between the units. The behavioral aspects of the unit are described by procedures which specify the sequence of control and data operations in the machine.

A complete separation between the specification of the structure and the behavior of a digital system is not an easily realizable or even desirable goal. Thus, ISPS favors the behavioral aspects over the structural or implementation aspects. The structural information is never completely eliminated, for example, register lengths, data path widths, and connections of registers. Other details of the structure, such as component speed, layouts, physical location, integrated circuit technology, are not required to be specified in an ISPS description.

Time is modeled in implicit discrete units. A built-in procedure is available to specify transport delay units. The amount of delay must be expressed in terms of the basic time unit.

MC6502 in ISPS

In this paragraph, a description example of MC6502 in ISPS is presented instead of describing details of ISPS notations. For the detail notations, consult [Bar81]. Figure 2.4 indicates a part of the description. MC6502 is a processor of 8-bit width. This description is a benchmark of high-level synthesis(mentioned in section 2.2.1).

```
! ISPS Description of the MOS Technology MCS 6502 Microprocessor

**MP.STATE**
    MACRO romlow:= |"F800 |,
    MACRO romhi := |"FFFF |,
    MACRO ramlow:= |"0000 |,
    MACRO ramhi := |"1000 |,
    MACRO maxb  := |"FFFF |, ! High end of byte memory
    Mb[0:maxb]<7:0>, ! Primary memory range
    ram[ramlow:ramhi]<7:0> := mb[ramlow:ramhi]<7:0>, ! RAM
    rom[romlow:romhi]<7:0> := mb[romlow:romhi]<7:0> ! ROM

**PC.STATE**
    Pc<15:0>, ! Program counter
    Y<7:0>, ! Index register
    X<7:0>, ! Index register
    S<7:0>, ! Stack pointer
    Dl<7:0>, ! Input data latch
    A<7:0>, ! Accumulator
    Ir<7:0>, ! Instruction register
    P<7:0>, ! Processor status

**ADDRESS.CALCULATION**
    immed()<15:0> :=            ! Immediate
        BEGIN
        immed = ab = Pc NEXT
        Pc = Pc + 1
        END,
    abs()<15:0> :=              ! Absolute
        BEGIN
        abs = ab(Pc + 1, Pc) NEXT
        Pc = Pc + 2
        END,
    ab(adh.<15:0>, adl.<15:0>)<15:0> :=     ! Address buffer
        BEGIN
        ab<15:8> = read(adh.) NEXT
        ab<7:0>  = read(adl.)
        END,
! Read and write memory access routines
    read(ab.<15:0>)<7:0> :=      ! Read from valid memory
        BEGIN
        Rw = 1 NEXT
        IF NOT Ready => RESTART run NEXT
        read = "FF NEXT          ! Fake a nonexistant memory access
        IF (ab. GEQ{US} ramlow) AND (ab. LEQ{US} ramhi) => read = ram[ab.];
        IF (ab. GEQ{US} romlow) AND (ab. LEQ{US} romhi) => read = rom[ab.]
        END,
```

Figure 2.4: Behavior of MC6502 in ISPS

2.3.2 VHDL

Motivation

VHDL is an abbreviation of the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language. In the United States of America, the VHSIC program was launched. The goals of the program were to reduce IC design time and effectively insert VHSIC technology into military systems. These goals, indicating the need for a standard means of communication, motivated the development of a hardware description language.

Requirements for VHDL were analyzed over a period of two years, from 1981 to 1983, and originally defined in a workshop that was organized by the member of industry, academic community, and the Government. Version 7.2 of the language was released in 1985 and became a standard IEEE-1076 in December, 1987. From 1989, all the digital ASICs supplied to DoD are under obligation to be specified in VHDL.

Feature

One of the characteristics of hardware devices is that their functionality can be defined independent of the environment on which they operate. VHDL reflects this characteristic in its overall organization, emphasizing the ability to describe isolated components, called *design entities*, which can then be combined with other component descriptions to form more complex descriptions.

A design entity consists of an *interface* description and a *body* description. The interface defines the I/O ports through which the design entity communicates with the outside world. The body, on the other hand, describes the internal operation or organization of the component being described. The internal details of the component may be described using structural, data-flow, or procedural styles of expression.

In fact, these three styles can even be mixed in a single description.

Structural descriptions define *components* and then interconnection of these components using *signals*. Each interconnected component is then bound to another design entity in the library. The design entity to which a component is bound may itself have a structural body. In this way, hierarchical descriptions of a design may be built up. A design entity may bind its components to other lower-level design entities, but it never binds itself to higher-level components. This characteristic allows a design entity to be inserted into any description, just as a physical hardware component can be wired into any design.

To illustrate the concept of a design entity and the structural style of description, an RS-flip-flop which is implemented in terms of cross-coupled NAND gates is considered. One way of describing the implementation in VHDL is to write an interface description for the RS-flip-flop which declares two input ports R and S and two output ports Q and QB, and to write a body description which contains two cross-coupled NAND gates. Figure 2.5-(a) is the interface description and (b) is the body description.

VHDL provides several types of *concurrent signal assignment statements*. These statements operate asynchronously with respect to one another in an event-driven manner. Changes in the values of inputs to a given signal assignment statement cause the statement to execute. This execution involves calculating new output values and assigning those values, after some delay, to the output signals of the statement.

As an example of the use of concurrent signal assignment, an alternative body description is shown in Figure 2.5. In this body, a pair of concurrent signal assignment statements take the place of the component instantiations G1 and G2. These signal assignments show the cross-coupling of the two NAND gates, like the

structural description,

In addition to the structural and data-flow styles, VHDL also provides for procedural description in which the behavior of a device is modeled without regard to its structure. Such descriptions take the form of algorithms for computing output responses to input changes. This style of description has the advantage of describing behavior without containing structural information.

Application Area

VHDL allows design and documentation of digital circuits from the system level down to the gate level. In addition, the language has the necessary syntactic framework for describing circuit geometries. This includes schematic data and layout data for hardware devices.

The underlying dynamic execution model defined by the language is event-driven. An event may cause execution of one or more processes. The execution of a process may cause further events to be scheduled. This execution cycle will continue until there are no events remaining to be processed. The above event-driven model is suitable for the simulation of digital circuits from the system level down to the gate level.

2.4 Discussions in the Survey

There has been a strong requirement of design assistance at higher levels of the design hierarchy. High-level synthesis is one of the answers to these requirements.

Although its results seem to be efficient, there still remain several problems to be solved as mentioned in 2.2.4. One of the most essential problems is that there exist few methods for treating feedbacks spreading over different design levels. This is itemized as *Integrating Levels of Design* in section 2.2.4. The process of improvements is the key point for an effective design, but with the current approach,

```
entity RS_Latch
(
 R,S : in BIT;     -- input ports
 Q,QB : out Bit  -- output ports
)
is
end RS_Latch
```

(a) the interface description

```
architecture Body1 of RS_Latch is
  B: block
        component  NAND_Gate
           port (A, B : in Bit;  C: out  Bit);

      begin
        G1: NAND_Gate port (S, QB, Q);
        G2: NAND_Gate port (R, Q, QB);
      end block;
end Body1;
```

(b) the body description

```
architecture Body2 of RS_Latch is
  B: block
      begin
        Q <= S nand QB after 10ns;
        QB <= R nand Q after 10ns;
      end block
end Body2;
```

(c) an alternative body description

Figure 2.5: RS Flip-Flop in VHDL

high-level synthesis hardly assists it.

So, an assistance system of another approach is suggested. The essential idea of this system is to make use designers' expert knowledge positively.

Hardware description language always plays a central role in CAD tools. In the proposed system, temporal logic is adopted as a behavioral description language. ISPS has enough power as a behavioral description language at the register transfer level, but has rather poor power at the gate level. The reason for this is that ISPS does not explicitly declare timing relations such as sequentiality and concurrency. On the contrary, temporal logic has clear semantics and enough ability to specify such relations. VHDL also has sufficient power as a behavioral description language at these levels. The event-driven model which underlies in VHDL and also is utilized in petri-net is suitable for specifying sequentiality and concurrency. Behaviors of asynchronous circuits can be specified more easily in event-driven model than in temporal logic. However, temporal logic can declare the aspects of *next* clock explicitly, whereas event-driven model cannot. This nature is important as a behavioral description language at the register transfer level and is convenient for specifying synchronous circuits. Some researches have been reported in practice which extend petri nets to "timed petri nets" [Zub80] or "Temporal petri nets" [SL89]. With these extended petri nets, timing requirements on the components of a digital system can be specified. In addition to that, VHDL has too wide varieties as a hardware description languages, which is often the case with standard languages. So, VHDL itself cannot be adopted directly as a core language in CAD tools. The first task for researchers is to define a subset of VHDL when they intend to use VHDL in their CAD tools. Some attempts to define a standard subset of VHDL have been made.

Therefore, temporal logic is adopted as a core language to describe behaviors in our system. The proposed system is presented in the remain of this thesis.

Chapter 3

Hardware Logic Design Assistance System

3.1 Objective

The objective of the proposed system is to assist an *effective* logic design. The approach of this system is not to design automatically but to introduce integrated knowledge or experience of designers positively. The target of the system is an *assistance* both at the function design level and at the gate design level. Designers can utilize their techniques in this system. Design verification becomes very important when such an approach has been adopted.

In this system, a logic design flow is regarded as follows. At first, the designers specify the algorithm of the behavior to be designed. Then, the register transfer level description is derived from this abstract form step by step with a system realization. This derivation is assured by simulating the behaviors.

The designers also give the structure of the data path to be designed. Although there have been many reports on synthesizing the data path from the behavioral description [MPC88][PPM86][PK86][Cam88], the derived data paths are not yet completely satisfactory and are difficult to be improved.

Usually, the behavior is refined along with the construction of the data path

structure. One of the most typical cases is to divide the control part from the data path part. This border line is decided while the behavior is refined, and of course, the data path structure is affected by this decision. In other words, the designers are always linking the behavior to the structure when they refine the behavior step by step.

This link information is the key for an effective synthesis. Particularly, this information is very important in the following cases:

- To improve the derived behavior and structure.
- To re-use the already designed structure for a new design.

Therefore, in our system, the structure as well as the behavior is assumed to be given by the designers, and the following assistances are realized.

- Link informations between the behavior and the structure are derived automatically.
- Consistency between the behavior and the structure is verified automatically.
- The logic of the control part is derived automatically.

Even though a good data path can be synthesized from the behavior in the near future, the process of improvement will still have to be done manually. Therefore, these assistances are still very important.

Then, the derived logic of the control part is transformed into networks of logical gates or micro-programs by manual or by using a commercial synthesizer. The obtained design should be verified formally.

As for the behavioral description language, the following three points are important in our design flow.

- The behavior is specified in an executable form.

- The behaviors at both the algorithmic level and the register transfer level are given in the same language.

- Sequentiality and concurrency can be specified accurately and simply.

Since Tokio is based on interval temporal logic [Mos83], its semantics is very clear. Due to this nature, Tokio meets these three requirements and therefore, provides a smooth assistance of register level synthesis.

3.2 Structure

The structure of the proposed assistance system is as shown in Figure 3.1.

The designers, at first, specify the algorithm of the behavior in Tokio. Then, the register transfer level description is derived. Currently, this derivation is verified by the simulator [KAFT85] which has been already developed. The register transfer level description is specified in RTL-Tokio (Register Transfer Level Tokio) which is a constrained form of Tokio. Both behaviors in Tokio and RTL-Tokio can be simulated on the same simulator after being compiled into Prolog. The designers also give the structure of the data path using a graphic editor. The part of translating the structural description into Prolog format remains to be developed.

Then the 'Data Path Verifier' verifies the consistency between the behavioral and the structural descriptions. After the verification has finished, the structure and the behavior are evaluated. If the design is satisfactory, the design process proceeds to a lower level synthesis. Otherwise, the design is improved manually and then verified again. In the process of verification, the *state transition table* and the *facility usage table* are derived. Facility usage table indicates link information between the behavior and the structure, such as what component in the structure is used in order to realize a certain data transfer in the behavior. The control part is

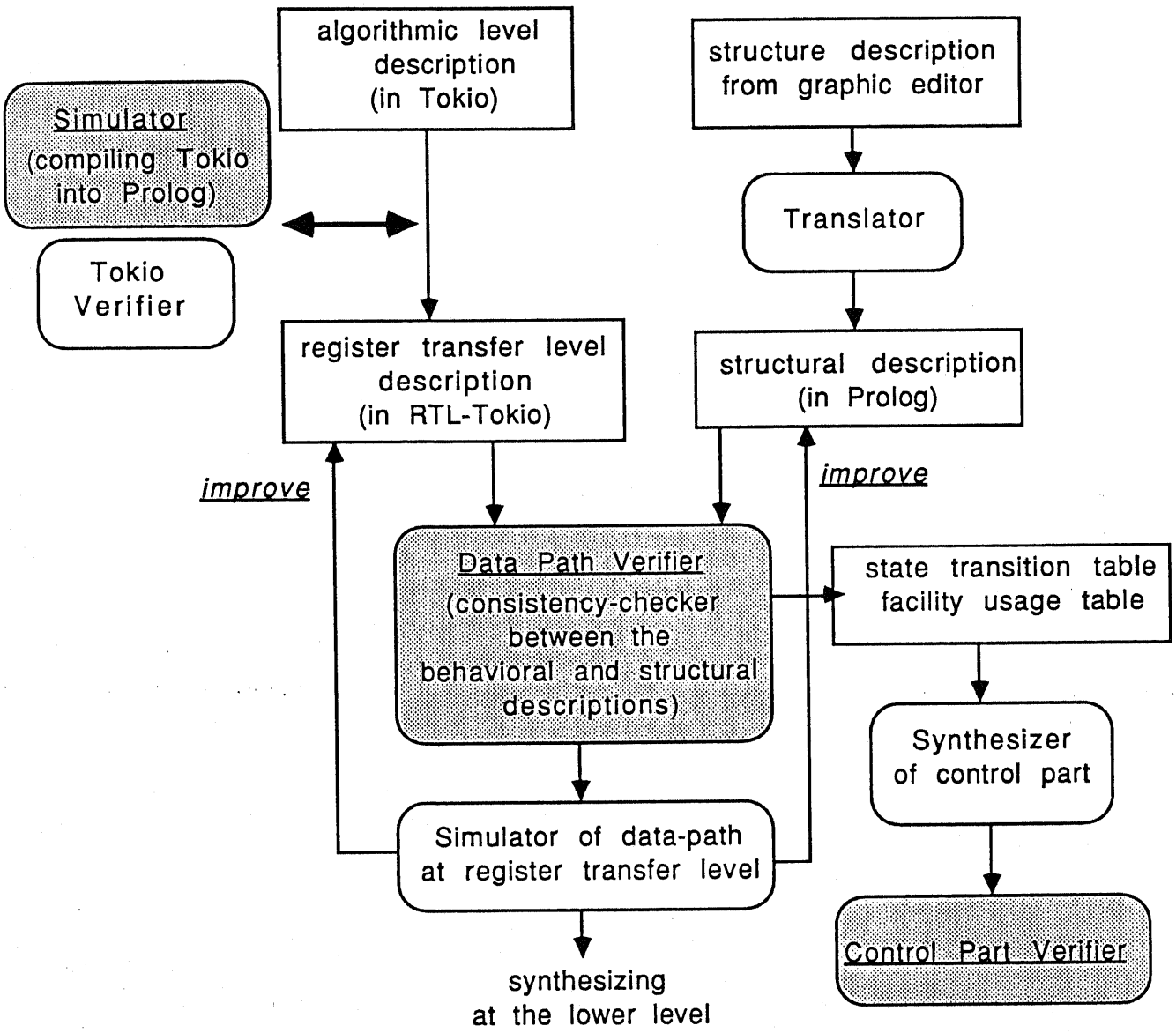


Figure 3.1: Structure of Assistance System

synthesized from these two tables. As for the control part, this system is connected to the verification system [NFKT87].

Chapter 4

Tokio as a Hardware Description Language

4.1 Temporal Logic as a Hardware Description Language

In this section, temporal logic is introduced and it is explained how to specify hardwares with it. While traditional logic uses such operators as $\sim, \wedge, \vee, \rightarrow$, etc., temporal logic introduces additional operators for dealing with temporal sequences.

An expression of traditional logic is assumed to specify properties of the system at a given state called the "present" state. Temporal logic can describe properties of all possible execution sequences that may evolve from the present system state.

Temporal logic has become very popular in the CAD fields [BCDM86] [CES86] [FTM85] [NFKT87] [Aba87] [MP81] [Wol82].

The reasons are as follows.

- Temporal logic can easily and accurately express the timing relations such as the concurrency and the sequentiality.
- Since temporal logic is based on formal theory, assistance, such as verification or synthesis, is easily implemented on computers.

Many kinds of temporal logic have been proposed differing from each other slightly. Most of them are defined not on continuous states but on discrete states.

In this section, Linear Time Temporal Logic (LTTL)[MP81, Wol82] and Interval Temporal Logic (ITL)[Mos83] are introduced. These logics are defined on discrete states, and therefore, there exists the next state for each state.

4.1.1 LTTL

Linear Time Temporal Logic (LTTL)[MP81][Man81] have four temporal operators: $\circ$ (next), $\Box$ (always), $\Diamond$ (sometime), and **U** (until). The first three are unary operators and the last is a binary operator. Meanings of each temporal operator are as follows.

- P (without temporal operators): P is true at current state.
- $\circ$P: P is true at the next state.
- $\Box$P: P is true in all future states.
- $\Diamond$P: P is true in some future state.
- P **U** Q: P is true for all states until the first state where Q is true.

Temporal operators are defined on discrete states. The strict definitions of the above operators are introduced in the following way.

Let s_0 be the present state and $s_1, s_2, \cdots$ be the states of the future in order. For an ω-sequence of states $\sigma = s_0, s_1, s_2, s_3, \cdots$, where each state is an assignment of truth values to the atomic propositions. The i-truncated suffix of σ, is denoted by $\sigma^i = s_i, s_{i+1}, \cdots$. Then, the followings are obtained.

- $\sigma \models f$ iff $s_0 \models f$ where f is an atomic proposition
- $\sigma \models \sim f$ iff not $\sigma \models f$
- $\sigma \models f_1 \wedge f_2$ iff $\sigma \models f_1$ and $\sigma \models f_2$

- $\sigma \models \circ f$ iff $\sigma^1 \models f$
- $\sigma \models \Box f$ iff $(\forall i \geq 0)(\sigma^i \models f)$
- $\sigma \models \Diamond f$ iff $(\exists i \geq 0)(\sigma^i \models f)$
- $\sigma \models f_1 \mathbf{U} f_2$ iff $(\forall i \geq 0)(\sigma^i \models f_1)$ or $(\exists i \geq 0)(\sigma^i \models f_2 \wedge \forall j(0 \leq j < i \rightarrow \sigma^j \models f_1))$

Notice that the **U** operator defined here (known as the "weak" until) does not have an eventuality component, in contrast with the one defined in [Man81]. The two temporal logic systems have the same expressive power.

An axiomatic system for LTTL is as follows [Man81]:

Axiom Schemas:

- $\vdash \Diamond p \equiv \sim \Box \sim p$
- $\vdash \Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$
- $\vdash \circ \sim p \equiv \sim \circ p$
- $\vdash \circ(p \rightarrow q) \rightarrow (\circ p \rightarrow \circ q)$
- $\vdash \Box p \rightarrow p \wedge \circ p \wedge \circ \Box p$
- $\vdash \Box(p \rightarrow \circ q) \rightarrow (p \rightarrow \Box p)$
- $\vdash \Box p \rightarrow p \mathbf{U} q$
- $\vdash p \mathbf{U} q \equiv q \vee (p \wedge \circ(p \mathbf{U} q))$

Inference Rules:

- If ω is a tautology, then $\vdash \omega$
- If $\vdash (\omega_1 \rightarrow \omega_2)$ and $\vdash \omega_1$ then $\vdash \omega_2$

- If $\vdash \omega$, then $\vdash \Box\omega$

LTTL is proved to be as expressive as the first order theory of linear order [MP81]. This has caused LTTL to be called "expressively complete" and would seem to imply that LTTL is perfectly adequate for expressing desirable properties of an execution sequence.

$\Box$ and $\Diamond$ are acquired from **U** as follows.

- $\Box f = f \ \mathbf{U} \ \mathit{false}$
- $\Diamond f = \sim (\sim f \ \mathbf{U} \ \mathit{false})$

Therefore, only **U** and $\circ$ operators are essential. However, $\Box$, $\Diamond$ operators are introduced for the ease understandings of the descriptions. $\Box$ and $\Diamond$ operators have the following properties, which are acquired from the axioms and inference rules mentioned above.

- $\Box\Box f = \Box f$
- $\Diamond\Diamond f = \Diamond f$
- $\Box\Diamond\Box f = \Diamond\Box f$
- $\Diamond\Box\Diamond f = \Box\Diamond f$

So, only $\Box\Diamond$ and $\Diamond\Box$ of combined $\Box$ and $\Diamond$ operators have different meanings from $\Box$ and $\Diamond$ operators. For example, the expression:

$$B \rightarrow \Box\Diamond A$$

means that if B is true at present, then at every future point in time, A will be true at some point following that point and, thus, A will be true infinitely often, possibly without change. And the expression:

$B \rightarrow \Diamond\Box A$

means that if B is true at present, then at some future point in time, A will be true at every future point following that point.

As a whole, systems are divided into two properties: *safeness* and *liveness* properties. Safeness property means "bad things never happen", for example, "deadlock will not happen", etc.. This can be described in LTTL like:

$\Box$("adequate initial condition" $\rightarrow \Box \sim$ "bad thing")

or

$\Box$("adequate initial condition" $\rightarrow \Diamond\Box \sim$ "bad thing")

On the other hand, liveness property means "good things eventually happen", for example, "the calculation will eventually terminate" etc.. This can be described in LTTL like:

$\Box$("adequate initial condition" $\rightarrow \Diamond$ "good thing")

or

$\Box$("adequate initial condition" $\rightarrow \Box\Diamond$ "good thing")

4.1.2 ITL

Interval Temporal Logic (ITL) was proposed by Moszkowski [Mos83]. The differences between ITL and LTTL are represented as follows. First, ITL is based upon "interval"s, which are successive states of finite length. The truth of variables or formulas depends not on states but on intervals. Second, ITL has the operator ";" (read semicolon) to divide an interval into two subintervals as shown in Figure 4.1.

A model in ITL is defines as a pair of Σ, M. consisting of a set of states $\Sigma = s, t, ...$ together with an interpretation M. M maps a propositional variable P and

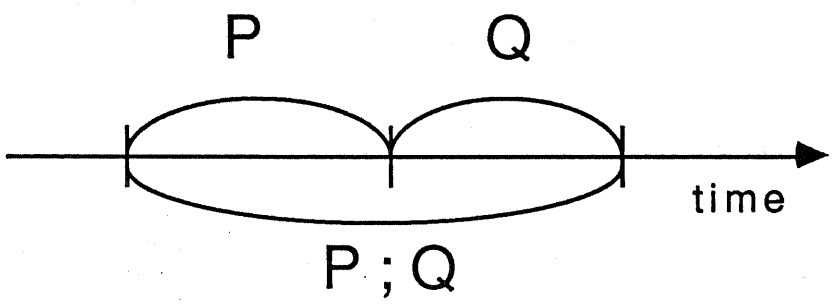


Figure 4.1: Temporal Operator ";"

nonempty interval $s_0, s_1, ..., s_n \in \Sigma^+$ to a some truth value $M_{s_0,...,s_n}[P]$. The length of an interval $s_0, s_1, ..., s_n$ is n.

Interpretation of Formulas

Fundamental operators in ITL are ";" (chop) and o (next) operator. Suppose w,$w1$, and $w2$ be formulas. Then interpretation M is given as follows.

- $M_{s_0,...,s_n}[\sim\ w] = true$ iff $M_{s_0,...,s_n}[w] = false$
- $M_{s_0,...,s_n}[w1\ \wedge\ w2] = true$ iff $M_{s_0,...,s_n}[w1] = true$ and $M_{s_0,...,s_n}[w2] = true$
- $M_{s_0,...,s_n}[\circ\ w] = true$ iff $n \geq 1 \wedge M_{s_1,...,s_n}[w] = true$
- $M_{s_0,...,s_n}[w1\ ;\ w2] = true$ iff for $0 \leq \exists i \leq n$, $M_{s_0,...,s_i}[w1] = true \wedge M_{s_i,...,s_n}[w2] = true$

All the other operators in LTTL can be represented by using ";" and o. For example, $\Diamond$ operator is expressed in ITL as follows.

$$\Diamond f \equiv (true; f)$$

Some operators which are useful to describe properties of hardware are defined in terms of ";" and o. The beginning and ending states of an interval, *beg* and *fin*, are defined as follows.

- $beg(P) \stackrel{\text{def}}{=} ((empty \wedge P); true)$
- $fin(P) \stackrel{\text{def}}{=} (true; (empty \wedge P))$ where $empty \stackrel{\text{def}}{=} \circ\ fail$

"*empty*" indicates an interval with 0 length, that is, an interval with only one state.

Register transfer statements in HDLs or assignment statements in programming languages are described with beg and fin in first order ITL.

$$A \leftarrow B \stackrel{\text{def}}{=} \forall c.(beg(B = c) \rightarrow fin(A = c))$$

4.2 Tokio

4.2.1 Logic of Tokio

Tokio [KAFT85][AFT85][FKTM86] is a temporal logic programming language which is based on **LITL** (Local Interval Temporal Logic).

In non local ITL, the truth of a variable is determined in the interval. On the contrary, the truth of a variable is determined at the beginning state of the interval in LITL. Therefore, the following formula succeeds in LITL.

$$M_{s_0,\ldots,s_n}[P] = M_{s_0}[P]$$

ITL is not decidable even in propositional logic, whereas propositional LITL is decidable. Therefore LITL is calculated more easily in computer than ITL is.

As for the decidability, LTTL and LITL are identical. However, it is much easier to describe sequentiality in LITL than in LTTL. For example, suppose the case to describe the formula that

"first execute P and then execute Q".

This formula is specified as "$(P;Q)$" in LITL. In contrast, all the following things should be declared in LTTL.

- First P is executed.
- All the states when P is executed, Q is not executed.
- If P is <u>ended</u>, <u>then</u> Q is started to execute.
- All the states when Q is executed, P is not executed.

Underlined part "ended, then" is rather cumbersome to describe compactly. This is described in LTTL as

$$\Box((P \wedge \circ \sim P) \rightarrow (\sim Q \wedge \circ Q))$$

or

$$\Box(\sim P \vee \circ P \vee (P \wedge \sim Q \wedge \circ \sim P \wedge \circ Q)).$$

4.2.2 Operators in Tokio

Tokio is regarded as an extension of Prolog with temporal operators intuitively. The difference between Tokio and Prolog is characterized by whether they have the notion of the time sequence or not. In Tokio, several operators are defined and the logical variables are handled in a different way from Prolog.

Tokio Operator

Tokio is based on Local Interval Temporal Logic and the temporal operators are defined on the intervals.

- without temporal operator: P :- Q,R.

  In the above expression, the goals Q and R is executed in the same interval where the predicate P is defined as shown in Figure 4.2. Therefore the concurrency is represented here.

- chop (&&): P :- Q && R.

  The chop operator && specifies the sequential execution of the two goals Q and R. As shown in Figure 4.3, the chop operator divides the interval Ip is defined into the two subintervals Iq and Ir. The goal Q is executed in Iq and R is executed in Ir.

- next (@): P :- @Q.

  This operator @ specifies the execution in the next interval. The goal Q is executed in the next interval Iq of the interval Ip. The relation between Ip

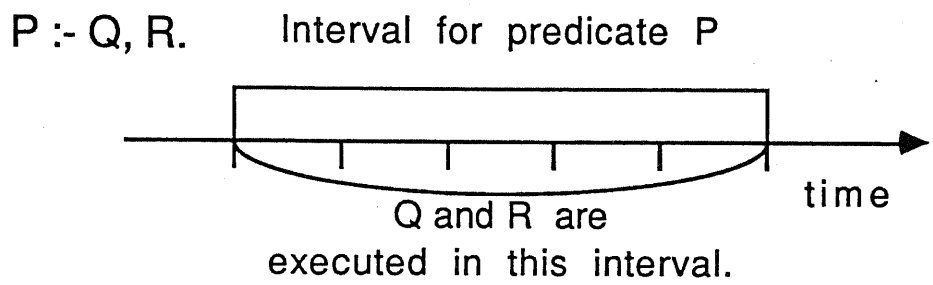


Figure 4.2: Without Temporal Operators

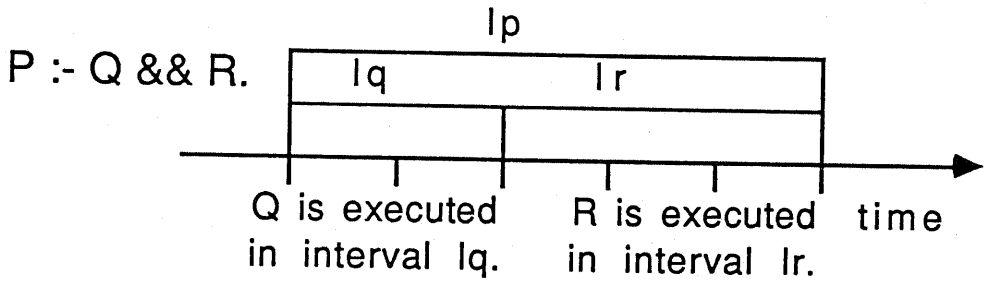


Figure 4.3: Chop Operator

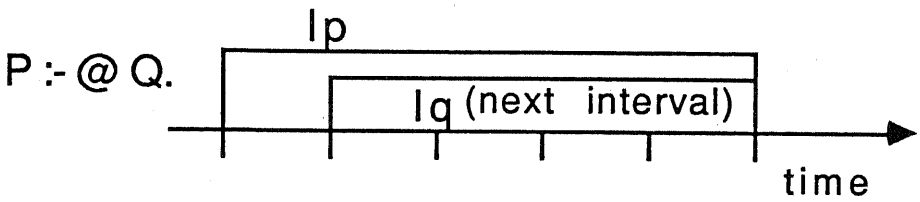


Figure 4.4: Next Operator

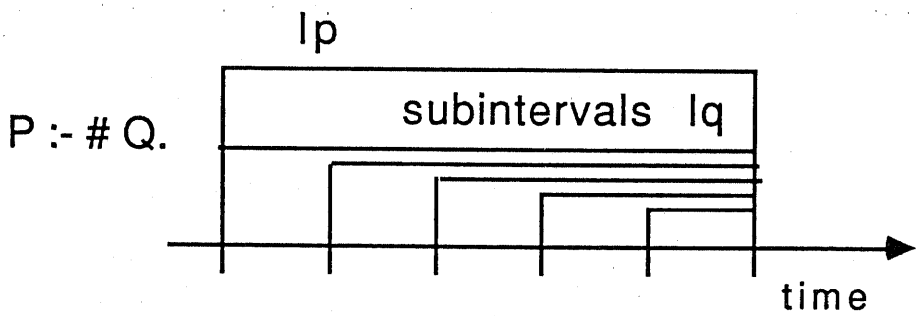


Figure 4.5: Always Operator

and Iq is as shown in Figure 4.4.

- always (#): P :- #Q.

  The always operator # indicates that the goal Q is executed in the all subintervals Iq. The relation between Ip and Iq is shown in Figure 4.5.

Tokio Variable

Tokio variables may have different values at each time(clock) in contrast that the values of Prolog variables do not change. In other words, one Tokio variable is a sequence of Prolog variables along the time sequence.

There are two kinds of variables in Tokio, which are the *local variables* and *global variables.* A global variable holds the value unless a new assignment to that variable occurs, whereas the value of a local variable varies along with the time sequence. As for the syntax, the name of a local variable begins with a capital letter and that of a global variable begins with a character "*". In Tokio, there are two kinds of assignments to the variables.

- temporal assignment:

  X <– Y (for local variables),

  *x <= *y (for global variables)

  This assignment is defined on the interval. The values of the variables Y or *y at the beginning of the current interval are assigned to the variables X or *x at the end of the interval as shown in Figure 4.6.

- immediate assignment:

  X = Y (for local variables),

  x := *y (for global variables)

  This assignment is defined on the discrete time like Figure 4.7.

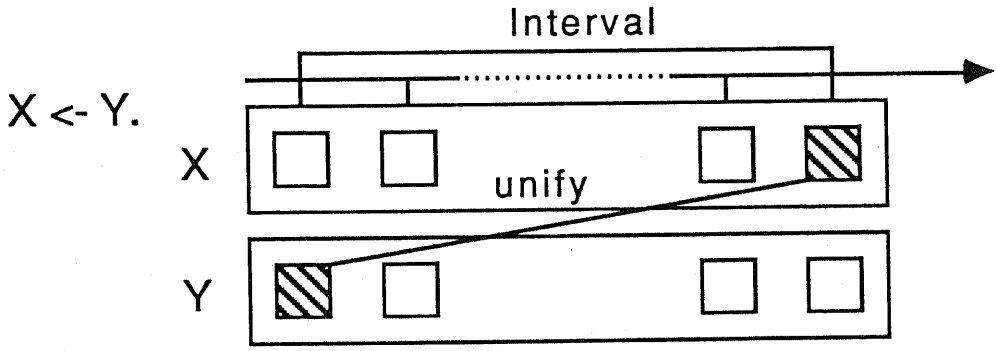


Figure 4.6: Temporal Assignment

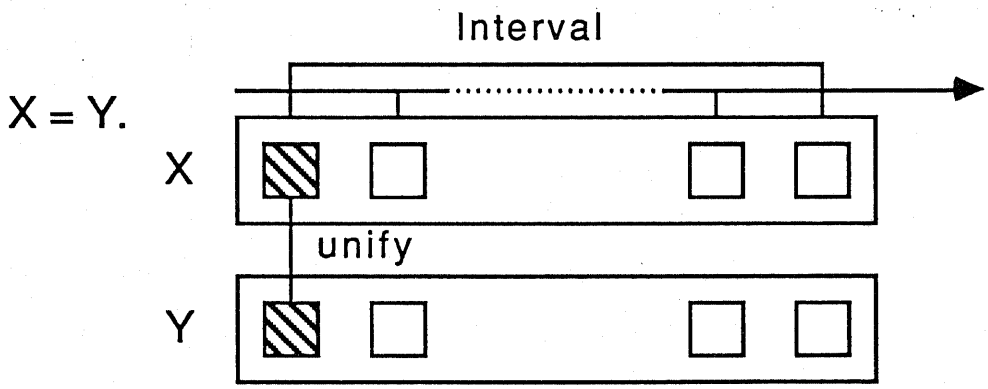


Figure 4.7: Immediate Assignment

4.3 RTL-Tokio

Tokio is defined on the discrete time sequence. The semantics of Tokio depends on what "the discrete time" in Tokio represents in the hardware. If "the discrete time" represents the clock of flip-flops in the hardware, the description in Tokio corresponds to a synchronous circuit. If "the discrete time" represents the absolute time, the description corresponds to an asynchronous action within one clock. Due to this flexibility, Tokio can describe the behavior at different levels.

RTL-Tokio is a register transfer level hardware description language and a constrained form of Tokio. The discrete time in RTL-Tokio indicates the machine cycle. The variables and backtrackings are constrained in RTL-Tokio as follows.

Variables

The global variables which hold the values represent the data of the registers or memories, and the local variables also represent these data indirectly. The current assignment of the global variables such as "`*q := *p`" is prohibited in RTL-Tokio because no transfers between registers or memories can be done immediately. Both the global and the local variables cannot stand for list structured variables.

Backtracking

Tokio has two kinds of backtrackings. One is the backtracking which occurs usually in Prolog and the other is the backtracking of time sequence. Since the behavior of hardware is decidable, RTL-Tokio is restrained from these backtrackings. In order to prevent RTL-Tokio from the former backtracking, the syntax of RTL-Tokio is constrained as follows,

```
head(arg) :- localCond, !, recursiveCall.
head(arg) :- localCond, !, actions && actions && ... && recursiveCall.
```

"`localCond`" should be estimated at the beginning of the interval and the number of localCond is more than or equal to 0. "`actions`" includes no recursive predicate calls, and "`recursiveCall`" may include them. Consequently, only tail recursions (that is, loop structures) are to be permitted in RTL-Tokio.

The latter backtracking (that is, the backtracking of the time sequence) occurs owing to the undecidability in determining the length of each interval. The length of intervals in Tokio is determined to be more than or equal to 1. (Details are described in [KAFT85]). In RTL-Tokio, the length of each interval is decided so that every temporal assignment is executed in the interval of length 1. This constraint represents that the data transfers between registers are executed in one machine cycle.

4.4 Derivation of RTL-Tokio from Tokio

Transforming Tokio into RTL-Tokio is to derive a behavior at the register transfer level. This process includes operation scheduling, operation allocation, register allocation, and so on. This derivation is currently done manually. An interactively automatic derivation using the method of high-level synthesis [MPC88] is an ideal design assistance. This remains to be solved.

In this subsection, it is shown that Tokio/RTL-Tokio has enough power as a behavioral description language by illustrating several examples.

Figure 4.8-(a) shows a simple program that computes the square root using Newton's method (cited from [MPC88]). Figure 4.8-(b) is the same algorithm as (a) in Tokio. In the description (b), neither operation schedulings nor register allocation have been processed yet. What is found is that some adders and multipliers are required.

Next, the description (c) is derived. Here, the operation "`Y <- (Y + X / Y) /`

```
Y := 0.222222 + 0.888889 * X;
C := 0;
    DO UNTIL C = 2 LOOP
    Y := (Y + X / Y) / 2;
    C := C + 1;
    ENDDO;
```

(a) Algorithm of Newton's Method

```
main(X) :-
    Y <-  0.222222 + 0.888889 * X, X <- X && sub(X,Y,0).
sub(X,Y,C) :- C = 2, !.
sub(X,Y,C) :- !, Y <- (Y + X / Y) / 2, C <- C + 1, X <- X
    && sub(X,Y,C).
```

(b) Algorithm in Tokio

```
sub(X,Y,C) :- C = 2, !.
sub(X,Y,C) :- !,
    ( Tmp <- Y + X / Y, X <- X && Y <- Tmp / 2, X <- X), C <- C + 1
    && sub(X,Y,CC).
```

(c) Partially scheduled description in Tokio

```
main :-
    *y <-  0.222222 + 0.888889 * *x , *cnt <= 0 && sub.
sub :- *cnt = 2, !.
sub :- !, *y <= (*y + *x / *y)
    && *y <= *y / 2, *cnt <= *cnt + 1
    && sub.
```

(d) Behavior at the register transfer level in RTL-Tokio

```
main :- *adr = 8, !, true.
main :- !, input && stage1 && main, (stage2 && true).
input :- !, *input1 <= *memory(*adr), *adr <= *adr + 1 &&
    *reg1 <= 0.222222 + 0.888889 * *input1.
stage1 :- !, *reg1 <= *input1 / *reg1 + *reg1 &&
    *reg2 <= *reg1 / 2, *input3 <= *input1.
stage2 :- !, *reg2 <= *input3 / *reg2 + *reg2 &&
    *output <= *reg2 / 2.
```

(e) Pipelined behavior in RTL-Tokio

Figure 4.8: Description Examples in Tokio

2" is scheduled and divided into the two steps "`Tmp <- Y + X / Y`" and "`Y <- Tmp / 2`. In this stage, the operation "`C <- C + 1`" has not been scheduled yet. Since both the sequentiality and the concurrency are declared clearly in Tokio, partially scheduled behavior like Figure 4.8-(c) is also able to be simulated. This is one of the most outstanding characters of Tokio. Due to this nature, smooth design assistance can be achieved.

Then, the description (d) is derived along with the register allocation and operation scheduling. Registers are denoted as global variables in RTL-Tokio. In the description (d), the register `cnt` controls the repetition times of the loop. Another candidate of the derivation is to delete the register `cnt` and control the loop by the control part. It is natural, for example, in the case this computation is processed in pipeline. Pipelined behavior is as shown in Figure 4.8-(e). In this description, the loop is decomposed into "`stage1`" and "`stage2`". The separation between the data path and the control part is represented like this in RTL-Tokio. The second line of Figure 4.8-(e)
"`main, (stage2 && true)`" indicates that "`stage2`" and the next pipeline "`main`" are started up concurrently. "`true`" is a built-in predicate which always succeeds and whose length is not decided.

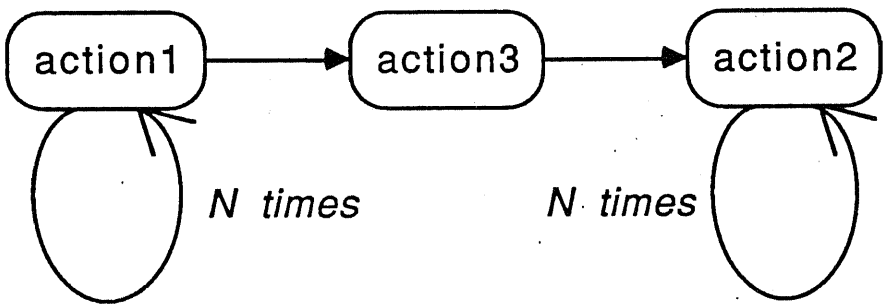
recursive call

In RTL-Tokio, only tail recursion is permitted among recursive structures. A simple example is shown in Figure 4.9. Figure 4.9-(b) denotes state diagram derived from (a). As found easily, implicit controller like a stack exists in the description (a) which counts how many times `action1` is executed. The controller should be declared explicitly as the behavioral description at the register transfer level. If one more register is added up, the description (c) is derived.

The constraint that only tail recursion is permitted represents that all the con-

```
pred :- cond, !, action1 && pred && action2.
pred :- !, action3.
```

(a) recursive call



(b) corresponding state diagram

```
% *reg is set to '0' initially
pred :- cond, !, action1, *reg <= *reg + 1 && pred.
pred :- !, action3 && pred_tail.
pred_tail :- *reg > 0, !, action2, *reg <= *reg - 1 && pred_tail.
pred_tail :- !, true.
```

(c) tail recursion

Figure 4.9: Recursion in RTL-Tokio

trollers should be declared explicitly. This is tightly coupled with the problem of separating control part from data path part. This constraint is natural for the behavior at the register transfer level.

Chapter 5

Verification of Control Part

In this chapter, a verification system of control part is presented. This is called a control verifier in Figure 3.1. Verification method adopted here is called as "proof checker". First, an assertion which the correct design should satisfy is given. This assertion is regarded as a part of the required specification. Then, the actual design is formally and automatically verified whether it satisfies the given assertion or not. This verification is basen upon the decision procedures of temporal logic.

5.1 Structure

The structure of the verification system is shown in Figure 5.1.

The input to this system are specification in propositional LTTL [MP81] and structural description in HSL [SKS80]. HSL is a structural description language that indicates networks between logical gates. A formula in propositional LTTL is translated into state diagram. The part of this translation is implemented in Prolog. All the other parts are implemented in C language. Since propositional LTTL has decision procedures as mentioned in section 4.2.1, the verification is processed automatically. Boolean expressions are handled in the cover (sum-of-product form).

Details of the verification methods are described in the next section 5.2 and

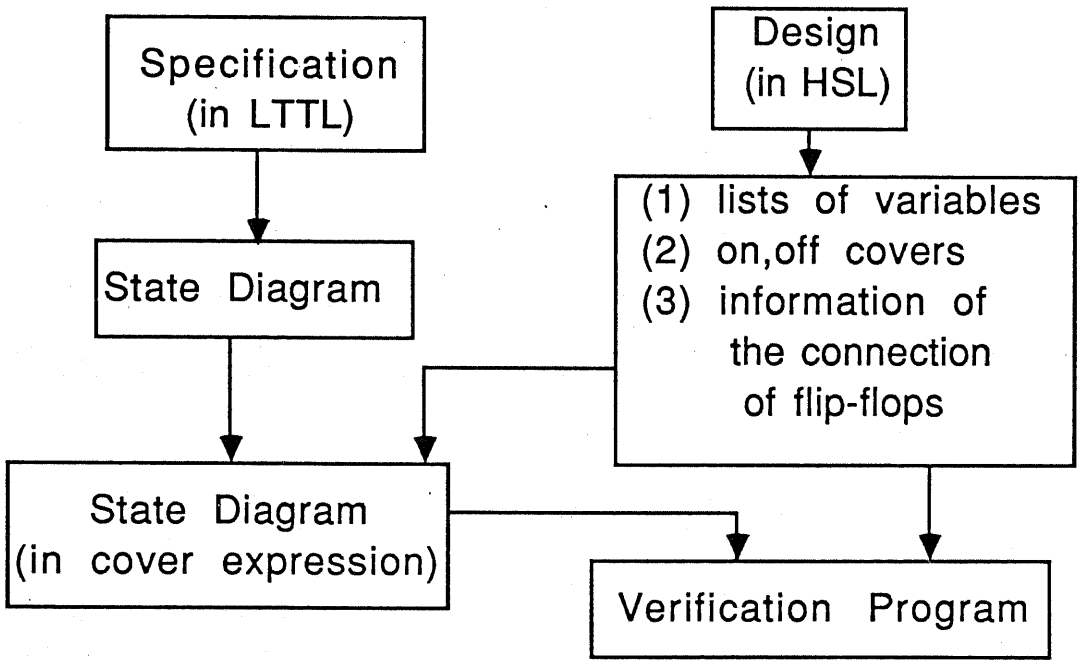


Figure 5.1: Structure of Control Part Verifier

experimental results are shown in section 5.3.

5.2 Verification Method

5.2.1 Translating LTTL Formula into State Diagram

The method to translate LTTL formulas into state diagrams is presented. The basic idea of this method is that LTTL formula can be decomposed into sets containing formulas which are either atomic (that is, without temporal operators) or that have $\circ$ as their main operator. The atomic sets are transitive conditions and the rest excluding outermost $\circ$ operator are conditions in next state (details are described in [Wol82]). Decompositions are repeated until all the newly obtained states are the same as those conditions which have been already produced.

The decomposition rules are as follows.

- $\Box F = F \wedge \circ\Box F$
- $\Diamond F = F \vee (\sim F \wedge \circ\Diamond F\{F\})$
- $F1\ U\ F2 = F2 \vee (F1 \wedge \sim F2 \wedge \circ(F1\ U\ F2))$

For example, let P, Q and R are atomic and translate

(A) $\Box P$

(B) $\sim ((P \wedge \circ\Box Q) \rightarrow \circ\Box R)$

into state diagrams using above rules. It goes as follows;

(A) $\Box P = P \wedge \circ\underline{\Box P}$

Since the condition in the next state '$\Box P$' (underlined) is the same as the condition at the current state, the decomposition is completed and the corresponding state diagram is obtained as shown in Figure 5.2.

The translation of (B) goes;

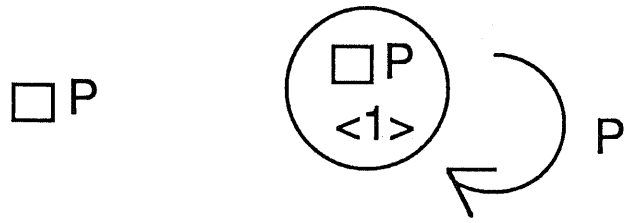


Figure 5.2: State Diagram (1)

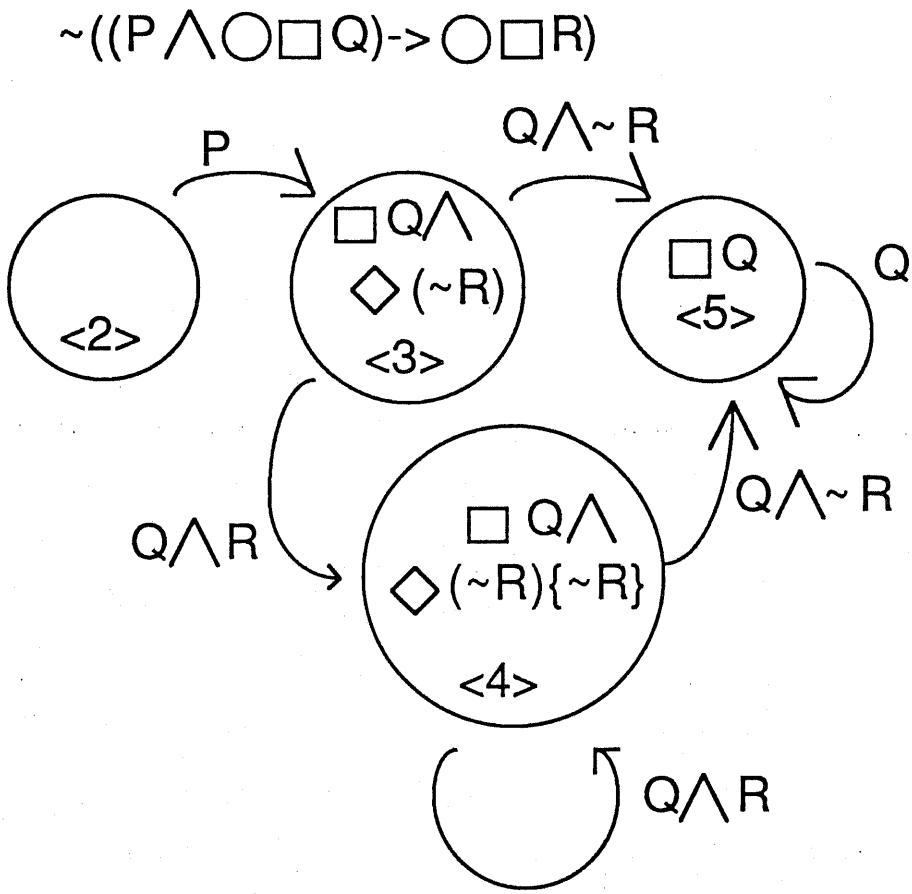


Figure 5.3: State Diagram (2)

(B) $\sim ((P \wedge \circ\Box Q) \rightarrow \circ\Box R)$

$= (P \wedge \circ\Box Q \wedge \sim (\circ\Box R))$

$= (P \wedge \circ\Box Q \wedge \circ\Diamond(\sim R))$

$= (P \wedge \circ\underline{(\Box Q \wedge \Diamond(\sim R))}$

$(\Box Q \wedge \Diamond(\sim R))$ is the condition in the next state and decomposed as follows.

$$(\Box Q \wedge \Diamond(\sim R))$$
$$= Q \wedge \circ\Box Q \wedge (\sim R \vee (R \wedge \circ\Diamond(\sim R)\{\sim R\})$$
$$= (Q \wedge \sim R \wedge \circ\Box Q) \vee (Q \wedge R \wedge \circ(\Box Q \wedge \Diamond(\sim R)\{\sim R\})$$

Therefore, the corresponding state diagram is as shown in Figure 5.3.

Satisfiability

A LTTL formula is satisfiable iff it has at least one infinite sequence of state transitions when it is translated into a state diagram.

The logic formula (A) is satisfiable because it has an infinite sequence of state transitions $< 1 >, < 1 >, < 1 >, \cdots$ in Figure 5.2. Similarly, the logic formula (B) is satisfiable because it has an infinite sequence of state transitions $< 5 >, < 5 >, \cdots$ in Figure 5.3. Here, the sequence $< 4 >, < 4 >, \cdots$ is not infinite because it does not satisfy the eventuality $\{\sim R\}$. The eventuality $\{P\}$ means that P must eventually be true in all the sequences of future states which follow the state P. Since '$\sim R$' is never true in the sequence $< 4 >, < 4 >, \cdots$, this sequence cannot be infinite.

The satisfiability of products of some logic formulas is easily checked by tracing each state diagram concurrently. For example, the satisfiability of the next formula is calculated in the following steps.

$\Box(Q \wedge R) \wedge \sim ((P \wedge \circ\Box Q) \rightarrow \circ\Box R)$.

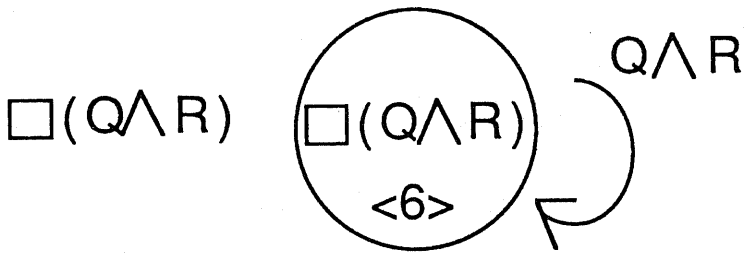


Figure 5.4: State Diagram (3)

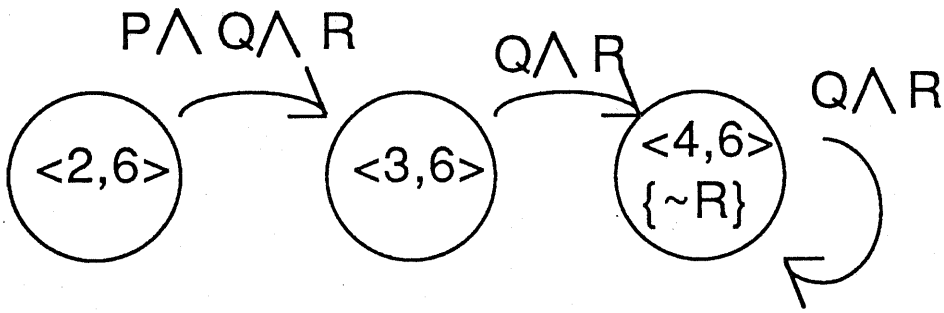


Figure 5.5: State Diagram (4)

The formula $\Box(Q \wedge R)$ is translated into a state diagram of Figure 5.4 in the same manner that Figure 5.2 is derived. Figure 5.3 represents the other formula $\sim ((P \wedge \circ\Box Q) \rightarrow \circ\Box R)$.

Then, Figure 5.5 is obtained by tracing each state diagram Figure 5.4 and Figure 5.3.

There exists no loop (even the sequence $< 4, 6 >, < 4, 6 >, \cdots$ does not satisfy the eventuality $\{\sim R\}$). Therefore, this formula is concluded as unsatisfiable.

5.2.2 Verification Method using Cover Expression

This system verifies that the hardware designs really satisfy the specifications. Let D be the temporal logic expression for hardware design and S be that for the specification. We must investigate whether the following formula is valid.

$$D \rightarrow S$$

In order to prove it, we show that the negation of the formula, that is

$$D \wedge \sim S$$

is unsatisfiable. Therefore, we have only to do the following.

1. To make state diagrams for $\sim S$ and D.

2. To check whether there is any infinite sequence of state transitions for both state diagrams $\sim S$ and D.

If there exists an infinite sequence, the design does not satisfy the specification. Otherwise, the design is correct with respect to the specification S.

The verification is processed with handling boolean expressions in the cover (sum-of-product form). Cover is introduced from now on.

Cube and Cover

Let p be a product term associated with a sum of products expression of a logic function with n input variables $(x_1, ...x_n)$ and m output variables $(f_1, ..f_m)$. Then p is specified by a row vector $c = [c_1, .., c_n, c_{n+1}, .., c_{n+m}]$, where

$$c_i = \begin{cases} 10 & \text{if } x_i \text{ appears complemented in } p, \\ 01 & \text{if } x_i \text{ appears not complemented in } p, \\ 11 & \text{if } x_i \text{ does not appear in } p, \\ 0 & \text{if } p \text{ is not present in the representation of } f_{i-n}, \\ 1 & \text{if } p \text{ is present in the representation of } f_{i-n}, \end{cases}$$

For example, suppose a boolean function with 4 input variables and 2 output variables. The function $f_1 = x_1 x_2 \overline{x_4}$ is represented by the cube $c = [01\ 01\ 11\ 10\ 1\ 0]$.

The input part of c is the subvector of c containing the first n entries of c. The output part of c is the subvector of c containing last m entries of c. A variable corresponding to a "11" in the input part is referred to as an input don't care. "00" never appears in the input part.

A set of cubes is said to be a cover C associated with a sum of products expression. For

$$f_1 = x_1 x_2 + \overline{x_2} x_3 + x_1 x_3;$$

$$f_2 = \overline{x_2} x_3 + \overline{x_3 x_4};$$

$$C = \begin{bmatrix} 01 & 01 & 11 & 11 & 1 & 0 \\ 11 & 10 & 01 & 11 & 1 & 1 \\ 01 & 11 & 01 & 11 & 1 & 0 \\ 11 & 11 & 10 & 10 & 0 & 1 \end{bmatrix} \text{ is obtained.}$$

Intersection Suppose that the intersection (logical and) of two cubes c and d, written as $c \cdot d$, is a cube e. Then, the entries e_i of the cube e are obtained from bit-and operation between cube c and cube d.

Example.
$$\begin{array}{ll} f_1 = x_1x_2 & [01\ 01\ 11\ 1] \\ f_1 = x_2x_3 & [11\ 01\ 01\ 1] \\ \Downarrow & \Downarrow \\ f_1 = x_1x_2x_3 & [01\ 01\ 01\ 1] \end{array}$$

On-cover and Off-cover For a certain output variable f_i, the set of all cubes which makes f_i be 1 is called on-cover for the output variable f_i; similarly the set of all cubes which makes f_i be 0 is called off-cover for the output variable f_i.

Verification Algorithm using Cover Expressions

Then, the verification algorithm using cover expressions is shown. Synchronous circuit is generally divided into combinational part and flip-flop part like Figure 5.6. The verification is processed with simulating combinational part in the form of cover expressions.

The verification algorithm using cover expressions is shown in Figure 5.7.

Figure 5.7 is explained by illustrating the actual verification process. The example used here is the control part of receiver by handshaking [FTM83]. The structure of the design is shown in Figure 5.8 and specification to be verified is

$\Box(\text{Call} \rightarrow \Diamond\ \text{Hear})$

with the condition that flip-flops are reset at initial state.

(preparation)

All cubes are represented in the form of [Call, CY, Hear-i, Call-o, Hear-o, Hear]. (input variables are Call, CY, and Hear-i; output variables are Call-o, Hear-o, and Hear) The on-cover and off-cover of the combinational part are as follows. Details of computing the on-cover and the off-cover are described in [Nak87].

$$\text{Con} = (\text{on-cover}) \begin{bmatrix} 01 & 11 & 11 & 1 & 0 & 0 \\ 01 & 10 & 11 & 0 & 1 & 0 \\ 01 & 01 & 01 & 0 & 1 & 0 \\ 11 & 11 & 01 & 0 & 0 & 1 \end{bmatrix}$$

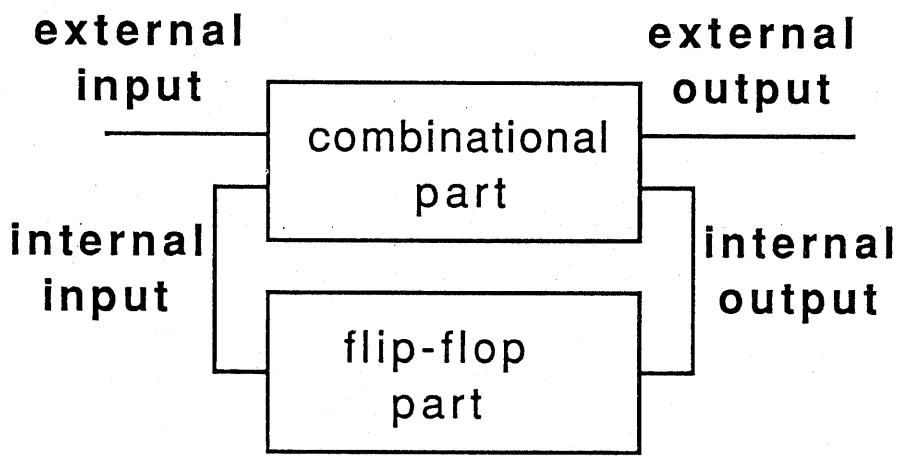


Figure 5.6: Structure of Synchronous Circuit

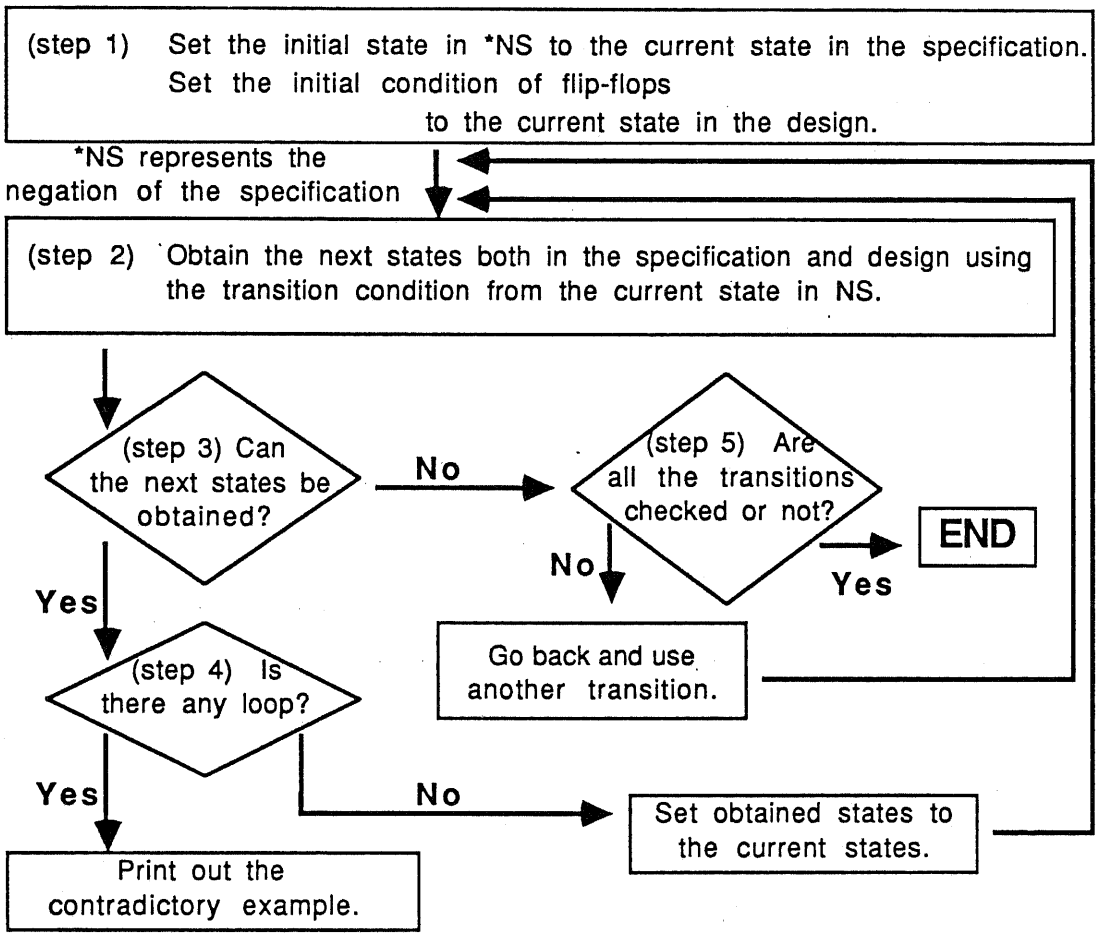


Figure 5.7: Flowchart of Verification

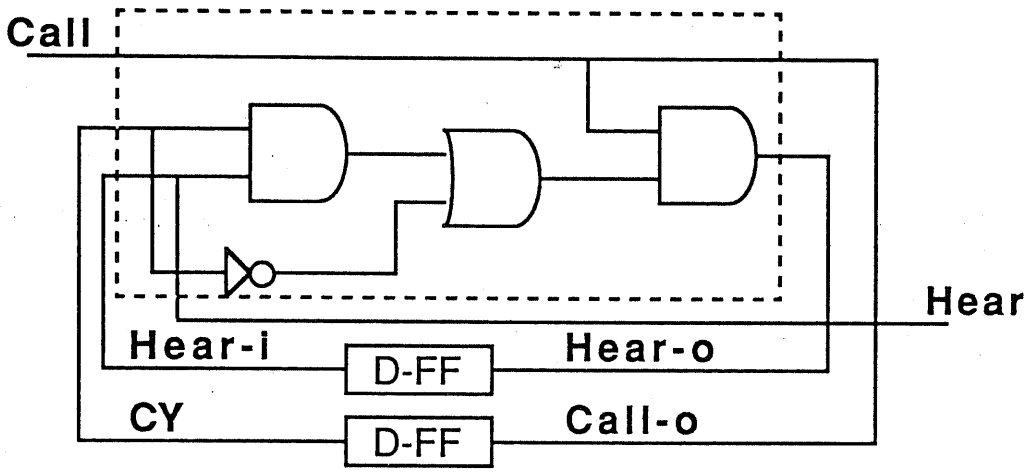


Figure 5.8: Control Part of Receiver by Handshaking

$$\text{Coff} = (\text{off-cover}) \begin{bmatrix} 10 & 11 & 11 & 1 & 0 & 0 \\ 10 & 11 & 11 & 0 & 1 & 0 \\ 11 & 01 & 01 & 0 & 1 & 0 \\ 11 & 11 & 10 & 0 & 0 & 1 \end{bmatrix}$$

For example, the second and the third rows of Con show that Hear-o on in two cases, whether Call is on and CY is off or Call, CY, and Hear-i are all off.

Connections between flip-flops and the combinational part is illustrated as oHear-i = Hear-o and oCY = Call-o.

The negation of the specification, that is $\sim \Box(\text{Call} \rightarrow \Diamond\text{Hear})$ is translated into a state transition diagram such as in Figure 5.9. This state diagram is represented as NS in Figure 5.7.

(step1) The initial state of NS is $< 1 >$ in Figure 5.9. The condition in which flip-flops are reset is described in the cube form as

Ccond = [11 10 10 1 1 1].

(step2) The transitive condition from the state $< 1 >$ into the state$< 2 >$ in NS is (Call $\wedge \sim$ Hear). The cube for the condition Call is

[01 11 11 1 1 1].

Then the cube for the condition $\sim$ Hear is

[11 11 10 1 1 1],

which is taken from the fourth row of Coff. The cover for (Call $\wedge \sim$ Hear) is obtained as

Ct = [01 11 10 1 1 1].

The next state in NS is $< 2 >$ and that in the design are calculated from

Cnext-on = Ccond·Con·Ct

and

Cnext-off = Ccond·Coff·Ct.

(step3) The next state in design is obtained from Cnext-on and Cnext-off in the following way. If there is a certain output variable that has the value 1 only in the

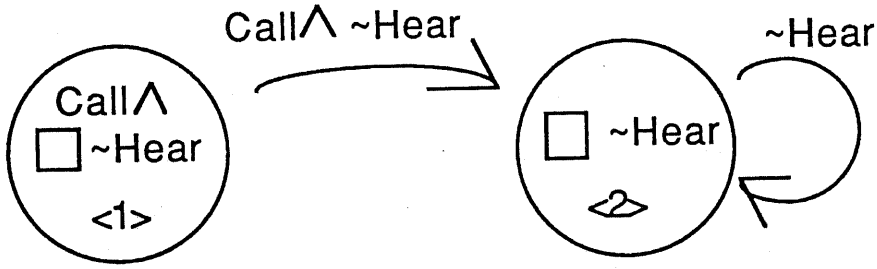


Figure 5.9: State Diagram for NS

cover Cnext-on, that variable should be 1 at the next state. Similarly, if there is a certain output variable that has the value 1 only in the cover Cnext-off, that variable should be 0 at the next state.

Judging from the covers

Cnext-on = [01 10 10 1 1 0]

Cnext-off = [01 10 10 0 0 1],

at the next state Call-o and Hear-o are 0 and Hear is 1. Considering the connection between flip-flops and combinational part, the next state in design are as follows.

Cnext = [11 01 01 1 1 1].

If there exist some variables that have the value 1 both in Cnext-on and Cnext-off, the values for those variables at the next state can not be decided.

If there exists a variable that has the value 1 neither in Cnext-on nor Cnext-off, the next state in design can not be obtained. In this case, verification flow goes to the step (step5) as shown in Figure 5.7.

(step4) Check whether there exist any loop both in NS and in the design. If there exists a loop which satisfies eventuality, that is a contradictory example.

Otherwise, set $< 2 >$ to the current state in NS and Cnext to the current state in design (in other words, set Cnext to *new* Ccond), and go to the (step2).

(step2) The transitive condition in NS is

Ct = [11 11 10 1 1 1].

In this case, since Ct·Ccond = **nil**, both Cnext-on and Cnext-off are **nil**. and the next state in design can not be obtained. Then go to the (step5).

(step5) There remains no state transition in NS, the process of verification has been finished. It is concluded that the design of Figure 5.8 satisfies the specification $\Box$(Call $\rightarrow \Diamond$ Hear).

5.2.3 Techniques for Increasing the Efficiency

Filtering Structural Description

One technique for suppressing the verification cost is *filtering structural description.* This technique extracts a part of the structure which is really required for the verification.

Specifications to be verified usually have a few external output terminals. At first, these external output terminals are marked. Second, the networks of the structure are traced backward from the output towards the input. Finally, the required part for the verification is obtained. This is the logic networks truly influencing the specification. This procedure is called *filtering design description.*

A system is generally divided into the control part and the function part. Since the verifier treats only control parts, what is really needed for verifying the given specification is much smaller than the whole descriptions. Suppose the case that the number of flip-flops in the design decreases by one with this technique. Then, the number of internal states of the circuit is reduced by half. Therefore, the filtering of a design description decreases the verification cost drastically and keeps it manageably small.

Memorization of States

In the verification flowchart Figure 5.7, when we get a state that has ever appeared, we need not check either this state or any states following this state again, provided that all the states which have ever appeared are remembered, This technique is called *memorization of states.* Using this technique, the size of necessary memory increases though the required CPU-time is suppressed.

5.3 Experimental Results

Two examples have been verified. One is a receiver by handshaking[FTM85] and the other is a DMA controller for a mini computer[Use]. The experimental results are shown in this section. The verification system is implemented on SUN3/260 (4MIPS).

5.3.1 Receiver Circuit

The receiver by handshaking is used as an example. The structure of the circuits is shown in Figure 5.10.

The following specification is verified with varying the bit-width of the data path. The data path part is in the upper part of Figure 5.10.

$$\Box(\mathsf{Reset} \rightarrow \Box(\mathsf{Call} \rightarrow \Diamond\mathsf{Hear})).$$

Since this specification has only one output variable Hear, we have only to verify filtered part about Hear. The filtered part is surrounded by broken lines in Figure 5.10. This is the same as Figure 5.8. The negation of the given specification is decomposed into the state diagram consists of 2 states.

Size of on and off covers Size of on and off covers are as shown in Table 5.1.

Sum of traced states Sum of traced states in the state diagram (~Specification ∧ Design) is 2 in all cases.

Memory usage The size of used memory in translating HSL description into cover and in verifying are 130KB and 30KB respectively in the worst case of data path of 16 bits without filtering. The size of required memory for other cases are less than these.

CPU time CPU time required for the verification is shown in Table 5.2.

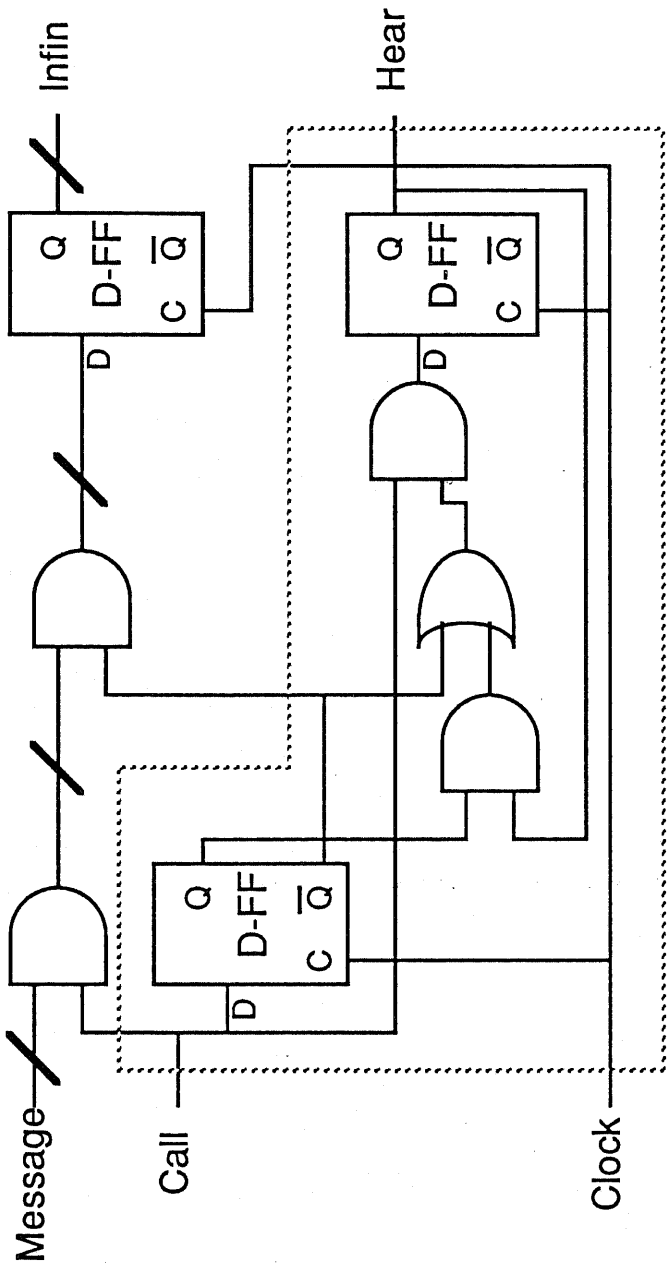


Figure 5.10: Receiver by Handshaking

		sum of input	sum of output	sum of cubes (on cover)	sum of cubes (off cover)
bit width of data path (not filtered)	1bit	5	5	6	8
	4bits	11	11	12	20
	8bits	19	19	20	36
	16bits	35	35	36	68
filtered		3	3	4	4

Table 5.1: Size of On and Off Covers for Receiver

CPU time SUN3-260 4MIPS [sec]		HSL ↓ cover	state diagram ↓ cover	verification part	
				memorizing states	
				without	with
bit width of data path not (filtered)	1bit	0.14(0.12)	0.07	0.07	0.07
	4bits	0.29(0.22)	0.12	0.13	0.13
	8bits	0.56(0.30)	0.27	0.27	0.29
	16bits	1.49(0.64)	0.84	0.84	0.94
filtered		*	0.06	0.06	0.06

* Since the design is filtered in making covers, the time depends on the bit width of data path. The time for each width is shown above in (parentheses).

Table 5.2: Required CPU Time for Verifying Receiver

5.3.2 DMA Controller

The next example is a DMA controller for a mini computer U-300 [Use]. The structure of the DMA controller is shown in Figure 5.11. The following two specifications are verified.

(1) $\Box((\text{Reset} \wedge \circ \Box \sim \text{Reset} \wedge \Box \sim \text{Acdt}) \rightarrow \circ\Box(\text{Rqdma} \rightarrow \circ\ \text{Rqdt}))$

(2) $\Box((\text{Reset} \wedge \circ \Box \sim \text{Reset} \wedge \circ \Box \sim \text{Rqdma}) \rightarrow \circ\Box(\text{Acdt} \rightarrow \circ \sim \text{Rqdt}))$

The negation of each specification is decomposed into the state diagrams consists of 4 states. Filtered part is dotted in Figure 5.11.

Size of on and off covers Size of on and off covers are as shown in Table 5.3.

Sum of traced states Whether filtering or not, sum of traced states in the state diagram ($\sim$ Specification $\wedge$ Design) are 7 for the specification(1) and 6 for the specification(2).

Memory usage The size of used memory during verification is about 50KB for each specification in either case of memorizing states or not. The size of used memory in translating HSL description into cover expressions is ,in the worst case, about 110KB without filtering.

CPU time CPU time required for verification is shown in Table 5.4.

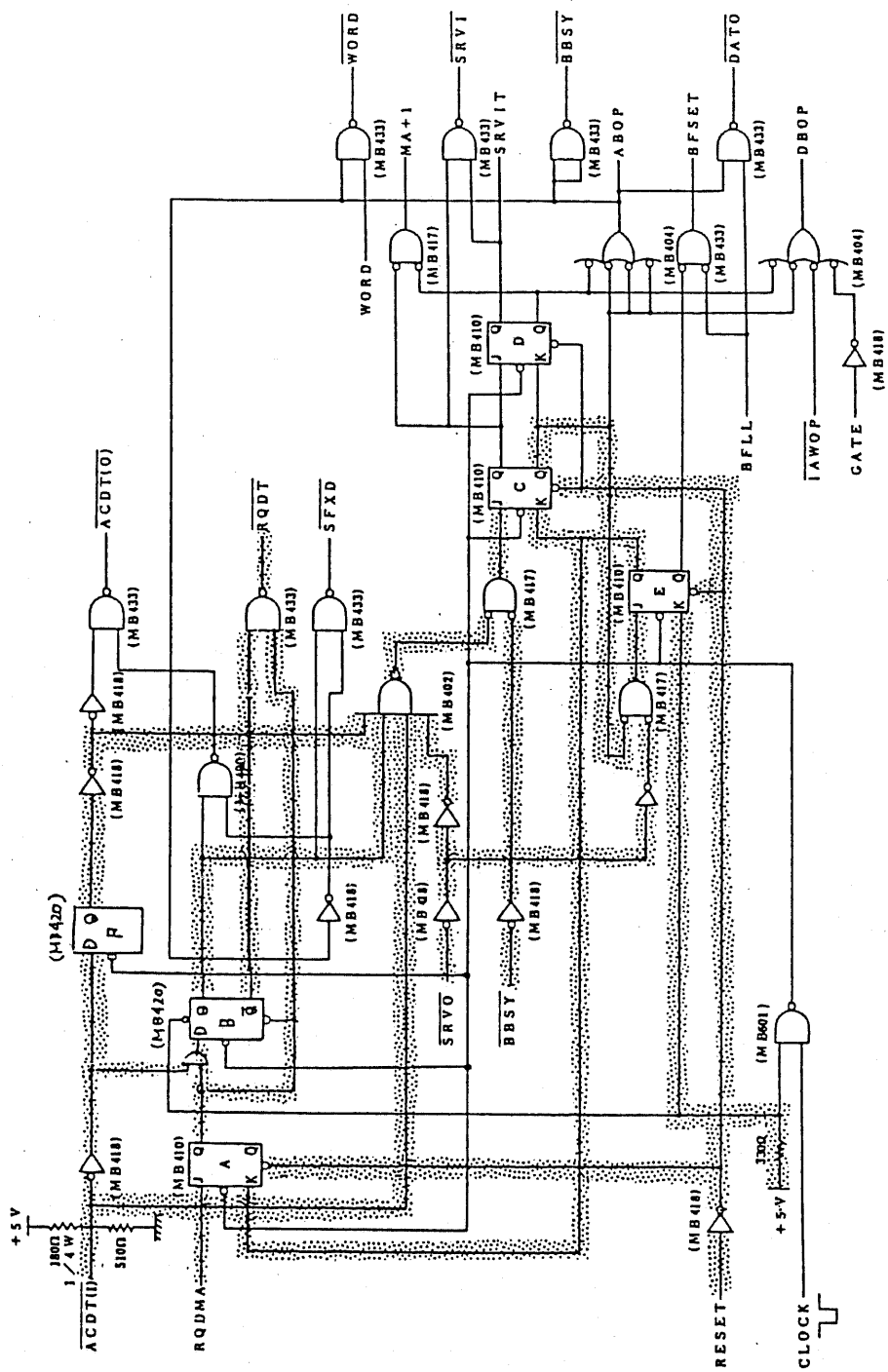


Figure 5.11: DMA Controller

	sum of input	sum of output	sum of cubes (on cover)	sum of cubes (off cover)
not filtering	16	18	46	59
filtering	11	6	11	32

Table 5.3: Size of On and Off Covers for DMA Controller

CPU time SUN3-260 4MIPS [sec]		HSL ↓ cover	state diagram ↓ cover	verification part	
				memorizing states	
				without	with
not filtered	(1)	1.36	0.44	0.66	0.58
	(2)	1.36	0.44	0.65	0.54
filtered	(1)	1.13	0.21	0.27	0.23
	(2)	1.13	0.21	0.26	0.22

Table 5.4: Required CPU Time for Verifying DMA Controller

Chapter 6

Verification of Data Path

In this chapter, a data path verification system at the register transfer level is presented. This is a central part of the proposed logic design assistance system. The input to the verifier are the behavioral description and the structural description at the register transfer level. The following assistances are realized by this verifier.

- Link informations between the behavior and the structure are derived automatically.
- Consistency between the behavior and the structure is verified automatically.
- The logic of the control part is derived automatically.

6.1 Structure

The structure of the verifier is shown in Figure 6.1. The input to this system are the behavioral description in RTL-Tokio, the structural description in Prolog, and the operation rules in Prolog. The operation rules are regarded as a part of the structural description. The other widely-used structural description languages can be translated into this style easily.

The assumptions in the verification are as follows.

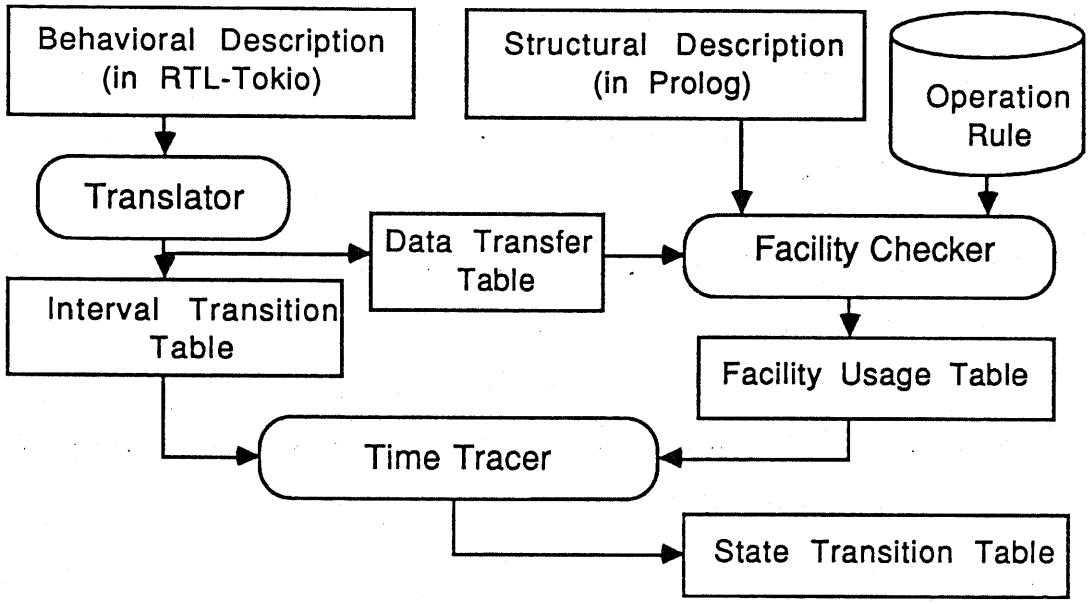


Figure 6.1: Structure of Data Path Verifier

1. All the data transfers occupy the necessary paths and operators during the whole one time period specified in the behavior.

2. Operations in the behavior are linked to the operators in the structure by using the operation rules.

3. Names of the registers and memories in the behavior are the same as those in the structure.

Instead of the second assumption, it may be another assumption that one type of operations should be linked to one type of operators. That is, the operation of '+' is realized by an operator of adder, and '>> 1' is realized by a shifter. The reason why the operation rules are introduced is as follows. By introducing the operation rules, modules can be regarded as black boxes. Suppose a data path with modules which have some functions. The designers frequently construct such a data path while they are deriving register level behavior. In order to verify such a structure, the second assumption is required. In addition to that, this system can verify a behavior without operation schedulings provided that required timing relations are declared. In other words, a behavior at a little higher level than the register transfer level can be verified owing to the second assumption.

The process of the verification is divided into the following two stages.

• To find sets of paths and operators which realize data transfers. This stage is executed by the *Translator* and *Facility Checker*.

• To check whether any paths or operators are used for more than two data transfers simultaneously. This stage is executed by the *Time Tracer*. The state transition table of control part is also extracted in this stage.

In the next section, the format of structural description is presented at first, and then each part of this data path verifier is explained. Experimental results are

shown in section 6.3.

6.2 Verification Method

6.2.1 Structural Description

There are three types of declaration as follows. Since a hierarchical structure can be expressed in this format, the other widely-used structural description languages can be translated into this style easily.

- type(type-name, facility-name, module-name).

"`type(aa,bb,cc).`" represents that a facility of name `bb` in the module `cc` belongs to the group of the type `aa`. The module name of the top level is reserved as *top*.

- path(path-name, input-port, output-port).

This expression represents the network between the facilities. To declare the bit-width explicitly, the following syntax is prepared.

```
path([p412,[0,15]],[bus1,[out,[0,15]]],[add1,[in,[0,15]]]).
```

- func([function, input-port, output-port], signal-line).

This declaration represents the function of each type.

6.2.2 Translator

The RTL-Tokio description is translated into the *interval transition table* and the *data transfer table* by the translator.

First, all the intervals in the RTL-Tokio description are identified. The identifier is in the form of

intId = (predName/arity, clauseNum,intNum). For example, `(main/0,1,2)` represents the second interval of the first clause whose name is main and whose arity is none. Second, the length of each interval is decided. Then, all the transitive relations are extracted with its transitive conditions. The transitive relations are caused by

chop operators and predicate calls. All these informations construct the *interval transition table.*

After the interval transition table has been obtained, all the data transfers in RTL-Tokio are extracted with interval-identifier (intId) and gathered into the *data transfer table.*

6.2.3 Facility Checker

The facility checker transforms the data transfer table into the *facility usage table* using the *operation rules.* From the given data path, this part selects a set of paths and operators which realize each data transfer listed in the data transfer table. The *facility usage table* consists of these selected ones. To put it in other words, this table represents the link informations between the behavior and the structure. This transformation is processed in accordance with the following steps.

- Find a operator or a set of operators which realizes the operation in each data transfer.
- Search for data paths from the source register to the input of the operator and those from the output of the operator to the destination register.

In the first step, care must be taken that one operation is realized in several ways. For example, the operation of 'multiplied by 2' is realized not only by using a multiplier, but also by using an adder or one-bit shifter. In order to find all the candidates, therefore, both the functions of the facilities and the rules of the equivalent operations should be given in advance. The latter is called the operation rules. The former is declared in the structural descriptions as follows.

```
func([add,[[adder,in1],[adder,in2]],[[adder,out]]],[adder,cnt]).
```

The operation rules are also given in the following Prolog form.

```
opSame([[multi,X,2],[shift_left,X],[add,X,X]]).
```

The operation rules are constructed in advance, but the user is allowed to add any necessary rules. The fact that the functions of the facilities and the operation rules are given in the same form provides us with a smooth design assistance.

If the facility usage table cannot be obtained or if a certain facility is used twice in the same interval, the RTL-Tokio behavior cannot run on the specified structure. In the case that a certain facility is used in some intervals, the *Time Tracer* checks whether these intervals occur simultaneously or not. Two method of time trace have been implemented. One is called as forward time trace and the other is called as backward time trace.

6.2.4 Forward Time Trace

The process of the forward time trace is divided into the following two stages. In the first stage, all the intervals occurring simultaneously are listed by tracing the intervals transition table forward. A state transition table in Moore type is also derived in this stage. In the second stage, all the listed intervals are checked whether they use the same facility or not. Conflicts of data transfers are detected in this stage.

Derivation of State Transition Table: Stage1

A state in Moore state diagrams is defined by $\{intervalName, clockNumber\}$ in RTL-Tokio description. A state transition table is obtained by traversing all the transitions listed in the interval transition table forward clock by clock. The algorithm is explained at first, and then the way of execution is illustrated by using a simple example.

step1 The initial interval I_{init} and initial clock I_{clock} (usually 0) is selected. $S_0 = \{[I_{init}, I_{clock}]\}$.

step2 (Predicate Call)

If S_i has predicate calls, all the called intervals are added to S_i. The obtained S_i' are recorded with the transitive conditions. The names of the ancestors are also recorded unless the predicate call occurs in the last interval of the predicate.

step3 To proceed one clock, and S_{i+1} next to S_i' are obtained. Contents of S_{i+1} are the list of states if the behavior contains concurrency. There are two methods in this step as follows.

1. Increment the clock number I_{clock} if the obtained clock number is not larger than the length of that interval.
2. Transfer to the next interval through *chop* operator &&.

Step3 is followed by step2.

Halting Condition Let S_n' be the newly obtained one in step2. If $0 \leq \exists\, i \leq n$, $S_n' \subseteq S_i'$ holds or S_{n+1} next to S_n' cannot be obtained, the execution halts. The trace always halts.

Proof. (abstract)

The number of intervals in the behavior is finite. Therefore, the number of states is finite if ancestors of predicate calls are not recorded infinitely in step2. Here, ancestors are recorded unless the predicate calls exist in the last interval. Since recursive calls exist only in the last interval by the constraint of RTL-Tokio, infinite ancestors are not recorded in step2. Thus, the number of states appearing during the trace is finite. Consequently, the time trace always halts.

Then, a practical execution of this stage is illustrated. Figure 6.2 shows an example adopted here.

- Initial set S_0 = "[[(start/0,1,1,0)]]". Here, "(start/0,1,1,0)" represents "(predicateName/arity, clauseNumber, intervalNumber, clock)". Thus, this denotes "0 clock in the first interval of the first predicate whose name/arity is start/0".

- S_0' = "[[(init/0,1,1,0),(start/0,1,1,0)]]" is obtained in step2.

- In step3, S_1 = "[[(execution/0,1,1,0)]]" is obtained as the first candidate and recorded with the transitive condition "[=<,*a,10]". Intervals are traversed in depth first. The second candidate "[[(execution/0,2,1,0)]]" is traced later.

- S_1' = "[[(sub1/0,1,1,0),(execution/0,1,1,0)]]". S_1' should be distinguished from "[[(sub1/0,1,1,0),(execution/0,2,1,0)]]" because the successors are not the same. This is the reason why ancestors of predicate call should be recorded in step2.

- S_2 = "[[(execution/0,1,2,0)]]".

- S_2' = "[[(sub2/0,1,1,0),(execution/0,1,2,0)],[(sub3/0,1,1,0),(execution/0,1,2,0)]]". In the case concurrency exists like this, contents of S_i are more than one.

- S_3 = "[[(execution/0,1,3,0)]]".

- S_3' = "[[(execution/0,1,1,0)]]" is obtained at first. The ancestor of predicate call "[[(execution/0,1,3,0)]]" needs not be recorded, because the successors of "[[(execution/0,1,1,0),(start/0,1,2,0)]]" are the same as those of "[[(execution/0,1,1,0),(execution/0,1,3,0)]]". This is the reason why the ancestors is not recorded when the predicate calls appear in the last interval.
  S_3' = "[[(sub1/0,1,1,0),(execution/0,1,1,0)]]" is obtained finally.

```
start :- init && execution.

init :- *a <= 1, *b <= 2, *c <= 4, *d <= 0.

execution :- *a =< 10,!, sub1 && sub2,sub3 && execution.

execution :- *a =< 40,!, sub1 && *a <= *d + *c && execution.

execution :- !, empty.

sub1 :- !, *d <= *a.

sub2 :- !, *a <= *d * *b.

sub3 :- !, *c <= *c + 1.
```

Figure 6.2: Traced Example

Here, S_3' is the same as S_1'. Therefore, the trace stops and another transition is searched for. Another state "[[(sub1/0,1,1,0),(execution/0,2,1,0)]]" is set to S_3'.

(The rest of the trace is omitted.)

Reduction of States:

In the example of Figure 6.2, S_1' = "[[(sub1/0,1,1,0),(execution/0,1,1,0)]]" and S_3' = "[[(sub1/0,1,1,0),(execution/0,2,1,0)]]" can be regarded as one state because the data transfers within them are the same. In order to put them together into one state, the transitive conditions should be changed. This system only produces a message about the possibility of the reduction.

Figure 6.3 shows a state diagram of control part which is derived from Figure 6.2.

Conflict Detection: Stage2

This stage is divided into the following two steps.

- Listing all the concurrent intervals.

  All the elements in each S_i' occur simultaneously unless they have exclusive transitive conditions. For the given two transitive conditions, they are judged as exclusive only if they have the exclusive conditions at the same time spot. In order to detect exclusiveness with time passing, linear programming method should be introduced. This remains to be solved.

- Detecting facility conflict

  Using *facility usage table*, all the concurrent intervals are checked whether they use the same facilities. If they use the same facilities, it results in data path conflict. In this check, care must be taken that there exist some alternative sets of the facility usage avoiding facility conflict.

```
Obtained State Transition :
   Ancestor = 0
   Obtained Successor = 1
       Obtained State = [[(init/0,1,1,0),(start/0,1,1,0)]]
Transitive Condition = [[],[]]

Obtained State Transition :
   Ancestor = 1
   Obtained Successor = 2
       Obtained State = [[(sub/0,1,1,0),(execution/0,1,1,0)]]
Transitive Condition = [[],[[=<,[*,a],10]],[]]

Obtained State Transition :
   Ancestor = 2
   Obtained Successor = 3
       Obtained State = [[(execution/0,1,2,0)]]
Transitive Condition = [[]]

Obtained State Transition :
   Ancestor = 3
   Obtained Successor = 2
       Obtained State = [[(sub/0,1,1,0),(execution/0,1,1,0)]]
Transitive Condition = [[],[[=<,[*,a],10]],[]]

Obtained = [[(sub/0,1,1,0),(execution/0,2,1,0)]]
    [[(sub/0,1,1,0),(execution/0,1,1,0)]]   State Id =  2
    can be the same state as the above state.
Obtained State Transition :
   Ancestor = 3
   Obtained Successor = 4
       Obtained State = [[(sub/0,1,1,0),(execution/0,2,1,0)]]
Transitive Condition =
                    [[],[[=<,[*,a],40]],(neg_of,[[=<,[*,a],10]]),[]]

Obtained State Transition :
   Ancestor = 4
   Obtained Successor = 5
       Obtained State = [[(execution/0,2,2,0)]]
Transitive Condition = [[]]
```

Figure 6.3: Extracted State Diagram of Control Part

6.2.5 Backward Time Trace

The process of backward time trace is in the reverse order to that of forward time trace. In the backward time trace, a pair of intervals which use the same facility is listed at first, and then they are verified whether they occur simultaneously or not by tracing the interval transition table backward. The technique of tracing the interval transition tables is similar to that of the forward time trace except for the direction. Here, the abstract of the algorithm is shown.

step1 All the pairs of intervals which occupy the same facilities are found.

step2 For a given set of intervals found in step1, the concurrency check begins. Suppose the initial set of intervals be $A_0 = \{I_a\}$ and $B_0 = \{I_b\}$. In case that I_a follows I_c or I_d and I_b follows I_e or I_f, $A_1 = \{I_c, I_d\}$ and $B_1 = \{I_e, I_f\}$ are obtained as the predecessor. Using the interval transition table, the backward trace continues in this way and constructs the A_{i+1} and B_{i+1} from A_i and B_i with recording the transitive conditions.

- $\exists int,\ 0 \leq \exists i \leq n;\ int \subseteq A_i \cap int \subseteq B_i$ holds and the transitive conditions from int to I_a are not exclusive of those from int to I_b, I_a and I_b are proved to occur simultaneously, which results in the design error.
- Suppose A_n and B_n be the newly obtained sets. If $\exists i; A_n \subseteq A_i \cap B_n \subseteq B_i$ holds, or either A_n or B_n is empty, the execution is aborted and goto stage3.

step3 For all the pairs listed in step1, step2 is applied. The obtained results in the previous step2 are used to cut off the same trace. This method decreases the execution time but increases the required memory.

6.3 Experimental Results

This verifier has been applied to three examples. One is a circuit which calculates square root using Newton's method. Another is a general processor cited from [Kar89]. The last is an application specific processor called NIP (Network Interface Processor). The data path verifier is implemented on SUN4/260 using SICStus-Prolog [CW88].

6.3.1 Computing Square Root

The algorithm of computing square root using Newton's method has been already shown in section 4.4. Figure 6.4 is the pipelined behavior with two times repetition in RTL-Tokio. The structure of the data path is shown in Figure 6.5.

This example is verified with altering the repetition times. The repetition times indicates how many times the loop in Figure 4.8-(a) is executed. Irrelevant to the repetition times, the degree of internal concurrency is 2. The verification halts only after all the conflicts are detected. The results are shown in Table 6.1. In this example, `addA` is used in `(input/0,1,2)` and `(stage2/0,1,2)` and these sub-intervals occur simultaneously. Figure 6.6 shows a part of the output during the backward time trace.

In Table 6.1, the intervals with data transfers are counted as the 'Number of Intervals', and 'Number of Backward Trace' is the number of the pairs listed during `step1` of backward time trace as mentioned in section 6.2.5.

Operation Rule including Timing Relation

A behavior which is not scheduled completely has been verified. I illustrate the behavior, required operation rule of timing relations, and a part of the execution in Figure 6.7. The structure is the same as Figure 6.5. This behavior has been verified successfully.

```
start :- !, main.

main :- *adr = 8,!,true.

main :- !,input && stage1 && main,( stage2 && true).

input :- !,
        *input1 <= *memory(*adr), *adr <= *adr + 1
        &&
        *reg1 <=  0.222222 + 0.888889 * *input1.

stage1 :-!,
        *reg2 <= *input1 / *reg1
         &&
        *reg2 <= *reg2 + *reg1
        &&
        *reg3 <= *reg2 / 2,  *input3 <= *input1.

stage2 :- !,
        *reg4 <= *input3 / *reg3
        &&
        *reg4 <= *reg4 + *reg3
        &&
        *output <= *reg4 / 2.
```

Figure 6.4: Behavioral Description for Computing Square Root

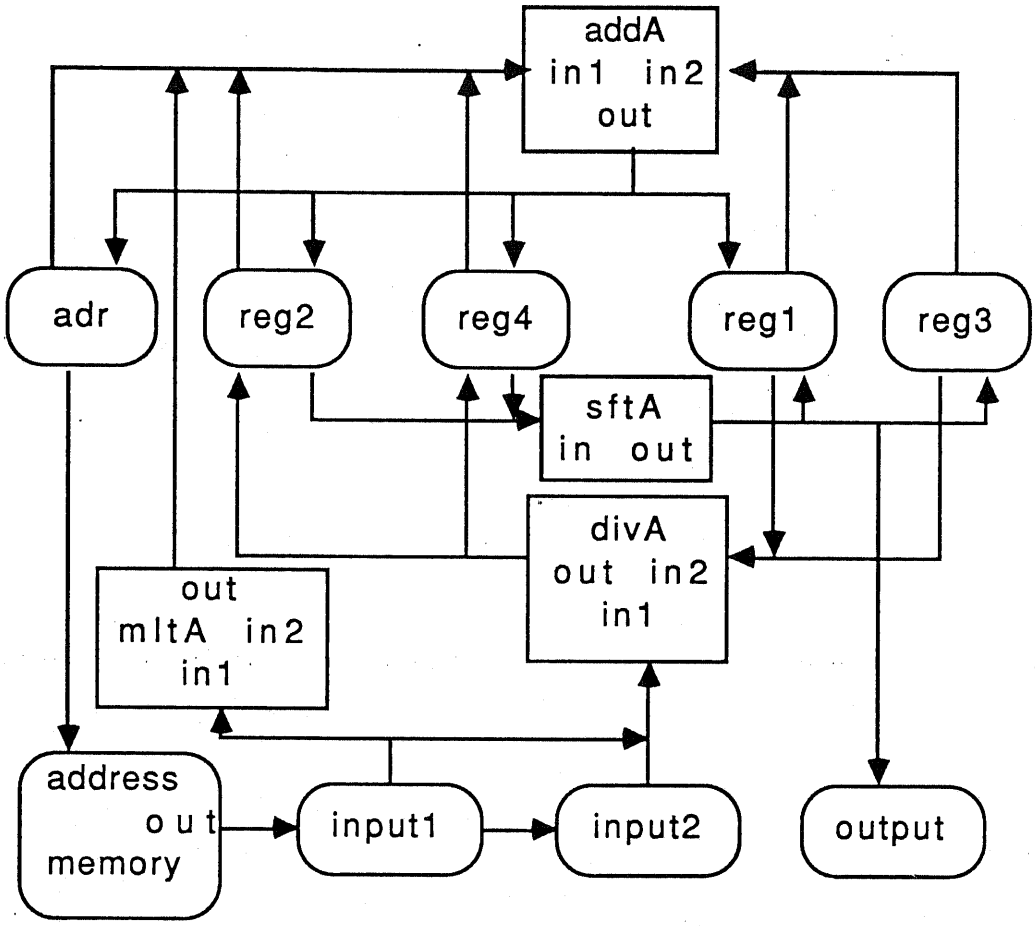


Figure 6.5: Data Path Structure for Computing Square Root

Number of Repetitions	CPU time (sec)				Number of Backward Trace	Number of Derived States
	Translator	Facility Checker	Forward Trace	Backward Trace		
2	0.29	2.23	0.53	2.10	6	12
4	0.43	3.44	1.03	9.23	28	21
8	0.62	5.99	2.15	78.8	120	39

Table 6.1: Results of Verifying the Circuit for Calculating Square Root

```
[((input/0,1,2),0)],1
 and
[((stage2/0,1,2),0)],1
 may conflict in
[[mltAaddA,[addA,[in1,[0,7]]]],[reg4addA,[addA,[in1,[0,7]]]]]

[((main/0,2,3),0)]
[((input/0,1,1),0)]
  [((input/0,1,1),1)]
  [((input/0,1,2),0)]

[((main/0,2,3),0)]
[((stage2/0,1,1),0)]
  [((stage2/0,1,1),1)]
  [((stage2/0,1,2),0)]

Happen.  These may happen at the same time.
```

Figure 6.6: A Part of Output during Backward Time Trace

```
%% Input Behavior
start :- !, main.
main :- *adr = 8,!,true.
main :- !,input && stage1 && main,( stage2   &&   true).
input :- !,
        *input1 <= *memory(*adr), *adr <= *adr + 1
        &&
        *reg1 <=  0.222222 + 0.888889 * *input1.
stage1 :-!,
        *reg2 <= (*input1 / *reg1)  + *reg1
        &&
        *reg3 <= *reg2 / 2, *input3 <= *input1.
stage2 :- !,
        *reg4 <= (*input3 / *reg3) + *reg3
        &&
        *output <= *reg4 / 2.

%% operation rule including timing relations
%% timing_rule(original scheduling,
%%                 new scheduling, unbounded variables).
timing_rule((*R <= (*A / *B) + *C) ,
                    [(*R1 <= *A / *B) && ( *R <= *R1 + *C)] ,R1).

%% a part of script file during the execution
| ?- tst('Root/root_time.tokio',extend).

*reg2<= *input1/ *reg1+ *reg1
    is replaced by
[*_1242<= *input1/ *reg1 && *reg2<= *_1242+ *reg1]
Please confirm undefined variables: _1242
|: reg2. % manual input
*reg4<= *input3/ *reg3+ *reg3
    is replaced by
[*_3902<= *input3/ *reg3 && *reg4<= *_3902+ *reg3]
Please confirm undefined variables: _3902
|: reg4. %manual input
```

Figure 6.7: Operation Rule including Timing Relation and Its Execution

6.3.2 General Processor

The next example is a general purpose processor (cited from [Kar89]). In this section, the following results are presented.

- Description of the processor in RTL-Tokio
- Computation of ackerman(1,1) function on this processor.
- Verification of the data path.

Description in RTL-Tokio

Figure 6.8 shows a part of the behavioral description in RTL-Tokio. In this description, instead of specifying the decoder, both the byte and the type of each instruction are declared explicitly for clarity.

This computer has 16 instructions. The computation consists of "ifetch" part and "execution" part. After "ifetch" fetches an instruction from the memory, "execution" is activated. For all the instructions except for add and adc, the next "iftech" stage begins after the "execution" stage has finished. In the case of add and adc, however, the next "ifetch" begins one clock earlier. Consequently, "ifetch" is executed in the last clock cycle of "execution" concurrently. The underlined part in Figure 6.8 represents this concurrency.

Computing Ackerman(1,1)

The description of Figure 6.8 has been simulated by computing ackerman(1,1). Figure 6.9 shows a part of instructions for ackerman(1,1). As shown in Figure 6.9, the computation begins by loading the instructions into the memory.

Required CPU time is shown in Table 6.2. This simulator is implemented on SUN4/260 using SICStus-Prolog. The input are the behavior of the processor (Figure 6.8) and the instructions for ackerman(1,1) (Figure 6.9). Descriptions in RTL-

```
ifetch :- !, *op1 <= *mem(*pc) , *pc <= *pc+1 && ifetch_branch.
    ifetch_branch :- *op1 = stop, !, empty.
    ifetch_branch :- (_,2,_) = *op1,!,ifetch2.
    ifetch_branch :- !, execution.
ifetch2 :- !, *op2 <= *mem(*pc) , *pc <= *pc+1 && execution.
execution:-  !, exec_first   && execution_latter.
    exec_first :- (add,_,_) = *op1, !,   *memd <= *mem(*op2).
    exec_first :- (adc,_,_) = *op1, !,  *memd <= *mem(*op2).
    exec_first :- (clr,_,_) = *op1, !,  *a <= 0.
    exec_first :- (com,_,_) = *op1, !,  *a <= alu0(*a).
    exec_first :- (push,_,_) = *op1, !, *mem(*sp) <= *a.
    exec_first :- (pull,_,_)  = *op1, !, *sp <= *sp+1.
    exec_first :- (lda,_,_)  = *op1, !,  *a <= *mem(*op2).
    exec_first :- (sta,_,_)  = *op1, !, *mem(*op2) <= *a.
    exec_first :- (lds,_,_)  = *op1, !,  *sp <= *mem(*op2).
    exec_first :- (sts,_,_)  = *op1, !, *mem(*op2) <= *sp.
    .....(omitted)........
execution_latter :- (Op,By,short) = *op1,!,ifetch.
execution_latter :- (Op,By,long) = *op1,!,
                    (exec_second && true),ifetch.
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^<===== concurrency
execution_latter :- (Op,By,sec2) = *op1,!,exec_sec2_begin.
exec_second :- (add,_,_) = *op1, !, *a <= alu1(*a,*memd).
exec_second :- (adc,_,_) = *op1, !, *a <= alu2a(*a,*memd,*c),
                                    *c <= alu2c(*a,*memd,*c).
exec_sec2_begin :-  !, exec_sec2 && ifetch.
    exec_sec2 :- (pop,_,_) = *op1, !, *a <= *mem(*sp).
    exec_sec2 :- (jsr,_,_)  = *op1, !, *pc <= *op2, *sp <= *sp-1.
    exec_sec2 :- (rts,_,_)  = *op1, !, *pc <= *mem(*sp).
    exec_sec2 :- (push,_,_) = *op1, !, *sp <= *sp-1.
'$function' alu0(A)   = Out :- alu0(A,Out).
        alu0(X,Out)   :- !, Out = -X.
'$function' alu1(A,B) = Out :- alu1(A,B,Out).
        alu1(X,Y,Out) :- !, Out = (X + Y) mod 256.
'$function' alu2a(A,B,C) = Out :- alu2a(A,B,C,Out).
        alu2a(X,Y,Z,Out) :- !, A = X + Y + Z,
            (if A > 255 then Out = A - 256 else Out = A).
'$function' alu2c(A,B,C) = Out :- alu2c(A,B,C,Out).
        alu2c(X,Y,Z,Out) :- !, A = X + Y + Z,
            (if A > 255 then Out = 1 else Out = 0).
```

Figure 6.8: Behavioral Description of Processor

```
init:- *pc := 0, *sp := 255, *a := 0, *c := 0,
       *memd := 0, *memout := 0, *op1 := (clr,1,short), *op2 := 0.

test :- init, *mem(0):=(lda,2,short),   *mem(1):=(66),
              *mem(2):=(jsr,2,sec2),    *mem(3):=(5),
              *mem(4):=(stop),          *mem(5):=(push,1,sec2),
              *mem(6):=(lda,2,short),   *mem(7):=(65),
              *mem(8):=(clc,1,short),   *mem(9):=(sta,2,short),
              *mem(10):=(67),           *mem(11):=(adc,2,long),
              *mem(12):=(63),           *mem(13):=(pull,1,sec2),
              *mem(14):=(brc,2,short),  *mem(15):=(19),
              *mem(16):=(add,2,long),   *mem(17):=(64),
              *mem(18):=(rts,1,sec2),   *mem(19):=(sta,2,short),
              *mem(20):=(68),           *mem(21):=(clc,1,short),
              *mem(22):=(adc,2,long),   *mem(23):=(63),
              *mem(24):=(brc,2,short),  *mem(25):=(35),
              *mem(26):=(lda,2,short),  *mem(27):=(64),
              *mem(28):=(push,1,sec2),  *mem(29):=(lda,2,short),
              *mem(30):=(67),           *mem(31):=(add,2,long),
              *mem(32):=(63),           *mem(33):=(jmp,2,short),
              *mem(34):=(8),            *mem(35):=(lda,2,short),
              *mem(36):=(67),           *mem(37):=(push,1,sec2),
              *mem(38):=(jsr,2,sec2),   *mem(39):=(54),
              *mem(40):=(sta,2,short),  *mem(41):=(68),
              *mem(42):=(pull,1,sec2),  *mem(43):=(sta,2,short),
              *mem(44):=(67),           *mem(45):=(lda,2,short),
              *mem(46):=(68),           *mem(47):=(push,1,sec2),
              *mem(48):=(lda,2,short),  *mem(49):=(67),
              *mem(50):=(add,2,long),   *mem(51):=(63),
              *mem(52):=(jmp,2,short),  *mem(53):=(8),
              *mem(54):=(lda,2,short),  *mem(55):=(68),
              ...........(omitted)............. , ifetch.
```

Figure 6.9: Instruction for Ackerman(1,1)

Tokio are first compiled into Prolog. Then, the obtained codes in Prolog are compiled again. After these two stages have been completed, the execution starts. The rightmost column indicates the number of required clocks in the processor.

Data Path Verification

The data path structure is shown in Figure 6.10. The bit-width of the structure is 8 except for the lines explicitly indicated.

In the verification, "ALU" is treated as a black box. Functions of "ALU" are specified in the lower part of Figure 6.8. In the structural description, those functions are declared as shown in Figure 6.11. If the function names such as "alu0" are common in both the behavior and the structure, this verifier can handle a module as a black box. If the number of the input to a signal-line is more than one, it results in a data path conflict. In this example, the functions of "alu2a" and "alu2c" can be executed simultaneously because the input signals of them are the same as shown in Figure 6.11.

The input to the verifier are the behavioral description (Figure 6.8) and the structural description (Figure 6.10). The required CPU time are shown in Table 6.3. The required time does not depend on the bit-width of the data path structure. This is because a path of 8 bit-width is handled together as a single path in the *facility checker*. The number of states which appear in the derived *state transition table* is 26. In this example, no states were reduced.

The backward time trace can test whether the given two intervals occur simultaneously or not. It is also possible in the backward time trace to specify one interval and search for other intervals which occur simultaneously with the specified one. As shown in Figure 6.8 obviously, `exec_second` is always included in the pairs of intervals which occur simultaneously. Therefore, we only have to trace backward with specifying one interval as `exec_second`. Required CPU time for this backward

	Tokio → Prolog	Prolog Compilation	Required CPU time for Execution	Required Clock Cycle
Behavior of Processor	1.73 sec	4.27 sec	2.49 sec	168 clock
Instructions	0.66 sec	1.18 sec		

Table 6.2: Required CPU Time for Simulating ackerman(1,1)

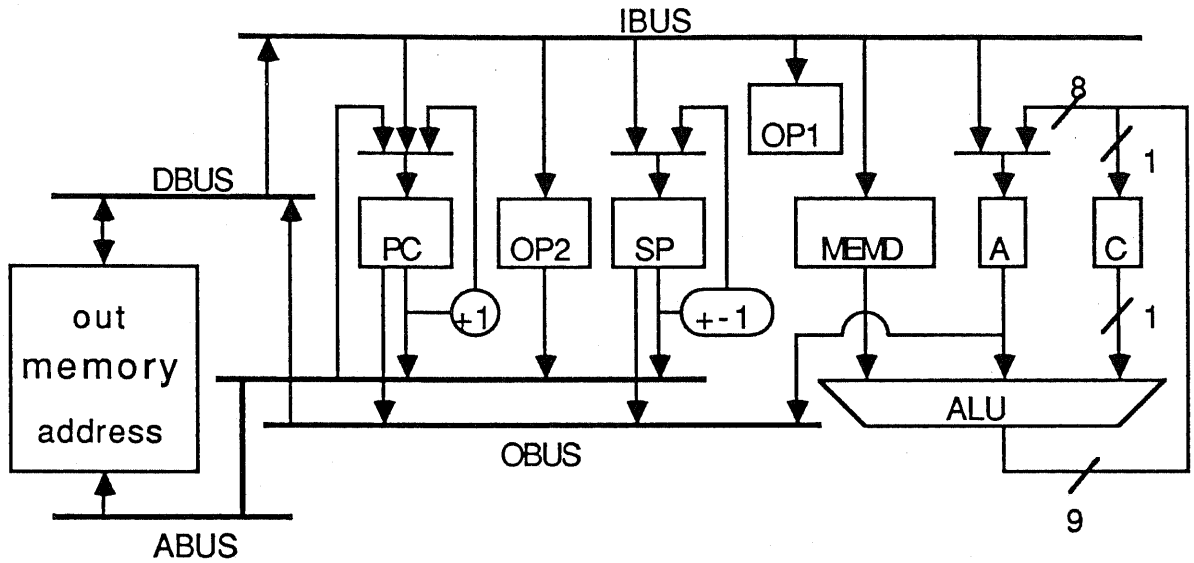


Figure 6.10: Data Path Structure of Processor

```
func([alu0,[[alu,[in2,[0,7]]]],[[alu,[outA,[0,7]]]]], [alu,alu0]).
func([alu1,[[alu,[in2,[0,7]]],[alu,[in1,[0,7]]]],
          [[alu,[outA,[0,7]]]]],[alu,alu1]).
func([alu2a,[[alu,[in2,[0,7]]],[alu,[in1,[0,7]]],[alu,[in3,[0,0]]]],
           [[alu,[outA,[0,7]]]]],[alu,alu2]).
func([alu2c,[[alu,[in2,[0,7]]],[alu,[in1,[0,7]]],[alu,[in3,[0,0]]]],
           [[alu,[outC,[0,0]]]]],[alu,alu2]).
```

Figure 6.11: Declaration of Function in ALU

CPU time (sec)				Number of Derived States
Translator	Facility Checker	Forward Trace	Backward Trace	
2.22	6.51	17.3	775	26

Table 6.3: Result of Verifying Processor

time trace was 1.38 sec.

6.3.3 Network Interface Processor

The last example is a network interface processor (NIP) in PIE64 [KT88]. PIE64 is a parallel inference machine which executes knowledge information processing in parallel, and is under development in our laboratory. This machine consists of 64 inference units and two high speed interconnection networks. Each inference unit has one inference processor, four network interface processors (NIP), one SPARC processor, and local memory modules. NIP processes the interface between each inference unit and the interconnection network.

The structure of the NIP is divided into four parts: a command-bus interface part, a memory-bus interface part, a process synchronization part, and a data transfer part as shown in Figure 6.12. The four parts of NIP acts in cooperation, but each part has its own sequencer and is controlled independently. The data path verifier has been applied to the process synchronization part and the data transfer part. NIP was designed by Takeshi Shimizu in our laboratory. The behaviors of the sequencers were originally written in FLDL-F which was developed at FIJUTSU LIMITED. The behaviors were translated into RTL-Tokio by Yuji Kukimoto in our laboratory. The structure of the data path was also translated into Prolog style by Yuji Kukimoto.

Process Synchronization Part

This part supports the following functions. Details are described in [SKT89].

- Addition of the contexts to the suspension record lists. This function is realized in **suspend** command.
- Collection of the suspension record lists. This function is a part of `activates` command.

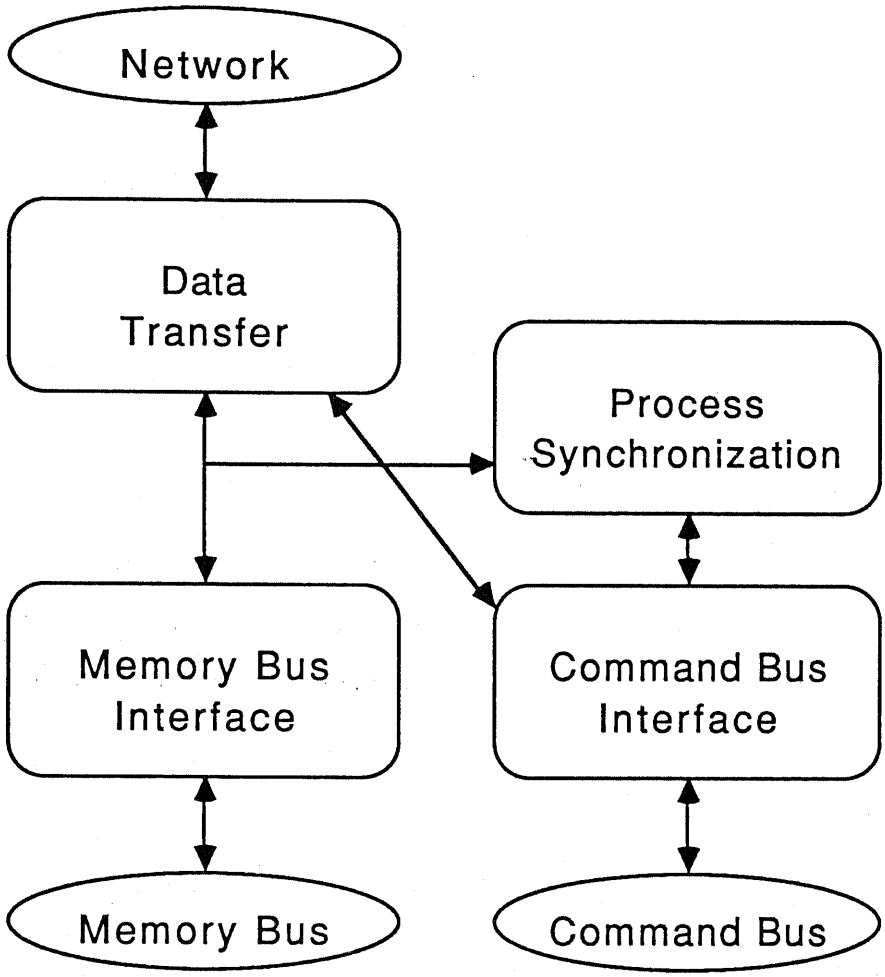


Figure 6.12: Structure of Network Interface Processor

```
%% state transition description
start :- !, idlestate.
idlestate :-  *req1=1,*req0=1,!,
        dataouts, btmp
        &&
        activatess.
idlestate :- *req1=0,*req0=1,!,
        dataouts, btmp
        &&
        localbind.
idlestate :- *req1=1,*req0=0,!,
        *btmp <= 0, dataouts, btmp
        &&
        localsuspends.
idlestate :- *req1=0,*req0=0,*xreqnd=0,*rsusp=0,!,
        dataouts, btmp
        &&
        slavebind.
idlestate :- *req1=0,*req0=0,*xreqnd=0,*rsusp=1,!,
        dataouts, btmp
        &&
        slavesuspend.
idlestate :- !,
        dataouts, btmp
        &&
        idlestate.

%% data output
dataoutf :- *xdack=0,!,
        *dataout <= *datain.
dataoutf :- !,
        *dataout <= dfree(*free).
dataoutv :- *xdack=0,!,
        *dataout <= *datain.
dataoutv :- !,
        *dataout <= dval(*val).
dataouts :- *xdack=0,!,
        *dataout <= *datain.
dataouts :- !,
        *dataout <= dsusrec(*susrec).

%%% address output
var :- !, *addrout <= avar(*var).
free :- !, *addrout <= afree(*free).
btmp :- !, *addrout <= abtmp(*btmp).
```

Figure 6.13: Behavioral Description of Process Synchronization Part in NIP

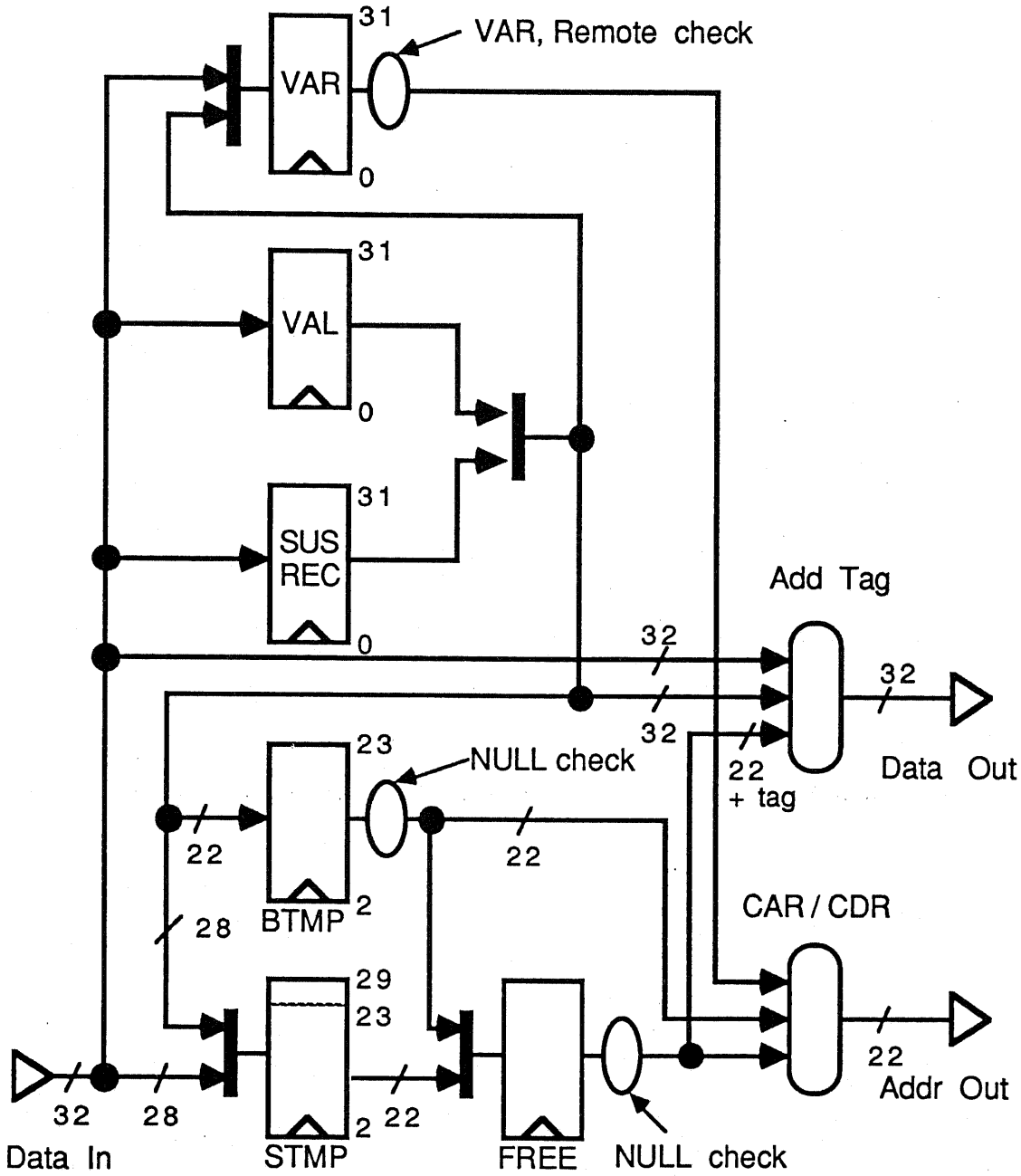


Figure 6.14: Data Path Structure of Process Synchronization Part in NIP

CPU time (sec)				Number of Derived States
Translator	Facility Checker	Forward Trace	Backward Trace	
5.97	7.85	1488	(not measured)	236

Table 6.4: Result of Verifying Process Synchronization Part in NIP

Its behavior and structure are shown in Figure 6.13 and Figure 6.14 respectively. The arities of the commands are held in the registers of VAR and VAL. SUSREC, BTMP, and STMP are used as temporary registers. The register FREE holds the free list. The results of the verification is shown in Table 6.4. In this example, 178 states are detected to be reduced.

Data Transfer Part

This part supports the data transfers through the network and the garbage collections. Data are transferred through the network as shown in Figure 6.15.

The data path structure of the data transfer part is shown in Figure 6.16. Since the required performance for this part is quite severe, the behaviors of both the read-side and the write-side NIPs are pipelined. In the read-side NIP, a data is transferred $BUF0 \rightarrow BUF1 \rightarrow NOB$ in the pipeline of 3 stages. In the write-side NIP, a data is transferred $NIB \rightarrow BUF2 \rightarrow NOB$ in the pipeline of 3 stages. Other registers in Figure 6.16 are used as buffers when data transfers cannot be pipelined.

The upper part of Figure 6.16 which includes $BUF0$ and $BUF1$ and the lower part which includes $BUF2$ and $BUF3$ are controlled respectively. The behaviors of the two parts are shown in Figure 6.17 and in Figure 6.18 respectively. Since the upper part is used mainly in a read-side NIP and the other part is used mainly in a write-NIP, we call them read-part and write-part of NIP. These behaviors are verified whether they can run on the structure illustrated in Figure 6.16. The results are shown in Table 6.5 and Table 6.6 respectively. In this example, no states and 20 states are detected to be reduced respectively.

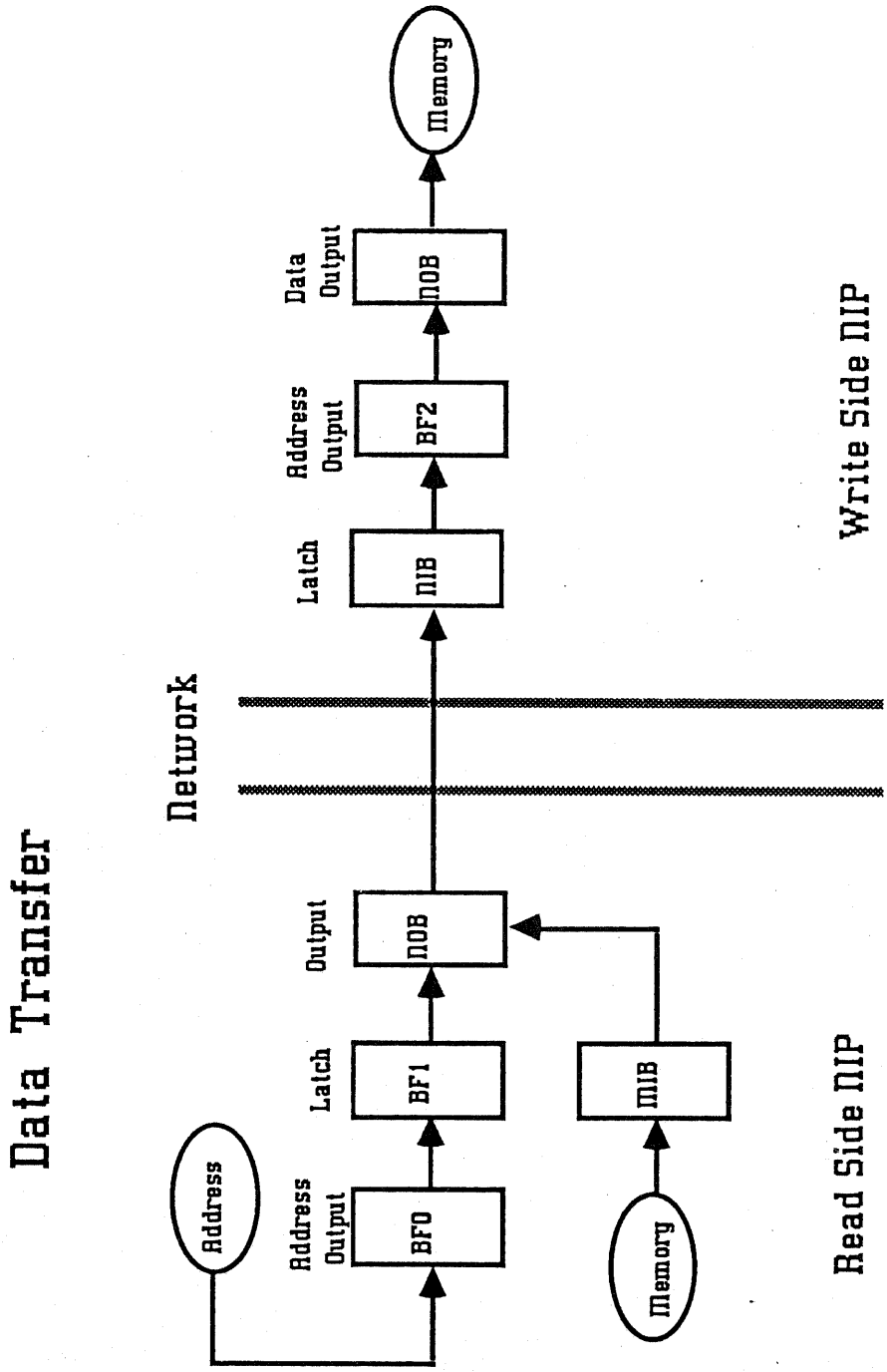


Figure 6.15: Data Transfer through Network

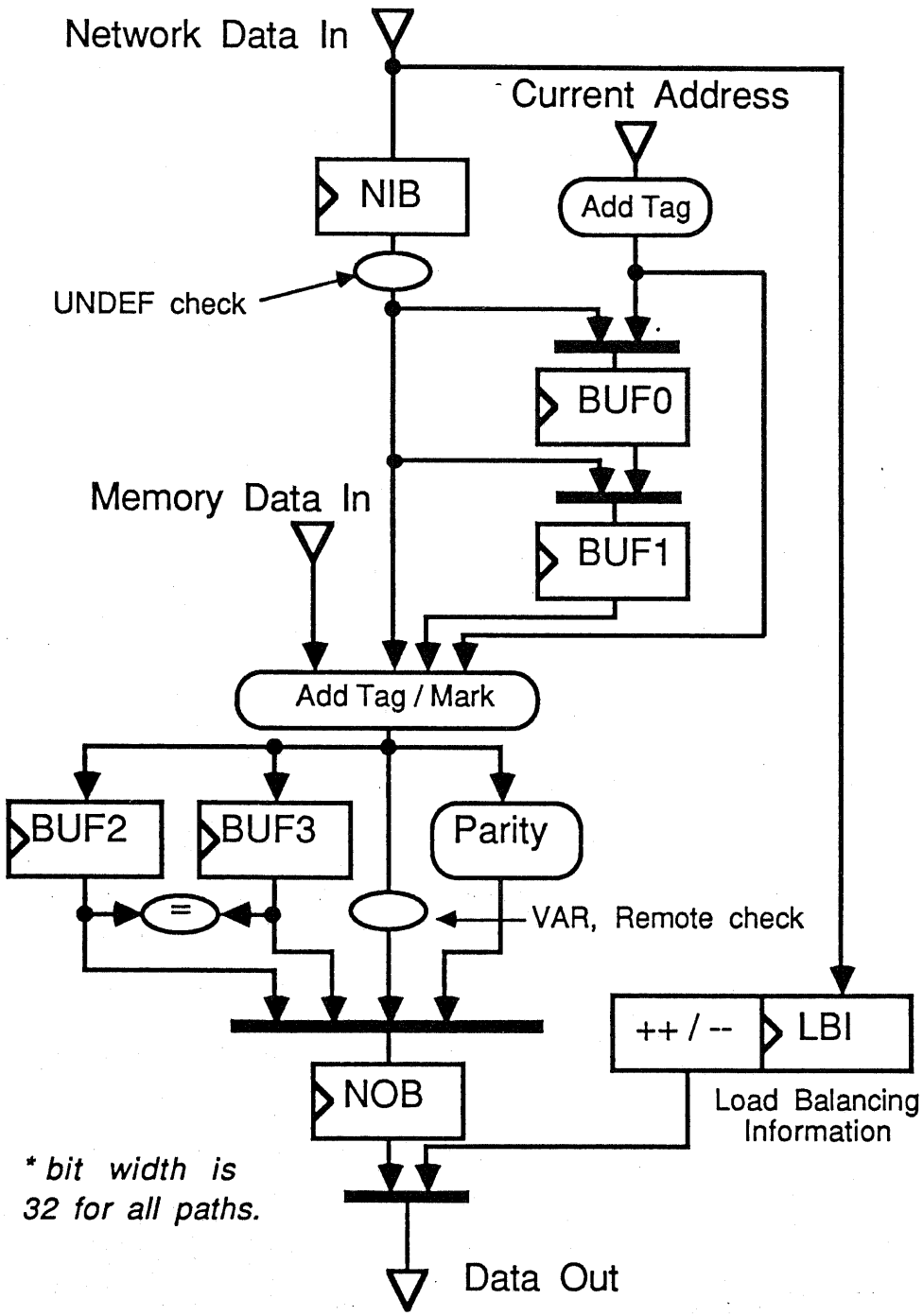


Figure 6.16: Data Path Structure of Data Transfer Part in NIP

```
start:-  !,initial.
initial:- *rmode=1,!,
        *buf0 <= *addr, *buf1 <= *buf0
        && initial.
initial:- *full=1,*xnack=0,!,
        *buf1 <= *nib
        && st001.
initial:- !, true,length(1) && initial.

st001:- *getd=1,*xnack=0,!,
        *buf1 <= *nib
        && st001.
st001:- *getd=1,!,
        true,length(1) && initial.
st001:- *xnack=0,!,
        *buf0 <= *nib
        && st011.
st001:- !, true,length(1) && st001.

st011:- *getd=1,*xnack=0,!,
        *buf0 <= *nib, *buf1 <= *buf0
        && st011.
st011:- *getd=1,!,
        *buf1 <= *buf0
        && st001.
st011:- *xnack=0,!,
        true,length(1) && st111.
st011:- !,
        true,length(1) && st011.

st111:- *getd=1,*xnack=0,!,
        *buf0 <= *nib, *buf1 <= *buf0
        && st001.
st111:- *getd=1,!,
        *buf0 <= *nib, *buf1 <= *buf0
        && st111.
st111:- !, true,length(1) && st111.
```

Figure 6.17: Behavioral Description of Data Transfer Part (Read-Part)

```
start:- !,initial.
initial:- *rmode=1, !, length(1) && rd000.
initial:- !, length(1) && wr000.

% memory to network data transfer
nobparfrom2:- *slmi=1,!,
        *nob <= *mdatain, *parity <= *mdatain.
nobparfrom2:- *slap=1,!,
        *nob <= *buf1, *parity <= *buf1.
rd000:- *dack=1, ! , true,length(1) && rd100.
rd000:- !, true,length(1) && rd000.

rd100:- *nack=1,*dack=1, !, nobparfrom2  && rd100.
rd100:- *nack=1, !, nobparfrom2 && rd000.
rd100:- *dack=1, !, nobparfrom2 && rd100x.
rd100:- !, nobparfrom2 && rd000x.

% network to memory data transfer
wr000:- *nack=1,!, *buf2 <= *nib && wr100.
wr000:- !, true,length(1) && wr000.

wr100:- *nack=1,!, *nob <= *buf2, *buf2 <= *nib
        && wr101.
wr100:- !, *nob <= *buf2
        && wr001.

wr101:- *dack=1,*valid=1, !, *nob <= *buf2, *buf2 <= *nib
        && wr101.
wr101:- *dack=1,*nack=1,!, *nob <= *buf2, *buf2 <= *nib
        && wr101.
wr101:- *dack=1,!,*nob <= *buf2  && wr001.
wr101:- *valid=1,!,*buf3 <= *nib && wr111a.
wr101:- *nack=1,!, *buf3 <= *nib && wr111a.
wr101:- !, true,length(1) && wr101.
```

Figure 6.18: Behavioral Description of Data Transfer Part (Write-Part)

CPU time (sec)				Number of Derived States
Translator	Facility Checker	Forward Trace	Backward Trace	
0.80	1.07	2.36	40.1	14

Table 6.5: Result of Verifying Data Transfer Part (Read-Part)

CPU time (sec)				Number of Derived States
Translator	Facility Checker	Forward Trace	Backward Trace	
3.23	4.66	183.3	(not measured)	81

Table 6.6: Result of Verifying Data Transfer Part (Write-Part)

Chapter 7

Discussions

In the previous chapters, a logic design assistance system is presented and explained. In this chapter, the performance of this system is evaluated and the practical logic design using the proposed assistance system is discussed. Several topics for future researches are listed finally.

7.1 Performance Evaluation of Control Part Verifier

7.1.1 Speed

As shown in Table 5.2 and Table 5.4, circuits of 20~30 gates have been verified within a few seconds. We discuss the two techniques for increasing the efficiency which is described in section 5.2.3.

Filtering Structural Description

Required CPU time is linearly proportional to both the size of on-, off-covers and the number of the state transitions which appear in the state diagram of (Design $\wedge$ $\sim$ Specification). The size of covers increases exponentially to the number of the input variables. The technique of "filtering structural description" extracts a truly required structure, decrease the number of input variables, and suppresses the size of the covers. In the control part, an output variable is usually a function of a

part of the input variables. In such case, this technique is very efficient because the extracted structure is much smaller than the original structure. The results illustrated in section 5.3 indicate that this technique is quite efficient for the adopted two examples.

Memorization of States

This technique suppresses the number of state transitions which are traced during the verification. On the other hand, this technique enlarges the size of required memory.

The results illustrated in section 5.3 indicate that this technique is not so efficient. The main reason for this is that the state diagram (Design $\wedge$ ~Specification) is small originally. For example, the number of traced states is 6 or 7 for each specification in DMA controller.

This technique becomes efficient when the given specification is complex and the state diagram of (Design $\wedge$ ~Specification) is large.

7.1.2 Size of Required Memory

The size of required memory is small for the adopted two examples as shown in section 5.3. It is the process of making on-, off-covers which requires the largest memory. Here, we estimate how much memory is needed during making covers in the worst case.

Let n be the number of the input variables (including both external input and flip-flop). The number of cubes in on (or off) cover is considered to be 2^{n-1} in the worst case. The worst case happens when the function is exclusive-or.

For example, suppose a logic formula with 4 input. A cover for the formula

$$f1 = x_1x_2x_3x_4 + x_1x_2\overline{x_3}\,\overline{x_4} + x_1\overline{x_2}x_3\overline{x_4} + x_1\overline{x_2}\,\overline{x_3}x_4$$

$$+\overline{x_1}\overline{x_2}x_3x_4 + \overline{x_1}\overline{x_2}\overline{x_3}\overline{x_4} + \overline{x_1}x_2x_3\overline{x_4} + \overline{x_1}x_2\overline{x_3}x_4$$

is as follows.

$$\text{Cover} = \begin{bmatrix} 01 & 01 & 01 & 01 & 1 \\ 01 & 01 & 10 & 10 & 1 \\ 01 & 10 & 01 & 10 & 1 \\ 01 & 10 & 10 & 01 & 1 \\ 10 & 10 & 01 & 01 & 1 \\ 10 & 10 & 10 & 10 & 1 \\ 10 & 01 & 01 & 10 & 1 \\ 10 & 01 & 10 & 01 & 1 \end{bmatrix}$$

In practice, however, the worst case seldom happens, because the logic of control part is rather simple. As for the DMA controller without filtering, the number of input variables is 16, and the number of cubes in on (off) cover is still 46 (59), which is much smaller than 2^{15}. It is the same in the Receiver.

Besides cover expressions, various methods for representing boolean functions have been proposed. Among them, BDD (Binary Decision Diagram) [Bry86] [MIY89] is reported to be very compact style. Using BDD, the size of required memory will be suppressed.

7.2 Performance Evaluation of Data Path Verifier

The performance of each part in the data path verifier is discussed.

7.2.1 Facility Checker

This part extracts the link information between the behavior and the structure. Required CPU time for this part is much smaller than that for the time tracer, except for the small example of computing square root. In other words, performance of this data path verifier is limitted by the power of the time tracer. The facility checker, in this sense, is efficient enough.

The procedure of this extraction is as follows.

1. Find a operator or a set of operators which realizes the operation in each data transfer.

2. Search for data paths from the source register to the input of the operator and those from the output of the operator to the destination register.

Therefore, if a certain operation can be realized by many operators, the cost increases in proportion to the number of the operators. For example, this situation arises in array processors. In such case, we had better to adopt the following procedure.

1. Search for data paths from the source register and select operators to which the source register is connected.

2. Among the selected operators, find a operator or a set of operators which realizes the operation in each data transfer.

3. Search for data paths from the output of the operator to the destination register.

7.2.2 Time Trace

Forward Time Trace

The cost for the forward time trace increases almost proportionally to the products of the number of state transitions and states in the obtained state diagram. Since this process traces all the transitions in the state diagram, its cost is in proportion to the number of the state transitions. Moreover, overheads for checking the halting condition increases proportionally to the number of states.

As for the first example of computing square root, the number of state transitions are small, though it is a pipelined behavior. The second example (general processor) has been also successfully verified. The last example of the network interface processor is an application specific processor. Since its main purpose is to achieve

very high performance, the behavior is complex. The forward time trace have also managed to verify this example.

Therefore, the forward time trace is concluded to have enough power.

Backward Time Trace

In the backward time trace, pairs of intervals which use the same facility are listed at first. Next, all the pairs are verified by tracing the interval transition table backward. Therefore, the number of backward time tracings is ${}_NC_2$ in the worst case, where N is the number of intervals. The worst case occurs when a certain facility is used in all the intervals. A bus often causes the worst case because it is used in almost all the data transfers. In the second example, for example, the structure has 4 busses and required much CPU time for the backward time trace.

The backward time trace can test whether the given two intervals occur simultaneously or not. It is also possible in the backward time trace to specify one interval and search for other intervals which occur simultaneously with the specified one. The general processor has been successfully verified using this technique.

The backward time trace, therefore, is suitable for checking a part of the design where designers are not sure of its correctness. It would be better that some parts of the design are tested partly using the backward time trace at first, and then the whole design are verified using the forward time trace.

7.3 Logic Design using Proposed System

In this section, we discuss the practical logic design using the proposed assistance system.

In the proposed system, the designers, at first, specify the algorithm of the behavior to be designed in Tokio. Next, this description is transformed into another description at the register transfer level. This derivation is processed step by step

with a system realization such as scheduling operations, allocating operators, and so on. This derivation is assured by simulating the behaviors. In this stage, the following three points are realized concerning with the behavioral description language.

- The behavior is specified in an executable form.
- The behaviors at both the algorithmic level and the register transfer level are given in the same language.
- Sequentiality and concurrency can be specified accurately and simply.

The second and the third points are enabled by the simple and accurate semantics of Tokio or temporal logic. Owing to these points, manual derivation is processed smoothly as mentioned in section 4.4. As for the first point, the simulator has been already developed. This simulator is enough powerful as shown in Table 6.2.

Then, the designers also give the structure of the data path to be designed. Both the derived behavior and the given structure are temporary. These should be verified, estimated and improved.

In the verification, consistency between the behavior and the structure is checked. If there exists a contradiction, both the behavior and the structure should be improved. Link information between the behavior and the structure is required for this improvement. This information indicates what parts of the structure are used in order to realize a certain behavior. This information is obtained automatically by the data path verifier.

In the estimation, the behavior should be simulated on the given structure and the frequency in use of a facility is estimated. If there exists a facility of low frequency, it is merged with another facility. Then, the improved structure should be verified. Simulators for this estimation have not been developed yet. This remains to be solved. However, the most required information for the improvement is also the

link information between the behavior and the structure. Therefore, the assistance of deriving these informations automatically is very beneficial.

After the steps of verification, estimation, and improvement have been repeated, the final behavior and structure at the register transfer level are obtained. The design process proceeds into the next stage of gate design. The data path verifier extracts the logic of control part automatically. The derived logic is transformed into networks of logical gates by manual or by a commercial synthesizer. The synthesized control part is verified by the control part verifier.

In this way, logic design is assisted smoothly and effectively in the proposed assistance system.

7.4 Future Work

There are several topics for future researches. Followings are the list of topics.

- In the control part verifier, the BDD (binary decision diagram) is suitable as the internal representation of boolean functions.

- The derivation of RTL-Tokio from Tokio is currently done manually. In order to achieve an efficient and practical assistance of functional design, it is recommended to assist a semi-automatic derivation of the behavior and the structure at the register transfer level. That is, an assistance system presents some alternatives of the derivation, and then designers selects one of them or they do another derivation manually. The system should adopt the techniques of artificial intelligence. In order to realize this assistance, interfaces between designers and assistance systems are one of the most important points.

- A total assistance system should be investigated. The target of the proposed system in this thesis is only the logic design. In the actual design process,

however, requirements to improve the results of the logic design often arise from the implementation design stage. A total assistance system should be investigated in order to meet such requirements which spread over different design stages.

Chapter 8

Conclusion

In this thesis, a logic design assistance system based on temporal logic is presented. There are three key points in this system: how to specify the behavior, how to verify the control part, and how to verify the data path.

In chapter 4, a behavior description language Tokio is presented. Tokio has the following three peculiarities.

- Sequentiality and concurrency can be specified accurately and simply.
- Behaviors are specified in executable forms.
- Behaviors at both the algorithmic level and the register transfer level are given in the same language.

In section 6.3, Three behaviors have been successfully specified which contain concurrency. This is the basis that Tokio is concluded to have the first characteristic. Ackerman(1,1) function has been also successfully simulated on the specified general processor. This is the basis for the second point. As for the third point, a manual derivation of register transfer level behavior is described in section 4.4.

The control part verifier was presented in chapter 5. Structures are verified whether they satisfy given specifications or not. Specifications are given in the form of propositional LTTL. Two examples have been successfully verified.

A data path verifier is presented in chapter 6. The inputs to this verifier are the behavior and the structure at the register transfer level. The following items are realized by this verifier.

- Link informations between the behavior and the structure are derived automatically.
- Consistency between the behavior and the structure is verified automatically.
- The logic of the control part is derived automatically.

This verifier has been applied to three examples successfully.

The proposed assistance system is concluded to have enough power to assist a practical hardware design.

Bibliography

[Aba87] M. Abadi. *Temporal-Logic Theorem Proving.* Technical Report STAN-CS-87-1151, Stanford University, 1987.

[AFT85] T. Aoyagi, M. Fujita, and H. Tanaka. Temporal Logic Programming Language Tokio. In *Logic Programming Conference '85*, pages 128–137, Springer-Verlag, 1985.

[Ayl86] J.H. Aylor. VHDL - Feature Description and Analysis. *IEEE Design and Test*, April, 1986.

[Bar81] M.R. Barbacci. Instruction Set Processor Specification (ISPS): The Notation and its Application. *IEEE Trans. on Computers*, C-30,No.1:24–40, 1981.

[BCDM86] M.C. Browne, E.M. Clarke, D.L. Dill, and B. Mishra. Automatic Verification of Sequential Circuits Using Temporal Logic. *IEEE Trans. on Computers*, C-35,No.12:1035–1044, 1986.

[BD88] G Borriello and E Detjens. High-Level Synthesis: Current Status and Future Directions. In *25th Design Automation Conference*, pages 477–482, ACM/IEEE, 1988.

[BMHS84] R.K. Brayton, C. McMullen, G.D. Hachtei, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer

Academic Publishers, 1984.

[Bry86] R.E. Bryant. Graph-Based Algorithm for Boolean Function Manipulation. *IEEE Trans. on Computers*, C-35,No.8:677–691, 1986.

[Cam87] R. Camposano. Structural synthesis in the yorktown silicon compiler. In *VLSI '87*, pages 61–72, IFIP, 1987.

[Cam88] R. Camposano. Design Process Model in the Yorktown Silicon Compiler. In *25th Design Automation Conference*, pages 489–494, ACM/IEEE, 1988.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, Vol8, No.2:244–263, 1986.

[CW88] M. Carlsson and J. Widen. *SICStus Prolog Users Manual.* 1988.

[DBS85] G. De Micheli, R.K. Brayton, and A. Sangiovanni-Vincentelli. Optimal State Assignment For Finite State Machines. *IEEE Trans. on CAD*, CAD-5,7:269–285, 1985.

[DD68] J.R. Duley and D.L. Dietmeyer. A Digital System Design Language DDL. *IEEE Trans. on Computers*, C-17,No.9:850–861, 1968.

[FKTM86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Aid to Hierarchical and Structured Logic Design using Temporal Logic and Prolog. In *Proceedings.Pt.E*, pages 283–294, IEE, 1986.

[FMF89] M. Fujita, Y. Matsunaga, and H. Fujisawa. On the Application of Binary Decision Diagrams to Formal Hardware Design. In *IMEC-IFIP*

International Workshop on Applied Formal Methods For Correct VLSI Design, pages 246–260, 1989.

[FTM83] M. Fujita, H. Tanaka, and T. Moto-oka. Verification with prolog and temporal logic. In *CHDL '83*, IFIP, 1983.

[FTM85] M. Fujita, H. Tanaka, and T. Moto-oka. Logic design assistance with temporal logic. In *CHDL '85*, pages 129–138, IFIP, 1985.

[HM88] G.D. Hachtel and P.H. Moceyunas. Algorithm for multi-level tautology and equivalence. In *International Symposium on Circuits and Systems*, June 1988.

[KAFT85] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of Temporal Logic Programming Language Tokio. In *Logic Programming Conference '85*, pages 138–147, Springer-Verlag, 1985.

[Kar89] O. Karatsu. VLSI Design Language Standardization Effort in Japan. In *26th Design Automation Conference*, ACM/IEEE, 1989.

[KT88] H. Koike and H. Tanaka. Multi-Context Procesing and Data Balancing Mechanism of the Parallel Inference Machine PIE64. In *Fifth Generation Computer Systems*, pages 970–977, ICOT, 1988.

[Man81] Z. Manna. *Verification of Sequential Programs: Temporal Axiomatization.* Technical Report STAN-CS-81-877, Stanford University, 1981.

[MIY89] S. Minato, N. Ishiura, and S. Yajima. Fast Tautology Checking Using Binary Decision Diagram - Benchmark Results. In *IMEC-IFIP International Workshop on Applied Formal Methods For Correct VLSI Design*, pages 580–584, 1989.

[Mos83] B. Moszkowski. A Temporal Logic for Multi-Level reasoning about Hardware. In *CHDL '83*, IFIP, 1983.

[MP81] Z. Manna and A. Pnueli. *Verification of Concurrent Programs Part1. The Temporal Framework.* Technical Report STAN-CS-81-836, Stanford University, 1981.

[MPC88] M.C. McFarland, A.C. Parker, and R. Composano. Tutorial on High-Level Synthesis. In *25th Design Automation Conference*, pages 330–336, ACM/IEEE, 1988.

[Nak87] H. Nakamura. *Fast Logic Design Verification System Based on Temporal Logic.* Master's thesis, University of Tokyo, February 1987.

[NFKT87] H. Nakamura, M. Fujita, S. Kono, and H. Tanaka. Temporal Logic Based Fast Verification systems Using Cover Expressions. In *VLSI '87*, pages 99–111, IFIP, 1987.

[Par84] A.C. Parker. Automated Synthesis of Digital Systems. *IEEE Design and Test*, November:75–81, 1984.

[PK86] P.G. Paulin and J.P. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *24th Design Automation Conference*, pages 195–202, ACM/IEEE, 1986.

[PPM86] A.C. Parker, J.T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *23rd Design Automation Conference*, pages 461–466, ACM/IEEE, 1986.

[Sas84] T. Sasao. Input Variable Assignment and Output Phase Optimization of PLA's. *IEEE Trans. on Computer*, C-33,No.10:879–894, 1984.

[Sha89] M. Shahdad. An Overview of VHDL Language and Technology. In *23rd Design Automation Conference*, pages 320–326, ACM/IEEE, 1989.

[SKS80] Y. Sugiyama, O. Karatsu, and T. Sudo. *VLSI Design System (in Japanese).* Technical Report 7-2, Monograph of Technical Group on Design Technology of Electronics Equipment IPSJ, December 1980.

[SKT89] T. Shimizu, H. Koike, and H. Tanaka. *Inter-pe Communication of the Parallel Inference Machine PIE64 (in Japanese).* Technical Report CA-79-4, Information Processing Society of Japan, 1989.

[SL89] I. Suzuki and H. Lu. Temporal Petri Nets and Their Application to Modeling and Analysis of a Handshake Daisy Chain Arbiter. *IEEE Trans. on Computers*, C-38,No.5:696–704, 1989.

[Tri87] H. Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Trans. on CAD*, CAD-6,2:259–269, 1987.

[Use] *User Device Design Manual for PANAFACOM U-series.*

[Van77] W.M. VanCleemput. A Hierarchical Language for the Structural Description of Digital Systems. In *14th Design Automation Conference*, ACM/IEEE, 1977.

[Wax86] R. Waxman. The VHSIC Hardware Description Language - A Glimpse of the Future. *IEEE Design and Test*, April, 1986.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications.* Technical Report STAN-CS-82-925, Stanford University, 1982.

[WS86] R. Wei and A. Sangiovanni-Vincentelli. Proteus: A Logic Verification System for Combinational Circuits. In *International Test Conference*, IEEE, 1986.

[Zub80] W.M. Zuberek. Timed Petri nets and preliminary performance evaluation. In *7th Symposium on Computer Architecture*, pages 88–96, 1980.

発表文献

1. 論文誌

(1) 中村宏，藤田昌宏，河野真治，田中英彦：”時相論理に基づく論理回路検証システム”, 情報処理学会論文誌，1989，6月号

(2) 中村宏，久木元裕治，田中英彦: ” 時相論理を動作記述に用いたデータパス検証システム”,(投稿中)

2. 国際会議

(3) Nakamura,H., Fujita,M., Kono,S. and Tanaka,H. : ”Temporal Logic Based Fast Verification System Using Cover Expressions”, VLSI '87, IFIP, pp99-111,Vancouver, August,1987

(4) Nakamura,H., Fujita,M., Kono,S., Nakai,M. and Tanaka,H.: ” A Data Path Verification System using Temporal Logic Based Language: Tokio”, IFIP WG10.2 Working Conference on the CAD Systems Using AI Techniques, Tokyo, June,1989

(5) Nakamura,H.,Kukimoto,Y.,Fujita,M. and Tanaka,H.: ”An Assistance System for Effective Register Level Synthesis Using Temporal Logic Language Tokio” (Submitted for publication)

3. 国内会議

(6) 藤田昌宏, 石曽根信, 中村宏, 田中英彦, 元岡達: ”時相論理型言語:Tokio によるアルゴリズム記述と CMOS ゲートアレイの自動合成”, ICOT,Logic Programming Conference'85,1985

(7) 中村宏，中井正弥，河野真治，藤田昌宏, 田中英彦: ”時相論理型言語 Tokio による論理設計支援”, ICOT,Logic Programming Conference'89, July,1989

4. 全国大会

(8) 中村宏, 藤田昌宏, 田中英彦, 元岡達:”時相論理型言語 Tokio によるハードウェア記述”, 情報処理学会第 31 回 (昭和 60 年後期) 全国大会,2J-9”,1985

(9) 中村宏, 藤田昌宏, 田中英彦, 元岡達:”Tokio による論理回路の検証 1- 高速命題論理検証系の実装 -”, 情報処理学会第 32 回 (昭和 61 年前期) 全国大会, 5U-2, 1986

(10) 中村宏, 藤田昌宏, 田中英彦: ”時相論理型言語 Tokio による論理回路検証系の評価”, 情報処理学会第 33 回 (昭和 61 年後期) 全国大会,6Q-10, 1986

(11) 河野真治, 中村宏, 藤田昌宏, 田中英彦: ”時相論理型言語 Tokio によるハードウェア記述 - 時間に依存する fact による同期”, 情報処理学会第 33 回 (昭和 61 年後期) 全国大会,6Q-9,1986

(12) 中村宏, 河野真治, 藤田昌宏, 田中英彦: ”Tokio によるレジスタトランスファレベル記述からの論理回路の自動合成”, 情報処理学会第 34 回 (昭和 62 年前期) 全国大会,1F-2, 1987

(13) 河野真治, 中村宏, 藤田昌宏, 田中英彦: ”Tokio による機能記述開発支援ツール”, 情報処理学会第 34 回 (昭和 62 年前期) 全国大会,1F-1, 1987

(14) 中村宏, 藤田昌宏, 河野真治, 田中英彦: ”Tokio によるレジスタトランスファレベル記述の動作解析ツール”, 情報処理学会第 35 回 (昭和 62 年期) 全国大会, 5F-3,1987

(15) 河野真治, 中村宏, 藤田昌宏, 田中英彦: "時相論理型言語 Tokio の処理系に関する考察", 情報処理学会第 35 回 (昭和 62 年後期) 全国大会,5F-4, 1987

(16) 中村宏, 河野真治, 藤田昌宏, 田中英彦: "Tokio に基づくパイプライン化支援ツールの実装", 情報処理学会第 36 回 (昭和 63 年前期) 全国大会,2X-7", 1988

(17) 河野真治, 中村宏, 藤田昌宏, 田中英彦: "時相論理型言語 Tokio-ITL における Unification の考察", 情報処理学会第 36 回 (昭和 63 年前期) 全国大会, 2X-8,1988

(18) 中村宏, 藤田昌宏, 河野真治, 田中英彦: "RTL-Tokio: レジスタトランスファレベル動作記述言語", 情報処理学会第 37 回 (昭和 63 年後期) 全国大会, 1U-3,1988

(19) 中村宏, 河野真治, 中井正弥, 藤田昌宏, 田中英彦: "RTL-Tokio に基づくパイプライン化支援", 情報処理学会第 38 回 (平成元年前期) 全国大会, 3S-10,1989

(20) 河野真治, 中村宏, 田中英彦: "時相論理型言語 Tokio における非決定的実行", 情報処理学会第 38 回 (平成元年前期) 全国大会,4S-5, 1989

(21) 中井正弥, 中村宏, 河野真治, 田中英彦: "時相論理型言語 Tokio に基づく論理設計支援システム", 情報処理学会第 38 回 (平成元年前期) 全国大会, 3S-9,1989

(22) 中村宏, 久木元裕治, 田中英彦: "RTL-Tokio 記述からの制御系の導出", 情報処理学会第 39 回 (平成元年後期) 全国大会, 3X-4,1989

(23) 久木元裕治, 中村宏, 田中英彦: "Tokio に基づくデータパス検証システムの評価", 情報処理学会第 40 回 (平成 2 年前期) 全国大会, (発表予定)

5. 研究会

(24) 中村宏 , 河野真治, 藤田昌宏, 田中英彦: "Tokio に基づく論理回路の検証", 情報処理学会設計自動化研究会 86-DA-34-1,1986

(25) 中村宏 , 藤田昌宏, 河野真治, 田中英彦: "時相論理に基づく論理回路高速検証法", 電子通信学会第 17 回 FTC 研究会, 1987

(26) 藤田昌宏, 藤沢久典, 中村宏, 田中英彦: "Tokio による論理設計支援システム", 情報処理学会設計自動化研究会 87-DA-40-18, 1987

(27) 久木元裕治, 中村宏, 田中英彦: "時相論理型言語 Tokio に基づくデータパス検証", 電子通信学会第 22 回 FTC 研究会, 1990 (発表予定)