Formal Verification of High-Level Design Based on Control/Data Separation

高位設計に対する制御と演算の分離による形式的検証

A Dissertation
Submitted to the Graduate School
of the University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Electronic Engineering

Supervisor: Prof. Masahiro Fujita

Thesis submitted: December 15th, 2009

Tasuku Nishihara

Abstract

With the size increase of VLSI, current designs are firstly written in high-levels, such as system-level, behavioral level, or register transfer level (RTL). High-level designs are typically verified by simulation. However, since simulation can only check patterns being input, some design bugs in corner cases may not be detected with it. Then, formal verification is used as a complement of simulation for such a case.

Currently, two problems can be considered on formal verification of high-level design. One is about performances of verification methods and tools. The other is the high-barrier to apply formal verification methods to actual designs. In this thesis, four methods are proposed for these problems. The first two methods improve verification performances, and the other two methods related to interfaces or preprocesses of formal verification methods. The first two methods are based on an approach which separates control and data portions in designs. Then, control portions and data portions can be analyzed separately, and word-level methods such as symbolic simulation can be applied effectively.

The first proposed method improves bounded model checking by decomposing one large bounded model checking into small pieces. It is very difficult for traditional bounded model checking methods which can only be verified with short bounds to detect deep bugs. In the proposed method, since the bound of each decomposed bounded model checking is small, the computation amount can be dramatically reduced in successful cases. In addition, symbolic simulation is applied to a control path of each counter example to support the connections between those decomposed bounded model checkings. When a connection fails, the former bounded model checking is retried after refining the condition not to get similar counter examples.

Experimental results showed that the proposed method can improve the performance of bounded model checking even with the simplest two-level method.

The second method proposed in this thesis improves equivalence checking between designs before and after behavioral optimization or high-level synthesis. The proposed method applies a preprocess that makes the data portions of the target designs identical. This is performed by tentatively synthesizing behavioral designs into virtual controllers and virtual datapaths. When the target designs are designs before and after high-level synthesis, the virtual datapath is identical to the datapath of the RTL design. When datapaths of two designs are identical, the same control signals are guaranteed to be equivalent in bit-level. Then such control signals can be replaced with uninterpreted functions, and word-level equivalence checking techniques can be applied with bit-level accuracy. In addition, a word-level rule-based equivalence checking method is proposed. The method uses pre-defined rules of equivalence to propagate input equivalences which are given by users to outputs. Since the rule based approach topologically traverses control FSMs, designs which include many conditional branches and loops can be verified faster than symbolic simulation based methods.

The third method proposed in this thesis is a preprocess for hardware/software co-design to solve the three problems in formal verification of hardware/software co-design in lower level than system-level, such as their size, the difference of hardware and software representations, and the interactions between hardware and software portions. The proposed method translates both hardware and software portions into a set of concurrent Finite State Machine with Datapaths (FSMDs). After the translation, the interactions between hardware and software portions are abstracted. Then, a sequentialization method which converts concurrent FSMDs into a single sequential FSMD and handles interruptions is applied. After the sequentialization, control states which do not have data dependences each other are merged. The experimental results showed that the proposed method could make formal verification more than 20 times faster than existing methods.

The last method proposed in this thesis is an useful intermediate representation of high-level designs for verification. In the proposed intermediate representation ExSDG, complicated syntax elements and structures are removed in preprocess steps. Since ExSDG has different representation levels correspond to untimed behavioral level, timed behavioral level, and register transfer level, respectively, various existing design representations in high-level can be directly translated into ExSDG. Therefore, verification tool developers only have to deal with ExSDG to support those representations. In addition, System Dependence Graph (SDG) and Control Flow Graph are integrated with Abstracted Syntax Tree (AST) in ExSDG, and users can directly access such information from the AST tree. An SDG edge shows a dependency relation between two portions of a design. Many researches use ExSDG as a tool implementation environment, and this fact shows the effectiveness of ExSDG.

With the four methods proposed in this thesis, formal verification in high-level can achieve more performances, wider range of designs can be verified with them, and tool implementations of them will be easier.

Acknowledgments

This thesis would not be completed without the kind support of my family, friends, and colleagues in my laboratory. I would like to thank all people who relate to my research especially the following people.

First of all, I wish to give grateful thanks to my supervisor Prof. Masahiro Fujita. I believe I could have the best graduate student experiences with this computer aided design field and this laboratory. He gave me many things, not only so many pieces of important advice to my research but also number of great chances which must never be experienced out of this laboratory, such as valuable internships, joint works with big companies, and presentations to and discussions with famous professors. All pieces of his advice were always smart, and each of them is a base of this thesis. I am lucky to have been supervised by him, and I will continue to make the best effort not to injure his honor.

Second, I would like to thank Prof. Takeshi Chikayama, Prof. Hitoshi Aida, Prof. Shuichi Sakai, Prof. Masaya Nakayama, and Prof. Masahiro Goshima who are the the members of the thesis committee. They gave me a plenty of valuable comments to refine this thesis. After the pre-defense, I have made the best effort to satisfy their requirements.

I would like to give the best "thank you" to Prof. Takeshi Matsumoto who is an assistant professor of the laboratory. He is the second author of most of my publications, and I can say that the second author of this thesis is also him since many portions of this thesis are formed of his valuable advice. As far as I know, he has been making tremendous efforts to organize the laboratory well even from when he was a master student, and I may be a student who received the most benefit from his efforts. He has also been a kindhearted senior student for me and other students, and I thank everything to him in my student life at the laboratory. I really hope and believe his future success in the academic field.

I am deeply grateful to thank three senior members of the laboratory who are currently working at the academic field. Prof. Satoshi Komatsu was the former assistant professor of the laboratory, and also made huge efforts to organize the laboratory. The first three years of my student life was fully under his support, and I really thank to his advice to my researches as well as the support. I would like to thank Prof. Kenshu Seto who was a postdoctoral fellow of the laboratory. He is really a kind person, and he spent a lot of time to care students including me. I got many pieces of meaningful advice about researches from him. I would like to thank Prof. Hiroaki Yoshida who is an assistant professor of the laboratory, and he has also been spending a lot of time to care students and organize the laboratory. He often gives me many pieces of advice not only about researches and developments but also about behaviors and efforts to be a good husband.

My laboratory has many foreign members. Communications and collaborations with them were really meaningful and valuable experiences for me. Dr. Bijan Alizadeh Malafeh and Dr. Amir Masoud Gharehbaghi are postdoctoral fellows from Iran. Dr. Thanyapat Sakunkonchak was a postdoctoral fellow from Thailand. Dr. Yu Liu and Dr. Shanghua Gao were senior Ph. D students from China, Ratna Krishnamoorthy is a Ph. D student from India, and Jiayi Zhang is a Ph. D student from China. Fei Gao, Jinmei Xu were Master students from China, and Jaeho Lee is a Master student from Korea. I would like to thank all of them for giving me such special experiences and having enjoyable discussions about cultures as well as researches.

The six years I spent in the laboratory were also meaningful term to train software development skills. I learned most parts of such skills from senior students of the laboratory. Yoshihisa Kojima was a senior Ph. D student in the laboratory, and I was always helped by his wonderful skills about software development. The knowledge I got from him helped me a lot on implementing tools for the research experiments. Ken Tanabe was a Master student in the laboratory, and I succeeded his software development project of SpecC program slicer. I learned many programming techniques from his code. Shunsuke Sasaki was a senior Master student in the laboratory, and the development project of SpecC program slicer was a joint work with him. I also learned some knowledge about program slicing from him. Ken Matsui was a senior Master student, and he has a great knowledge about design modeling with UML. I learned most of my knowledge about UML from him. I would like to give thanks to all of them.

Discussions and competitions with same grade and junior students in the laboratory were really great stimuli for keeping high motivation which is important to make good research results. Shota Watanabe was a same grade student in the laboratory when I was a master course student, and he was really a smart student. Ken Ishii was also a same grade student in the laboratory when I was in the undergraduate course. Daisuke Ando and Yuji Ishikawa were one year junior Master students in the laboratory, and their warm atmosphere made the laboratory members happy. Yeonbok Lee is also a one year junior Ph. D student in the laboratory, and her earnest research style has been making good effects to other students. Hideo Tanida is a junior Master student. He is currently making a lot of efforts to administrate the laboratory computer systems. Takaaki Tagawa and Hiroki Harada are also junior Master students, and I hope they will make good results in their researches as main members of the laboratory. Altanbileg Adyiasuren, Hiroyuki Murata, Satoshi Morishita, Tatsuya Kurano, and Yuki Nakai are previous and current undergraduate students in the laboratory. I would like to thank all of them, and I hope their future success in their fields.

Office procedures about travels and research budgets are also mandatory to continue research activities. Tomoko Tate, Meiko Houda, Norie Hiraide, Mie Funabiki, Shinobu Nagasawa kindly supported me on such issues, and I would like to thank them. They sometimes had joyful tea time breaks with the research members, and I really liked such events.

Finally I would like to thank to my fiancee, Nozomi Ogawa. She continuously encouraged me during these five years, and that gave me a strong will and confidence to achieve all tough goals and overcome all difficulties.

Contents

Abstract **i**

Acknowledgments **iv**

1 Introduction **1**

1.1 High-Level Design and Representations 2

1.2 Requirement of Formal Verification 3

1.3 Control/Data Separated Formal Verification 4

1.4 Formal Verification of Hardware/Software Co-Design 6

1.5 Efficient Implementation of Formal Verification Methods for High-Level Designs 8

1.6 Overall Flow of the Proposed Methods 8

1.7 Organization 10

2 Preliminaries and Basic Notions **11**

2.1 Formal Verification Techniques 12

2.1.1 Model Checking 12

2.1.2 Equivalence Checking 17

2.2 Control and Data in High-Level Design 22

2.3 Control/Data Separated Design Representations 29

2.3.1 Finite State Machine with Datapath 29

2.3.2 Definition of Other Notions Related to FSMD 34

2.3.3 System Dependence Graph 35

2.4 Symbolic Simulation 45

3 Multi-Level Bounded Model Checking 50
3.1 Introduction 51
3.2 Basic Bounded Model Checking Algorithm 54
3.3 Concatenation of Two Bounded Model Checking Results 58
3.3.1 Bounded Model Checking without the Initial State Condition 58
3.3.2 Reachability Analysis to the Counter Example 62
3.3.3 Algorithm 65
3.3.4 Example 65
3.4 Two Level Bounded Model Checking 70
3.4.1 Generation of Condition to Enter Counter Example Path and Violate Property 70
3.4.2 Replacement of Timed Symbolic Values 72
3.4.3 Construction of Complete Counter Example 74
3.4.4 Algorithm 75
3.4.5 Example 75
3.5 Multi-Level Bounded Model Checking 80
3.5.1 Division of the Second Step 80
3.5.2 Deciding Levels and Bounds 82
3.5.3 Combination with Simulation 82
3.6 Experimental Results 83
3.6.1 Examples 83
3.6.2 Results 84
3.7 Conclusion 85

4 Equivalence Checking with Synthesizing Designs onto Identical Datapath 86
4.1 Introduction 87
4.2 Basic Notions 88
4.2.1 Separation of Designs' Equivalence to that of Controllers and Datapaths 88
4.2.2 NISC(No Instruction Set Computer) Compiler 92
4.3 Generation of RTL Designs with Identical Datapath 93

4.3.1 Verification Flow 93
4.3.2 Generation of a Controller for a Given Datapath 95
4.4 Equivalence Checking of RTL models which have same datapaths . . 96
4.4.1 Equivalence Checking in Bit-Level Accuracy 96
4.4.2 Equivalence Checking of Different Control Signals 97
4.4.3 Input of Equivalence Checking 100
4.4.4 Definition of Equivalence 102
4.4.5 Equivalence Checking of Symbolic Expressions 106
4.4.6 Equivalence Checking of FSMDs by Symbolic Simulation . . . 107
4.4.7 Rule-Based Equivalence Propagation 110
4.5 Experimental Results 122
4.5.1 Tool Implementations 123
4.5.2 Examples 123
4.5.3 Verification Results 125
4.6 Conclusion 126

5 Formal Verification of Hardware/Software Co-Design with Translation into FSMD 127
5.1 Introduction 128
5.2 Basic Notions 132
5.2.1 Modeling of Connection between Hardware and Software . . . 132
5.2.2 Sequentialization of Concurrent Processes with Race Condition Verification 133
5.3 Formal Verification Method Based on Translation into FSMD 138
5.3.1 Abstraction of Communication between Hardware and Software 140
5.3.2 Translation into FSMD 141
5.3.3 Definition of Synchronization Point and Its Detection 144
5.3.4 Applying Sequentialization 147
5.3.5 State Merging with Data Dependence Analysis 155
5.3.6 Comparison with Partial Order Reduction 157
5.3.7 Limitation 158
5.4 Experimental Results 160

5.5 Conclusion and Future Work 163

6 ExSDG: Design Representation for Efficient High-Level Design Verification 164

6.1 Introduction 165
6.2 Verification Framework 168
6.3 Language of the AST in ExSDG 170
6.3.1 Basic Syntax 170
6.3.2 Three Representations in Different Levels 171
6.3.3 Translation from Other Languages 174
6.3.4 AST Simplification 178
6.4 Extended System Dependence Graph 184
6.4.1 Nodes in ExSDG 184
6.4.2 Edges in ExSDG 185
6.4.3 Example of ExSDG 187
6.4.4 Experimental Result 187
6.5 Verification methods using SDG 188
6.5.1 Design Error Detection 188
6.5.2 Synchronization Verification 189
6.5.3 Equivalence Checking 189
6.6 Conclusion 190

7 Conclusion and Future Work 191

7.1 Conclusion 192
7.2 Future Work 194

Publications 209

List of Figures

1.1 Design flow from system level 3
1.2 Control and data portions in RTL/behavioral level design and program Code 5
1.3 Multi-level bounded model checking 6
1.4 Translation into models having identical datapaths 7
1.5 Overall formal verification flow using proposed methods in this thesis 9

2.1 Basic model checking flow 13
2.2 Binary Decision Diagram 15
2.3 Design refinement and synthesis flow 17
2.4 Miter Circuit 18
2.5 Miter circuit of sequential designs 19
2.6 Conversion of sequential miter circuit into combinational 20
2.7 Example C code to show the design element classification 23
2.8 Control of the example in Figure 2.7 24
2.9 Controller and datapath 25
2.10 Behavioral flow of controller and datapath 26
2.11 An example of FSMD with Definition 6 34
2.12 Example program for dependence graph 36
2.13 Dependence graph of the code in Figure 2.12 37
2.14 Example program for SDG 39
2.15 SDG of the code in Figure 2.14 40
2.16 Example concurrent program for SDG 41
2.17 SDG of the code in Figure 2.16 41

2.18 Example SpecC code 44
2.19 Structure of the code in Figure 2.18 45
2.20 Symbolic simulation result for the path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1$ in Figure 2.11 48

3.1 Multi-Level Bounded Model Checking 52
3.2 Concatenation of two bounded model checking results 59
3.3 State space image of the first step of two bounded model checking results concatenation 61
3.4 State space image of the second step of two bounded model checking results concatenation 64
3.5 Toy FSMD example and property 68
3.6 Two level bounded model checking flow 71
3.7 State space image of two level bounded model checking 72
3.8 Symbolic simulation result on the path $s_f \rightarrow s_f \rightarrow s_f$ at the design in Figure 3.5 77
3.9 Symbolic simulation result on the path $s_c \rightarrow s_e \rightarrow s_f$ at the design in Figure 3.5 78
3.10 State space image of the multi-level bounded model checking (four level) 82

4.1 An example of behavioral description [50] 89
4.2 An example of RTL description [50] 90
4.3 An example of controller and datapath [50] 91
4.4 Comparison between RISC and NISC architectures [136] 92
4.5 Proposed equivalence checking flow between a behavioral level design and an RTL design 94
4.6 Proposed equivalence checking flow between two behavioral level designs 95
4.7 An example of datapath for bit-level analysis 99
4.8 A set of control signals and the generated combinational circuit . . . 100
4.9 The other set of control signals and the generated combinational circuits 101
4.10 The FSMD of the design in Figure 4.2 101
4.11 An output of NISC compiler and the corresponding FSMD 102

4.12 Example for Equivalence Definitions 104
4.13 Example 1 108
4.14 Example 2 113
4.15 Example 3 115
4.16 Example 4 115
4.17 Loop unrolling for the proof of Rule 5 117

5.1 Typical hardware/software co-design implementation 128
5.2 Hardware/software co-simulation 129
5.3 Memory Mapped I/O 132
5.4 Interruption Driven I/O 133
5.5 Example SpecC code having race condition 134
5.6 Example SpecC code for sequentialization 137
5.7 Sequentialized code from Figure 5.6 137
5.8 Proposed Verification Flow 139
5.9 Example Software code in C 141
5.10 Example hardware code in Verilog-HDL 142
5.11 Software code after memory map replacement from Figure 5.9 143
5.12 Pointer removal with point-to analysis 143
5.13 Example FSMDs of HW/SW Co-Design 144
5.14 Synchronization Points on the FSMDs in Figure 5.13 147
5.15 Design using OCP and its synchronization points 147
5.16 Synchronization related portion removal from Figure 5.13 149
5.17 Sequentialization Result of the FSMDs in Figure 5.13 154
5.18 Sequentialization and State Merging Results of the Design using OCP in Figure 5.15 155
5.19 Data Dependences of the FSMD in Figure 5.20 156
5.20 State Merging Result of the FSMD in Figure 5.17 157

6.1 Proposed Verification flow 169
6.2 Example code in untimed behavior level 172
6.3 Example code in timed behavior level 173
6.4 Example code in register transfer level 175

6.5 Difference of concurrent process execution in ExSDG and SystemC . 177
6.6 Example code in SystemC 178
6.7 ExSDG code translated from Figure 6.6 179
6.8 Example code in Verilog-HDL equivalent to Figure 6.4 180
6.9 Conversion from channels and interfaces 181
6.10 ExSDG code translated from Figure 6.6 182
6.11 Replacement of **for** statement with **while** statement 183
6.12 Replacement of **switch** statement with **if-else** statement 184
6.13 AST structure of ExSDG 185
6.14 Example design 187
6.15 ExSDG of the design in Figure 6.14 188

List of Tables

2.1 Data of the example in Figure 2.7 25

3.1 Size of the examples 84
3.2 Verified properties 84
3.3 Results of the proposed method and simple bounded model checking . 85

4.1 Information of Examples 124
4.2 Verification time of rule-based equivalence propagation 126

5.1 Memory map for the design in Figure 5.9 140
5.2 Examples 161
5.3 Result of Sequentialization 162
5.4 Results of Model Checking with SPIN 163

6.1 Syntax correspondence between SystemC and ExSDG 177
6.2 Syntax correspondence between Verilog-HDL and ExSDG 179
6.3 Node types in the AST 185
6.4 Node types in the SDG 186
6.5 Types of edges in SDG 186
6.6 Experimental results in SDG generation of IDCT 187

Chapter 1

Introduction

1.1 High-Level Design and Representations

With the size increase on VLSI, said as Moore's Law, the standard starting stages of VLSI designs has been moving from low level design stages, where designs are precisely described, to high-level design stages where designs are written in more abstract form, such as register transfer level (RTL), behavioral level, and system level.

RTL is a design stage that computations performed at each clock cycle are defined typically in bit-level. Here, bit-level means that bit-widths of registers and computation units are decided and described exactly. In RTL design, usually computation resources (Number of computation units and registers, bus structures) are fixed. RTL designs are written in Hardware Description Languages (HDLs), such as Verilog-HDL and VHDL.

Behavioral level is another design stage more abstract than RTL where only behaviors of designs are defined and neither execution timing nor clock cycle is considered. Behavioral descriptions look like software program codes, and typically C-based languages, such as SystemC[160], SpecC[54], SystemVerilog[82], BachC[177], BDL[172], and CataplutC[28] are used to describe them. In behavioral level designs, behaviors can be defined both in word-level and bit-level. Here, word-level means that bit-widths of variables and operators are not precisely considered or ignored. For example, if a designer writes a 4-bit multiplication in C code, the designer has to write is as an 8-bit multiplication since only 8, 16 or 32 are available bit-widths in C. Then, such descriptions include unnecessary bit computations. Therefore, word-level descriptions may not exactly describe hardware behaviors.

In system level, hardware and software are not separated, and written in a same language, typically C-based languages like behavioral level. Then, hardware/software partitioning can be flexibly explored. The difference from behavioral level is only the concept that system level descriptions include both hardware and software portions, and the abstraction level of behaviors is same.

Figure 1.1 shows the typical design flow of hardware/software co-designs from system-level. Firstly, designs are written in system level without separating hardware and software portions. After architecture exploration, designs are separated to

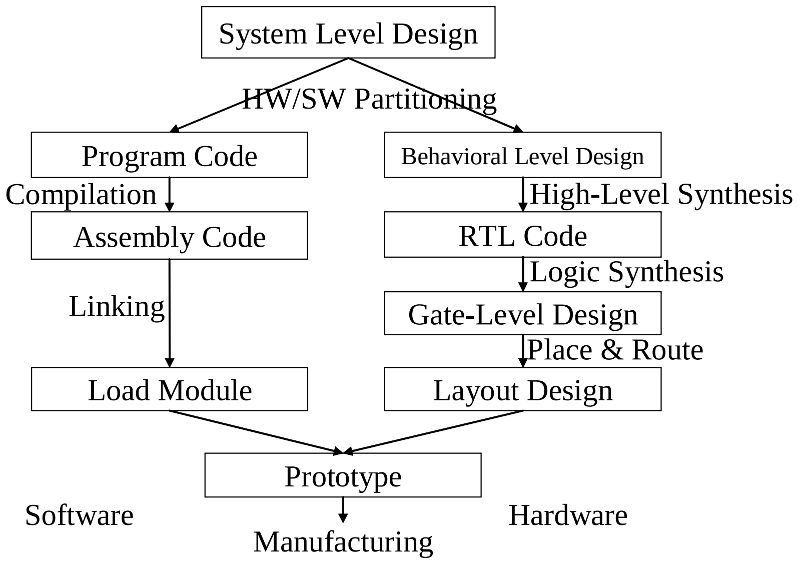


Figure 1.1: Design flow from system level

hardware and software portions, and software portions are converted step by step down to load modules which can be directly run on processors. Hardware portions are also converted to layouts through high-level synthesis, logic synthesis, and place and route.

Currently, the design flow of hardware/software co-design shown in Figure 1.1 is still in a research stage, and most practical designs are started from the stages after hardware/software partitioning, such as program code/behavioral level description or program code/RTL code. Hardware portions of such design stages are called "high-level", and then those design stages are defined as "high-level" of hardware/software co-design in this thesis.

This thesis focuses on verification of such high-level hardware/software co-designs.

1.2 Requirement of Formal Verification

Design verification which is a process that finds and fixes design bugs is important since more than 80% of the whole design period is said to be used for the verification process[43]. Therefore, if the verification period is shorten, it also contribute to shorten the whole design period.

For that purpose, it is important to find and fix design bugs exhaustively in high-level since finding and fixing design bugs are performed easier in more abstract stages. Costs to go back to the earlier design stages when bugs are found are also smaller.

Most parts of current design verification are performed by simulation. Here, simulation is defined as a process that runs a design with a given input sequence and compares its output with a reference. Simulation is fast, but only one input sequence can be checked per one execution. In current practical designs, it is quite difficult to simulate all possible input sequences exhaustively, since numbers of considerable inputs are huge in most designs because of large design sizes.

Formal verification is a strong technique to verify designs exhaustively. In formal verification, designs are analyzed with mathematical techniques. Since results of formal verification is valid for all input sequences (details of current formal verification techniques are introduced in Section 2.1), formal verification techniques have been used in the verification process as an important step. However, the computation amounts of formal verification methods typically increase exponentially with design sizes. Therefore, it is strongly required to improve the performance of formal verification techniques.

Formal verification techniques can be classified into two categories, model checking and equivalence checking. Model checking decides whether a given specification of a design (called property) is satisfied on the design. Equivalence checking decides whether two given designs are equivalent on a given criterion. In this thesis, both model checking and equivalence checking methods are investigated.

1.3 Control/Data Separated Formal Verification

To improve the performance of formal verification techniques, this thesis focuses on control and data portions of designs. A control portion is the flow of the behaviors of a design, and a data portion is the set of actual computations executed at the control steps. For example, in an RTL design control portions are control Finite State Machines (FSMs), and the data portion is a datapath including computation units as shown in Figure 1.2. In a behavioral design or a software program code,

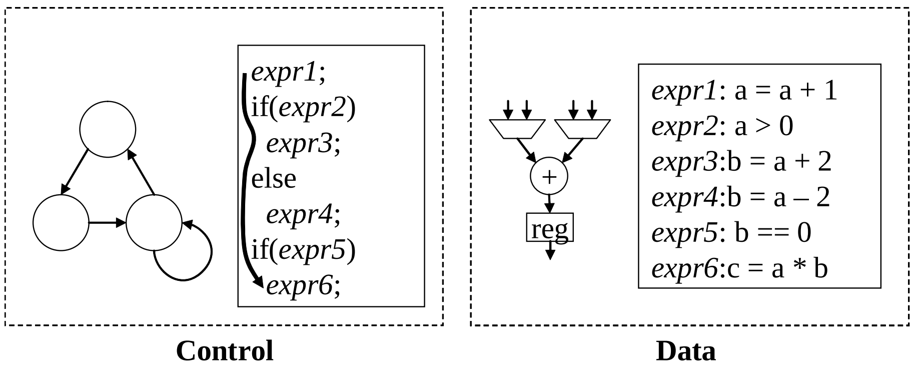


Figure 1.2: Control and data portions in RTL/behavioral level design and program Code

control is a control flow with conditional branches and loops, and data is a set of expressions executed at the control steps which mean the points in the control flow. The formal definition and details about control and data portions of designs are explained in Section 2.2.

The main idea of this thesis is to separate control and data portions in designs. When control paths of different executions are same, computed formulae on those executions are identical, and the difference is the used data for the computations. Such a feature can be utilized on formal verification, and two formal verification methods are mainly proposed in this thesis.

One of the proposed methods in Chapter 3 is a model checking method that concatenates multiple bounded model checking results as shown in Figure 1.3. Bounded model checking[22] is a model checking method that restricts the state space to be verified with a given number of transitions from the initial states, called bound. Since it does not handle infinite state space, the problem can be translated into satisfiability (SAT) problem, and efficient SAT solvers can be used to solve it. However, since its computation amount increases exponentially with the bound, it is difficult to detect the bugs at states far from the initial states. To deal with the problem, the proposed method concatenates multiple results of such bounded model checkings so that some states far from the initial states may be reached. The notion of control and data is utilized to concatenate multiple bounded model checking results

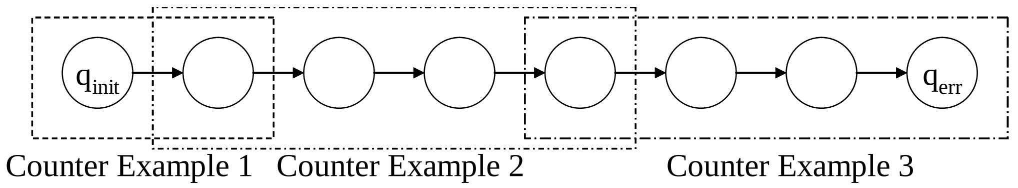


Figure 1.3: Multi-level bounded model checking

efficiently. On model checking, a set of error states derived from a property tend to be at a same or similar control sequences. Then, if we find a counter example for a property, there can be other counter examples on the control sequence of the first counter example. With symbolic simulation techniques introduced in Section 2.4, multiple counter examples on a same control sequence can be gathered. This counter example set corresponds to a set of states in the state space. Therefore, with using a set of states instead of a single state, bounded model checking results can be connected much more efficiently.

Another formal verification proposed in this thesis is an equivalence checking method with translating two designs into models which have an identical datapath as shown in Figure 1.4. Since computations are executed by computation units in hardwares, when the set of computation units and their connections (bus structures) are identical in two designs, the results of same computations must be exactly equivalent in bit-level. In addition, on such a case, if two designs are equivalent, their control flows must be equivalent or at least similar. These features make equivalence checking with symbolic techniques much easier. Therefore, such a translation is applied as a pre-process of symbolic equivalence checking methods. A symbolic equivalence checking method with rule-based approach is also proposed, and its result is compared with that of symbolic simulation method.

1.4 Formal Verification of Hardware/Software Co-Design

Additional difficulties to apply formal verification exist when target designs are hardware/software co-designs. One is the increase of total sizes of designs since

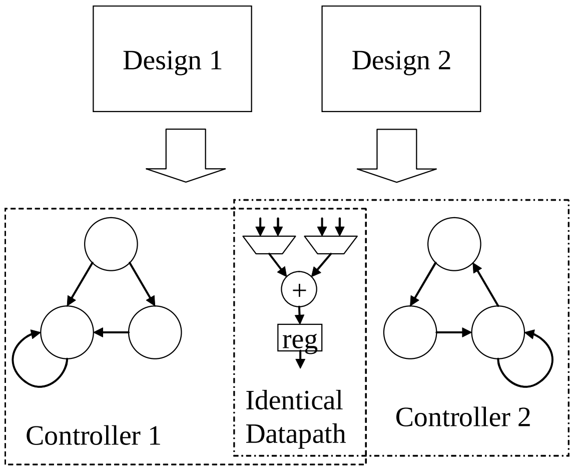


Figure 1.4: Translation into models having identical datapaths

both hardware and software portions are handled at once. In addition, since hardware and software portions interact with each other through memory mapped I/Os and interruptions, the interface parts between hardware and software must be handled. Such interface ports include processor memory addresses, processor buses, bus controllers, and bus interface modules. The other is the difference of abstract level and design languages between hardware and software portions. For example, in a typical case, a software portion is written in program code which corresponds to behavioral level in hardware designs, and a hardware part is written in RTL with a hardware description language, such as Verilog-HDL or VHDL.

Since the above two difficulties prevent practical applications of formal verification to hardware/software co-designs, current hardware and software are verified separately with formal verification in most cases. To apply the two formal verification methods in this thesis to hardware/software co-designs, a preprocess method to convert hardware/software co-designs into simpler forms is proposed in Chapter 5.

1.5 Efficient Implementation of Formal Verification Methods for High-Level Designs

One critical problem to implement formal verification methods for high-level designs is the diverseness of design descriptions in multiple abstraction levels. As mentioned in Section 1.1, many C-based languages, such as SystemC[160], SpecC[54], SystemVerilog[82], BachC[177], BDL[172], and CataplutC[28] exist in behavioral level and system level. In RTL, Verilog-HDL and VHDL are the main design languages. Since there are such many languages, it is redundant to implement a tool for each language.

A simple solution for the problem is to specify an intermediate representation, and implement tools for that representation. Then, only a translator must be implemented to handle designs in a new language. Actually net-list based representations, such as And-Inverter Graph (AIG)[120] are used for low level designs. Formal verification methods can be implemented on such net-list based representations efficiently since the handled design sizes can be reduced by extracting only the portions relating to the problem. For example, on model checking, if the design portions only relating to a verified property are extracted, the verification complexity can be reduced. Such an extraction can be done by tracing wires in the net-list from the signals in the property.

Meanwhile, for high-level designs, such a standard representation does not yet exist. Therefore, an unified representation for high-level designs based on System Dependence Graph (SDG)[75] is proposed in Chapter 6.

1.6 Overall Flow of the Proposed Methods

Figure 1.5 is a verification flow using the proposed methods in this thesis, and shows the position of each method proposed in each section.

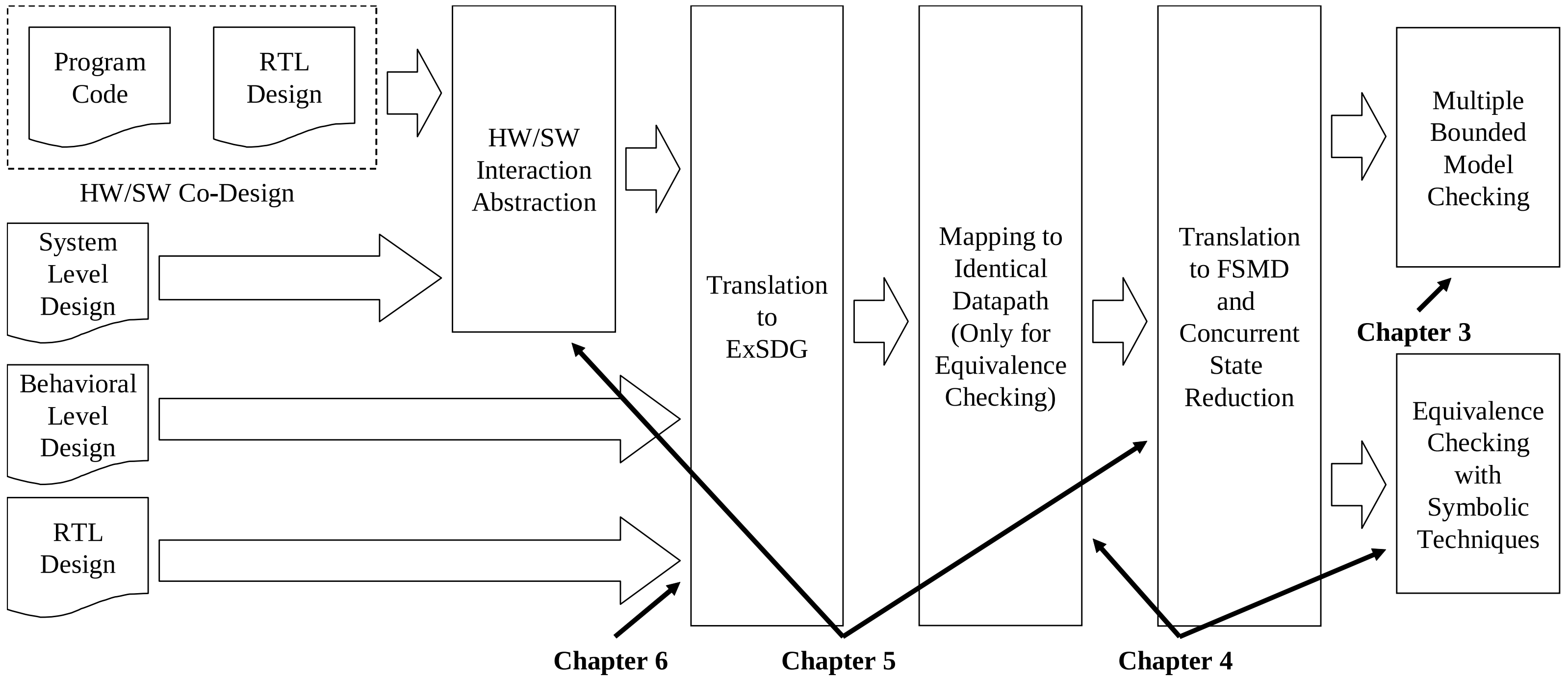

Program Code
RTL Design
HW/SW Co-Design
System Level Design
Behavioral Level Design
RTL Design
HW/SW Interaction Abstraction
Translation to ExSDG
Mapping to Identical Datapath (Only for Equivalence Checking)
Translation to FSMD and Concurrent State Reduction
Multiple Bounded Model Checking
Equivalence Checking with Symbolic Techniques
Chapter 3
Chapter 6
Chapter 5
Chapter 4

Figure 1.5: Overall formal verification flow using proposed methods in this thesis

The inputs can be classified into four types. The first and the second ones are hardware/software co-designs. The first one is written in RTL description and program code, and the second one is written in system-level description. The third and the last ones are hardware designs, and written in behavioral level and RTL, respectively. Firstly, interactions between hardware and software in hardware/software co-designs are abstracted by the method proposed in Chapter 5. Secondly, all designs can be translated into an intermediate representation ExSDG proposed in Chapter 6. This step is only for efficient implementations and the translation itself is not necessary. In the case of equivalence checking, given two designs are translated into models which have an identical datapath. This preprocess is proposed in Chapter 4. Then, all designs are translated into FSMDs and state reduction techniques are applied to concurrent designs. This step is explained in Chapter 5. Finally, multi-level bounded model checking proposed in Chapter 3 or a symbolic equivalence checking techniques proposed in Chapter 4 is applied.

1.7 Organization

The organization of this thesis is as follows.

In Chapter 2, some preliminaries and basic notions for the methods proposed in this thesis are introduced. In Chapter 3, an efficient model checking method by concatenating multiple bounded model checkings with focusing on the control of a design is proposed. It has briefly been introduced in Section 1.3. In Chapter 4, an equivalence checking methods with separating verifications of control and data portions of designs is proposed. It has also briefly been introduced in Section 1.3. To extend the target of the method proposed in Chapter 3 and Chapter 4, a preprocessing technique to apply such formal verification methods to hardware/software co-designs is proposed in Chapter 5. For efficient implementations of such formal verification methods, an intermediate representation for high-level designs is introduced in Chapter 6. Finally, this thesis is concluded in Chapter 7 with some possible future works.

Chapter 2

Preliminaries and Basic Notions

In this chapter, some preliminaries and basic notions for the methods proposed in this thesis are introduced.

Firstly in Section 2.1, existing formal verification methods including model checking and equivalence checking methods are introduced.

Next, what are control and data portions in high-level designs are defined in Section 2.2 since the notion of control and data is a key element in this thesis.

Some representations in which control and data portions are written separately, such as Finite State Machine with Datapath (FSMD)[53] and System Dependence Graph(SDG)[75] are introduced in Section 2.3. These representations are used as a basis or an intermediate representation in the proposed methods.

Finally, symbolic simulation is introduced in Section 2.4, and this technique is applied in both the verification methods proposed in Chapter 3 and Chapter 4.

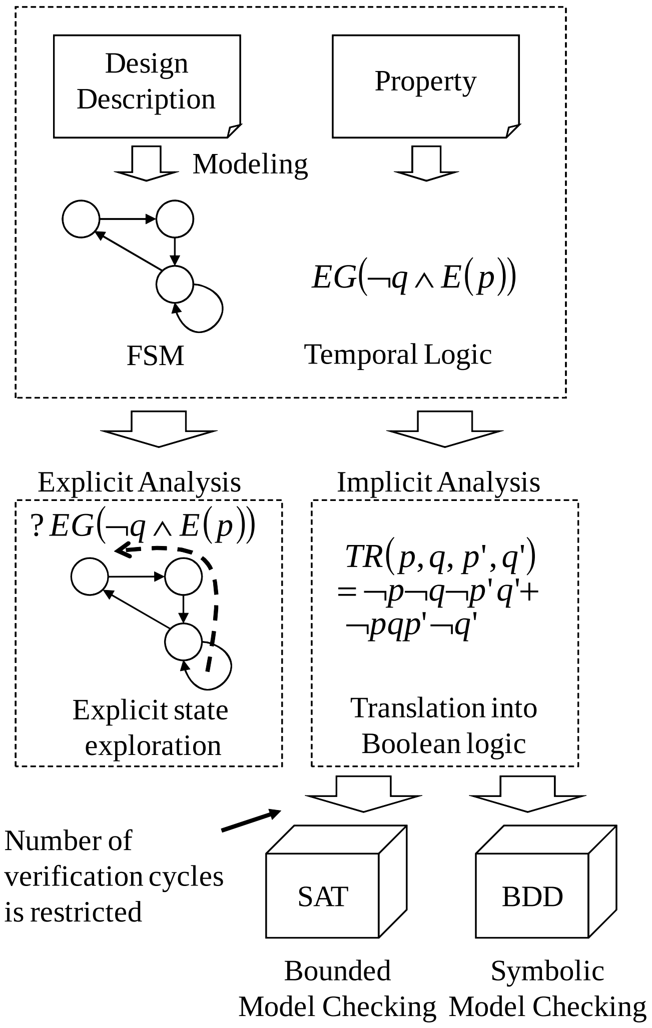
2.1 Formal Verification Techniques

Formal verification methods are mathematical methods that can verify designs exhaustively without considering their input sequences. Formal verification methods can be classified into model checking and equivalence checking methods. In this section, existing model checking and and equivalence checking methods are introduced in Section 2.1.1 and Section 2.1.2, respectively.

2.1.1 Model Checking

Model checking[34] is a method which checks that a design satisfies a specification given as a property, and it is also called "property checking".

A typical model checking flow is shown in Figure 2.1. Inputs are a design represented in Finite State Machine (FSM), and a property represented in temporal logic[45]. Temporal logics are formulae which can represent conditions over multiple states and time steps. There are many temporal logic representations. The basic ones are Computation Tree Logic (CTL)[34], Linear Temporal Logic (LTL)[127], CTL*[88] which is a super-set of CTL and LTL. In practical cases in the industry, Property Specification Language (PSL)[83] and System Verilog Assertion (SVA)[82] are widely used.



Design Description
Property
Modeling
$EG(\neg q \wedge E(p))$
FSM
Temporal Logic
Explicit Analysis
Implicit Analysis
$?EG(\neg q \wedge E(p))$
$TR(p,q,p',q') = \neg p \neg q \neg p' q' + \neg p q p' \neg q'$
Explicit state exploration
Translation into Boolean logic
Number of verification cycles is restricted
SAT
BDD
Bounded Model Checking
Symbolic Model Checking

Figure 2.1: Basic model checking flow

Then, model checking decides whether the FSM satisfies the property. When it is proved that the property is not violated in all of the reachable states from the initial states, the design is guaranteed to be correct for the property. Otherwise, a sequence of state transitions from an initial state which violates the property is generated as a counter example. Basically, the worst case complexity of model checking increases exponentially with a number of state variables.

Model checking methods can be classified into the following two types.

- Explicit method

- Implicit method

The former is the fundamental method, and evaluate the property formula by explicitly tracing each state transition in the FSM[34]. Since the straight forward approach directly traverses each state, it can handle up to about 20 state variables (2^{20} states). Then, state reduction techniques, such as symmetry reduction[105] and partial order reduction[55, 166] are applied to improve the performance. Symmetry reduction generates only one state for each set of symmetric states, and partial order reduction reduces the number of execution orders to be considered by ignoring execution orders among concurrent processes where no interactions occur to each other. Spin[73] is a major public tool implements such a explicit approach, and suits to verify concurrent designs.

The latter one is called symbolic model checking[117]. State transitions in the FSM are represented in a single logical formula, and its conjunction with the formula of the property is evaluated. Though the number of states to be traversed can increase exponentially with the length of sequence of state transitions from the initial states in the explicit methods, the state transitions are represented in a single logical formula in symbolic model checking. Therefore, a same FSM can be represented with less information in symbolic model checking than that of the explicit methods. Evaluation of the formula is performed by Binary Decision Diagram (BDD). BDD is an acyclic graph which has a single departure node as shown in Figure 2.2. Two square nodes are termination nodes and the other nodes are decision nodes. Decision nodes in the same row correspond to a single Boolean variable. Each decision node has two sub nodes and connected with a 0-edge and a 1-edge, respectively. They represent the values of the corresponding Boolean variable. In Figure 2.2, 1-edges are represented by solid arrows, and 0-edges are represented by dashed arrows. The BDD in Figure 2.2 represents NAND logic. Since typical logical formulae can be compactly stored in BDD, relatively large designs which have about 100 state variables can be verified. SMV[117] is a major public tool which implements symbolic model checking.

Bounded model checking[22] is an extended version of the above model checking methods which restricts the maximum length of sequences of state transitions from the initial states. The maximum length is called “bound”, and a bound is given

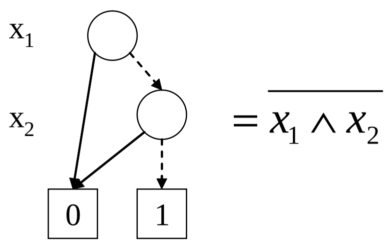


Figure 2.2: Binary Decision Diagram

as a number of cycles. Since the worst case complexities of model checking methods increase exponentially with bounds[31], bounded model checking can control its complexity. If a bound is set as a small number, large designs can be handled. On the other hand, bounded model checking cannot verify with large bounds. Although bounded model checking can be applied to both the explicit and implicit methods, typically it is applied to the implicit methods. In that case, the formula is converted to a satisfiability (SAT) problem. SAT problem is a decision problem to decide whether a set of Boolean variable assignments which makes a given conjunctive normal form (CNF) logical formula true exists. Since the performances of SAT solvers have been advancing dramatically in this century, they contribute the performance of bounded model checking. NuSMV[29] is a major public tool which implements bounded model checking.

To accelerate such model checking methods, several abstraction methods, such as predicate abstraction[58, 16, 106] which abstracts expressions in a description with focusing on some predicates, lazy abstraction[69] which applies predicate abstraction locally, and datapath abstraction[7, 5] which replaces computation units in a datapath with uninterpreted functions, are proposed. With those abstraction methods, counter example guided abstraction refinement (CEGAR) approach[30] which refines the abstraction model when a counter example found in the model is actually not a counter example in the original design. Though the worst case complexity cannot be reduced with such abstraction methods, typical designs can be verified faster.

Many applications of such model checking methods to the design flow shown in Figure 1.1 have been reported. Here, some of them are shown as follows. [144, 123, 59, 60] target on system level and behavioral level designs. [144, 123] focus on properties about synchronization, such as deadlock and race condition, and target on SpecC and SystemC descriptions, respectively. [59, 60] target on SystemC descriptions and verify general properties written in temporal logics. [176, 109] target on hardware/software co-designs composed of RTL codes and program codes, and abstract interactions between hardware and software portions before translating hardware and software portions into a same representation.

Model checking of program code is one of the main research topics in the model checking field, and a number of researches have been reported. A model checker Spin[73] introduced above is mainly targeting on concurrent software programs, and various translators to its input language Promela, such as Feaver[74] which is an ANSI-C front-end, Bandera[39] which is a Java front-end, exist. VeriSoft[56] is the first model checker for ANSI-C program which handles program code directly. Predicate abstraction based model checker SLAM[16, 15, 17] is also practically used to model check software drivers for Windows written in ANSI-C. Lazy abstraction is implemented in BLAST tool[21]. Java PathFinder[171, 108] is a model checker for Java programs, which is originally a translator from Java into Promela which is the input language of Spin, and currently symbolic techniques[95, 130] and heuristic search algorithms[61, 62] are implemented on it. SAT based bounded model checking method for ANSI-C program is proposed in [102] and implemented in a tool CBMC.

Basic model checking methods introduced above[34, 117, 22] are originally proposed for hardware in RTL or lower abstraction levels, and a large number of extended methods have been proposed that cannot be introduced here. Although those methods target on FSM verification, there are some methods which directly target on hardware design[7, 5, 6, 85, 86]. [7, 5, 6] propose a datapath abstraction method for hardware designs and apply CEGAR method with it. [85, 86] propose a method to apply predicate abstraction and also CEGAR to RTL Verilog-HDL codes.

In this thesis, a new extension of bounded model checking, multi-level bounded model checking method, is proposed in Chapter 3.

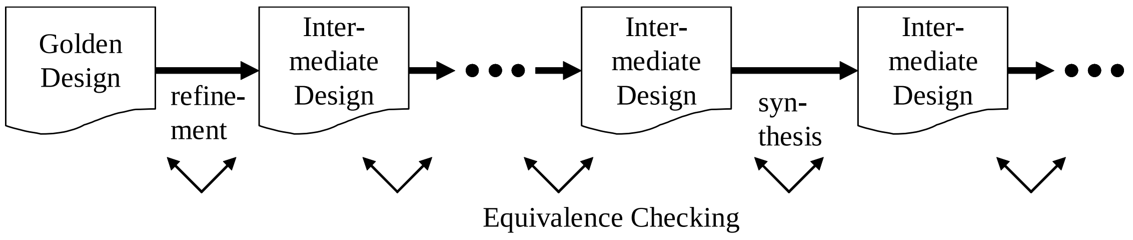


Figure 2.3: Design refinement and synthesis flow

2.1.2 Equivalence Checking

Equivalence checking is the method which decides the equivalence of two given designs, and mainly researched in the hardware verification field.

Details of the flow in Figure 1.1 is shown in Figure 2.3. In each design stage, designs are refined from an initial design (called golden design) step by step with inserting more precise descriptions, modifications and optimizations. A synthesis process, such as high-level synthesis, logic synthesis, or place and route, may also be applied at each stage. As mentioned in Section 1.2, it is important to find and fix design bugs exhaustively in early design stages. Formal equivalence checking is a strong technique for that purpose. If two designs between a single refinement step or synthesis step are compared and proved to be equivalent, then it is guaranteed that no additional bugs were inserted during the refinement or synthesis step. Therefore, if the golden design can be proved not to have any bugs by simulation or model checking, then further bug insertions can be avoided with equivalence checking. Because of such backgrounds, equivalence checking methods have been actively researched in hardware verification field from lower abstraction levels.

Combinational Equivalence Checking

Equivalence checking method for combinational circuits is called combinational equivalence checking, and based on miter-circuit[25] shown in Figure 2.4. In a miter-circuit, each corresponding pair of inputs and outputs of two combinational circuits to be compared are connected, and an XNOR gate is inserted for each output pair. If all the XNOR gate values are always true for all input patterns, the two

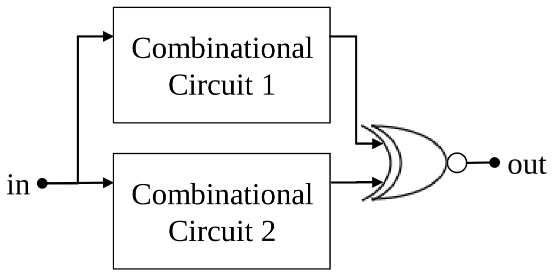


Figure 2.4: Miter Circuit

circuits are proved to be equivalent. This evaluation is performed by BDD based approach[20, 27, 135, 116], SAT based approach[57, 42, 9], or their combination[134]. With cut-point(equivalent nets in a circuit) techniques[116, 93, 103], currently circuits contain millions of gates can be verified, and practically used in the industry.

Sequential Equivalence Checking and Equivalence Checking of RTL Designs

Although combinational equivalence checking methods can efficiently verify large circuits, most of practical circuits are sequential ones which include flip-flops. Since combinational equivalence checking cannot be applied directly to such circuits, a simple method compares sequential circuits as combinational ones. Firstly, a miter circuit of sequential circuits is created as shown in Figure 2.5 just same as that of combinational circuits. The difference from the combinational one is the existence of flip-flops. Secondly, the sequential miter circuit is converted into a combinational one as shown in Figure 2.6. The inputs of the flip-flops are treated as primary outputs, and the outputs of the flip-flops are treated as primary inputs. Here, this miter circuit can be verified with combinational equivalence checking techniques which have already been introduced in the previous section. However, to apply above conversion, the correspondences of flip-flops between two circuits must be known by applying register correspondence algorithms, such as introduced in [103]. Moreover, the flip-flops must be corresponding one by one between the two designs. This is a strong restriction in sequential equivalence checking.

A more general approach which can be applied to any sequential circuits is state

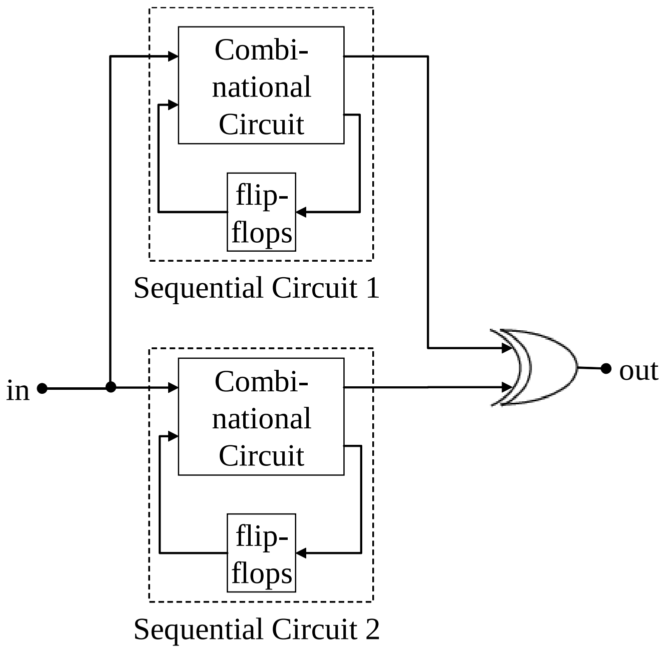


Figure 2.5: Miter circuit of sequential designs

traversal based approach. With regarding flip-flops as state variables, sequential miter circuit as shown in Figure 2.5 can be translated into finite state machine (FSM). Then model checking techniques[34, 117, 22] are used to check the property that the miter outputs never be 1. If the model checking formally proves this property, then the two sequential circuits in the miter circuit are proved to be equivalent. This method can be applied to any sequential circuits, but its computation amount is large since all states in the FSM are considered. Usually, the number of states grows exponentially with the number of flip-flops in a circuit (when all states are reachable from the initial states).

The above two approaches have defects in generality and complexity, respectively. Therefore, there are mainly two approaches to improve the general performance under a restriction of generality.

One is a conservative approach which allows false-negatives (there can be some cases that even given two circuits are equivalent, their equivalence is not proved), but it can prove equivalence quickly. Efficient combinational equivalence checking

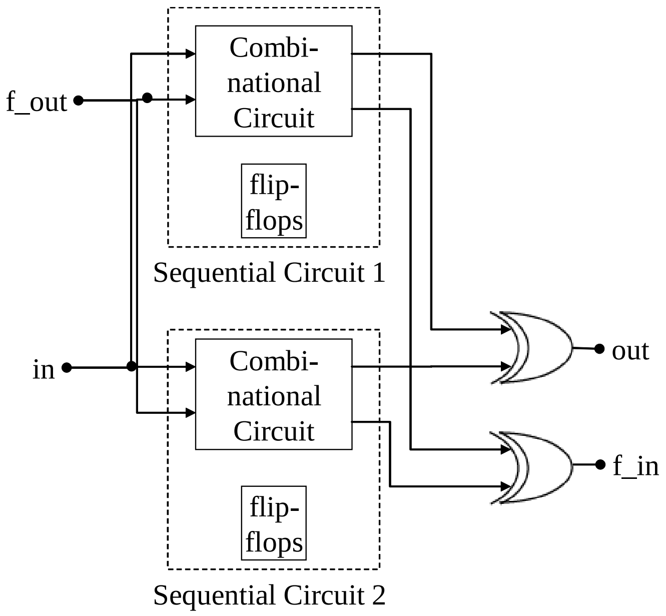


Figure 2.6: Conversion of sequential miter circuit into combinational

based methods are proposed in [133, 92]. Reachability analysis methods specific for FSMs translated from sequential circuits has been proposed in [111, 64]. To avoid full state space traversal in reachability analysis, induction based methods have been proposed in [167, 168, 23, 79, 87, 157, 158]. Divide and conquer approach can also be used for equivalence checking to reduce the complexity[94, 122, 121]. Sequential equivalence checking methods using sequential ATPG techniques were proposed in [78, 76, 77]. In those methods, ATPG generates an input sequence which can detect stuck-at-0 of a miter output. If it cannot generate such an input sequence, then two circuits in the miter circuit is proved to be equivalent.

The other approach keeps completion and does not allow false-negatives. However, it restricts applied sequential transformations between compared two designs. [10, 110] focus on distinguishability of states by an input sequence of a finite length, and they can completely verify retiming and resynthesis transformations.

Though most commercial tools, such as Formality[47], are based on combinational equivalence checking methods, it can handle typical sequential circuits (without feedback loops) with moving positions of flip-flops or removing them.

In RTL and higher abstraction levels, some word-level methods are used. Word-level methods verify designs without extracting multi-bit variables (flip-flops and nets) into Boolean logics, and just treat them as words, then it can reduce the complexities dramatically from the bit-level methods which have already been introduced. [137] applies symbolic simulation which is a word-level method introduced in Section 2.4, and verifies RTL hardware designs. A commercial equivalence checker for sequential circuit, SLEC RTL[155] also uses this technique.

Equivalence checking between Designs before and after High-Level Synthesis

Equivalence checking between designs before and after high-level synthesis is also a major topic. [150] translates behavioral level designs into RTL and applies sequential equivalence checking, but the performance depends on the back-end sequential equivalence checking methods. [33] converts both two designs into Boolean logic formulae, and applies bounded model checking method to check that the two Boolean logic formulae are equivalent. Since this method is a bit-level method, large designs cannot be handled.

[11, 112, 90, 91] use symbolic simulation techniques to compare two designs in word-level. [11, 90, 91] also use divide and conquer approach to handle larger designs. However, since [11] uses information from synthesis tools, they can only been applied to automatic high-level synthesis results. In [90, 91], equivalences of paths between conditional branches are checked, and the results are gathered to prove the entire equivalence. To apply this method, correspondences of input/output signals and flip-flops must be known. Therefore, they do not applicable for designs before and after complete high-level synthesis (Actually, [90, 91] target on designs before and after scheduling). SLEC SYSTEM[156] is a commercial equivalence checking tool which uses symbolic simulation technique.

[50, 48, 49] propose another approach to compare designs before and after high-level synthesis that maps behavioral designs into a virtual controller and a virtual datapath, and compares the controllers and the datapaths separately. The proposed method in Chapter 4 is based on this approach.

Equivalence Checking of System Level Designs and Behavioral Level Designs

Since system level designs are written in behavioral level, the same framework is used for equivalence checking of both behavioral level and system level designs.

[143] proposes a method which compares two C-based designs with symbolic simulation. However, since symbolic simulation can handle only a single control sequence at once, the computation amount is doubled per each conditional branch. Descriptions which contain loops also cannot be handled without unrolling them for fixed numbers of iterations. Then, large designs are difficult to be directly handled. [113] proposes a method only handle the different portions between the two designs to solve that problem. When the differences between two designs are small, even large designs can be handled. The problem that loops must be unrolled as a preprocess is solved by [115, 152]. In these methods, array indices are symbolically represented and symbolic simulation is iteratively applied to the inside of the loops until the equivalence is inductively proved.

[145] proposes a method to handle C-based descriptions include concurrencies. Model checking with synchronization property, such as no deadlock or race condition exists, is applied as a pre-process, and it guarantees that the concurrent description can be equivalently translated into a sequential description.

[52, 151] proposes a rule based bottom-up approach to check the equivalence. If the differences between two designs are in the ruled cases, the equivalence can be proved quickly. [1] also proposes a graph based approach and graphs generated from designs are converted to a canonical form with a pre-defined rules.

2.2 Control and Data in High-Level Design

In this section, the details and definitions of control and data in high-level designs are explained.

As briefly explained in Section 1.3, control is the flow of the behavior of a design, and data is the set of actual computations executed at the control steps. Firstly, control and data in each design stage are defined as follows.

```
int main(int a){
  int b;
  if(a == 2) //(1)
    b = 1; //(2)
  else
    b = 2; //(3)
  b = b + a; //(4)
  return b; (5)
}
```

Figure 2.7: Example C code to show the design element classification

Control and data of system-level design, behavioral hardware design, and software program code

System-level design, behavioral hardware design, and software program code are described in programming languages or their extensions. Since such languages are typically functional languages, only functional languages are focused on here. In functional languages, functional design components can be briefly classified into functions, statements, and expressions. A function is a unit of a behavior flow in a design, and composed of a sequence of statements. A statement is a control point in a design, such as conditional branch, loop entry, expression execution like assignment. In a statement, executed expressions, next statements to transit, and transition conditions are defined. An expression is a actual computation, such as addition, multiplication, and comparison. In the example ANCI-C code shown in Figure 2.7, **main** is a function. Each line with a commented number shows a statement, such as an **if** statement, assignment statements, and a return statement. Each element under the statements is an expression, such as an equal expression, assignment expressions, an addition expression, variable expressions, and constant expressions.

In such a design written in a functional programming language, its control is defined as follows.

Definition 1 (Control of Functional Programming Language)**.** The control portion of a design written in a functional programming language is a tuple $CONTROL =$

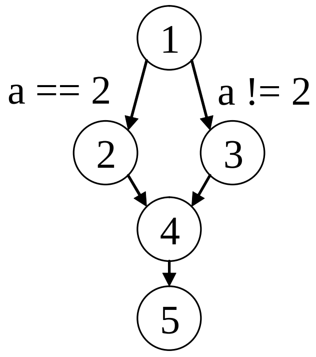


Figure 2.8: Control of the example in Figure 2.7

(S, T), where S is a set of statements. T is a transition function defined as

$$T : S \times E \rightarrow S$$

where E is a set of expressions. T returns the next statement when a current statement and a conditional expression which suppose to be true are given.

For example, the control of the code in Figure 2.7 can be represented by a directed graph shown in Figure 2.8. A number of each node represents a statement ID which is a commented number in Figure 2.7, and edges represent statement transitions with conditional expressions. Edges without conditional expressions show that their transition conditions are $true$.

The data of a design written in a functional programming language is also defined as follows.

Definition 2 (Data of Functional Programming Language)**.** The data portion of a design written in a functional programming language is a tuple $DATA = (E, R)$, where E is a set of expressions and R is a function that returns an expression executed in a given statement defined as follows.

$$R : S \rightarrow E$$

For example, the data of the code in Figure 2.7 can be represented by a table shown in Table 2.1.

From the above two definitions, a design written in a functional programming language can be defined as a tuple $(CONTROL, DATA)$.

Table 2.1: Data of the example in Figure 2.7

Statement ID	Executed Expression
1	–
2	$b = 1$
3	$b = 2$
4	$b = b + a$
5	$return\ b$

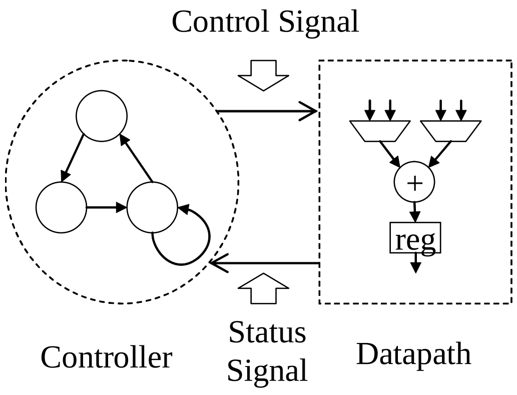


Figure 2.9: Controller and datapath

Control and data of RTL hardware design

It is impossible to generally define the control and data of RTL hardware designs without any assumptions, since a hardware design is just a circuit and it can be designed without the notion of control and data. For example, combinational circuits do not have a notion of control. As mentioned in the sequential equivalence checking part in Section 2.1.2, most of practical designs are sequential circuits. To design sequential circuits, designers usually consider flows of computations which directly relate to the notion of control and data. Then, here hardware designs are assumed to be sequential circuits which are composed of controllers and datapaths as shown in Figure 2.9.

A controller is a control FSM which represents a set of control points in a design and transitions among them. A controller has state registers (flip-flops) and their values correspond to the states. A datapath is composed of computation units and

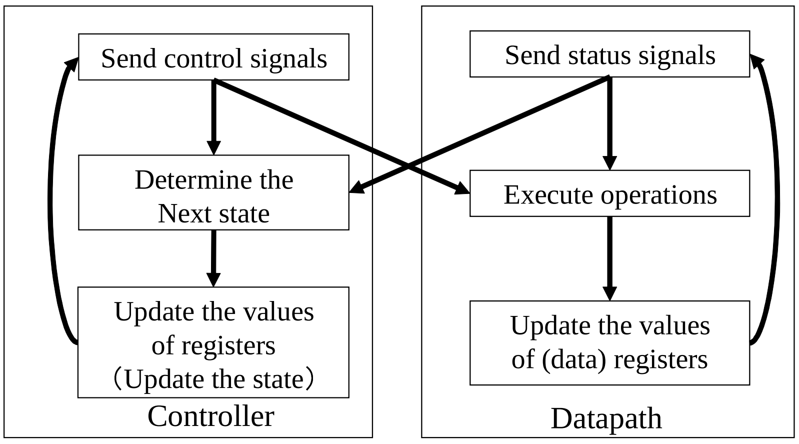


Figure 2.10: Behavioral flow of controller and datapath

their connections, and represents computations actually performed at those control points. A controller and a datapath are executed parallely as the flow shown in Figure 2.10. At each clock cycle, first, the controller sends control signals to the datapath depending on the current state. Next, the datapath executes computations based on the control signals. Finally, the datapath returns status signals to the controller, and the controller determines the next state.

The structure of a controller and a datapath is directly represented by Finite State Machine with Datapath (FSMD)[53] introduced in Section 2.3. The precise definitions of controller and datapath can be found in the definition of FSMD.

Control/Data Separation

As introduced in Section 2.1, most of the formal verification methods target on FSMs which is Kripke structures or Deterministic FSMs (DFSMs) defined as follows.

Definition 3 (Finite State Machine (FSM))**.** An FSM is a Kripke Structure which is a tuple

$$M_{FSM} = (S, S_0, R, L)$$

where S is a set of states, $S_0 \subseteq S$ is the set of initial states. R is a transition relation defined as follows.

$$R \subseteq S \times S$$

It represents the existence of the transition between each pair of states. L is a labeling function defined as

$$L : S \to 2^{AP}$$

where, AP is a set of atomic propositions.

Definition 4 (Deterministic Finite State Machine (DFSM))**.** A DFSM is a tuple

$$M_{DFSM} = (S, \alpha, I_V, T, L)$$

where S is a set of states, α is the initial state, and I_V is a set of input values. T is a transition function defined as follows.

$$T : S \times I_V \to S$$

T returns the next state from a current state and a current input value. L is a labeling function defined as

$$L : S \to 2^{AP}$$

where, AP is a set of atomic propositions.

Since next states are deterministic in DFSM, only deterministic designs can be represented. Therefore, concurrent models cannot be described with DFSM.

Since states are not weighted nor biased in the definition of FSM nor DFSM, all states in them are treated equally in the most of formal verification methods. However, in practical designs, usually there is a bias among the states, since those designs have the notion of control and data in most cases. In a design written in functional programming language, a state corresponds to a pair (s, Val), where $s \in S$ is a current control point (executed statement) and Val is a set of the current variable values. Also in an RTL hardware design, a state corresponds to a list of the current values of control and data registers. The list of control register values corresponds to a control point. In those states, states at a same control point can be considered to be in a same group. Then, sequences of state transitions of a same control path correspond to a same sequence of state groups. Since corresponding control paths are same, those sequences can have some same features. For example, they may tend to violate a same property. In this thesis, that feature is utilized by treating multiple sequences belong to a same group at once with symbolic simulation introduced in Section 2.4. Since symbolic simulation treats variables and operators as symbols, it can handle multiple sequences corresponds to a same control path at once. Therefore, the control and data of a design can be analyzed separately. In the multi-level bounded model checking method proposed in Chapter 3, symbolic simulation is applied to connect multiple counter examples efficiently. With grouping counter examples with the symbolic method, the connection points of those counter examples are expanded to a set of states instead of just a single state. This makes the verification much more efficient.

In addition, the notion of control/data separation can also help equivalence checking since if the data portions of two designs are identical, the comparison of control portions becomes much easier. Symbolic methods can be applied without any restrictions since computations performed at corresponding control points are identical. Control portions also become equivalent or at least more similar in equivalent designs. In the method proposed in Chapter 4, the data portions of two designs are forcibly made identical, and symbolic techniques including symbolic simulation are effectively applied.

To apply such symbolic techniques, representations whose control and data can be separated are required. Therefore, Finite State Machine with Datapath (FSMD)[53] is used as the common intermediate representation in this thesis. The details of FSMD is introduced in Section 2.3.1.

The notion of control and data dependencies are also utilized in the methods proposed in Chapters 5 and 6 to merge FSMD states without any dependencies each other and make verification tool implementation easier, respectively. Control dependence is a directed relation between two control points that one determines the execution of the other, such as between an **if** statement and a statement under that **if** statement. Data dependence is also a directed relation between two expressions that the data of one expression affect the other, such as between the left hand side and the right hand side of a same assignment, and between the left hand side variable of an assignment and the next expression which uses the value of the variable. These dependencies are expressed as edges in System Dependence Graph introduced in Section 2.3.3.

2.3 Control/Data Separated Design Representations

In this section, two representations, Finite State Machine with Datapath (FSMD) and System Dependence Graph (SDG), are introduced as control/data separated representations. These representations are used as intermediate representations in the proposed methods in this thesis.

2.3.1 Finite State Machine with Datapath

General FSMD FSMD[53] is a specification description for sequential RTL designs. Actually, it is standardized by Accellera as a specification description for RTL design[63]. In an FSMD, control and data portions of a design can be specified separately. The basic definition of FSMD is as follows.

Definition 5 (Finite State Machine with Datapath)**.** FSMD is a tuple

$$M_{FSMD} = (S, \alpha, I_V, V_V, \beta, O_V, fs, fv, fo)$$

where S is a set of control states, $\alpha \in S$ is the initial state, I_V is a set of input values, V_V is a set of data register values, $\beta \in V_V$ is an initial data register value, and O_V is a set of output values. fs is a transition function which returns the next state from a current state, a current data register value, and an input value defined as follows.

$$fs : S \times I_V \times V_V \to S$$

fv is a next register value function which returns the next register value from a current state, a current register value, and a current input value defined as follows.

$$fv : S \times I_V \times V_V \to V_V$$

fo is an output function which returns the current output value from a current state, a current register value, and a current input value defined as follows.

$$fo : S \times I_V \times V_V \to O_V$$

In Definition 5, it can be considered that fs represents a controller FSM, and fv and fo represents a datapath in a typical RTL design shown in Figure 2.9 since fs defines the structure of control state transitions, and fv and fo define the computations which determine the values of data registers and outputs, respectively.

It is clear that FSMD is equivalent to DFSM. Compared to DFSM, control states and data states (values of data registers) are separated in FSMD. Particularly, $S \times V_V$ in Definition 5 corresponds to S in Definition 4. Then, if S is considered as a set of control points, then the control/data separation methods which have been introduced in Section 2.2 can be applied.

Also, since FSMD corresponds to DFSM and three functions fs, fv, and fo are deterministic, it cannot represent non-deterministic designs, such as designs having concurrency. However, it can represent any designs representable in DFSM, including software programs, behavioral hardware designs, and RTL designs.

FSMD with Symbols and Expressions Though Definition 5 is simple and easy to understand, it does not directly correspond to actual design descriptions having symbols and expressions. Actually, there are many other definitions of FSMD[53, 63, 90, 147, 91], having the notion of symbols and expressions. However, since none of them directly corresponds to the general FSMD definition (Definition 5) nor the DFSM definition (Definition 4), it is difficult to know their representabilities. Therefore, FSMD is re-defined in an original style that directly corresponds to Definition 5 in this thesis by extending the general FSMD definition.

Firstly, the notion of symbols is introduced as follows. I_V, V_V, and O_V in Definition 5 can be decomposed as

$$\begin{aligned} I_V &= I_{V1} \times I_{V2} \times \cdots \times I_{Vl} \\ V_V &= V_{V1} \times V_{V2} \times \cdots \times V_{Vm} \\ O_V &= O_{V1} \times O_{V2} \times \cdots \times O_{Vn} \end{aligned}$$

where, I_{Vj}, V_{Vj}, and O_{Vj} represent sets of values of the jth input, data register, and output respectively. Let I, V, O denotes symbol sets of inputs, data registers, and outputs, respectively. Then, a function σ is defined as follows.

$$\sigma : I \cup V \cup O \rightarrow \{I_{V1}, I_{V2}, \cdots I_{Vl}, V_{V1}, V_{V2}, \cdots V_{Vm}, O_{V1}, O_{V2}, \cdots O_{Vn}\}$$

It returns the set of possible values taken by a given input, data register, or output symbol. For example, $\sigma(i_j) = I_{Vi}$, where $i_j \in I$ is the symbol of the jth input.

Secondly, the notion of assignments is introduced as follows. Let $A = A_V \cup A_O$ denote a set of assignments where A_V and A_O are sets of assignments to data registers and outputs, respectively. Each assignment is a function defined as follows.

$$a_v : I_V \times V_V \to \sigma(v) \mid a_v \in A_V, v \in V$$
$$a_o : I_V \times V_V \to \sigma(o) \mid a_o \in A_O, o \in O$$

An assignment returns the next value of a data register or the current value of an output, and the value is defined with the current input values and data register values. Let P denote a relation that defines a assignment set belongs to each control state defined as follows.

$$P \subseteq S \times A$$

In each state, for every data register and for every output, there must be only one assignment to v. Therefore, the following condition must be true.

$$\forall s \in S, \forall x \in V \cup O, \forall i \in I_V, \forall v \in V_V, \exists 1 a \in A, ((s, a) \in P \wedge a(i, v) \in \sigma(x)) \quad (2.1)$$

Since a set of assignments determines the next data register values and the current output values from a current input values and data register values. It must be equivalent to fv and fo in Definition 5 when representing an identical design. Then, a set of assignments $A = A_V \cup A_O$ and fv and fo in Definition 5 must satisfy the following equation.

$$\forall s \in S, \forall i \in I_V, \forall v \in V_V,$$
$$\left(fv(s, i, v) = \prod_{a_v \mid a_v \in A_V, (s, a_v) \in P} a_v(i, v) \right) \wedge \left(fo(s, i, v) = \prod_{a_o \mid a_o \in A_O, (s, a_o) \in P} a_o(i, v) \right)$$

The notion of transition conditions is also introduced as follows. Let G denote a set of conditions. Each condition is defined as follows.

$$g : I_V \times V_V \mid g \in G$$

A condition returns 0(false) or 1(true) for a given input value and a data register value. Let R denote a set of state transitions defined as

$$R = \{(s, fs(i, v)) \mid s \in S, i \in I_V, v \in V_V\} \subseteq S \times S$$

Let Q denote a function returns a condition belongs to a state transition defined as follows.

$$Q : R \rightarrow G$$

Note that an executed transition from a state must be determined with a given condition, which means that transition conditions of all the transitions from a state are exclusive, and the disjunction of all the transition conditions from a state is true. Then the following equation must be true.

$$\forall s \in S, \forall i \in I_V, \forall v \in V_V, \exists 1 s' \in S, ((s, s') \in R \wedge (i, v) \in Q((s, s')))$$

Since the next states are determined by transition conditions, they must be equivalent to fs in Definition 5 when representing an identical design. Then, R, Q and fs in Definition 5 must satisfy the following equation.

$$fs(s, i, v) = s' \leftrightarrow (s, s') \in R \wedge (i, v) \in Q((s, s')) \mid s, s' \in S, i \in I_V, v \in V_V$$

Next, the notion of expression is introduced. An expression is a symbolic label of computations on an assignment or a transition condition. Computations are described with F_{call} which is a set of function calls defined as follows.

$$F_{call} = \{(f, e_1, e_2, ..., e_n) \mid f \in F, e_1, e_2, ..., e_n \in E\}$$

where f is an operator as a function label, $e_1, e_2, ..., e_n$ are arguments, n is the number of arguments. E is a set of expressions defined as follows.

$$E = I \cup V \cup O \cup K \cup F_{call}$$

Each assignment or condition is labeled with an expression, and the following labeling functions are defined.

$$\begin{aligned} L_A &: A \rightarrow (V \cup O) \times E \\ L_G &: G \rightarrow E \end{aligned}$$

Finally with the above definitions, the FSMD used in this thesis is defined as follows.

Definition 6 (FSMD with Symbol and Expression)**.** An FSMD is defined as a tuple $M_{FSMD} = (D, C)$, where D is a datapath with data registers and operators, and C is a controller FSM.

A datapath is defined as a tuple:

$$D = (I, O, V, \beta, \sigma, K, F, F_{call}, A, G, L_A, L_G)$$

where I is a set of input symbols, O is a set of output symbols, V is a set of data register symbols. $\beta \in \prod_{v \in V} \sigma(v)$ is an initial data register value. σ is a function which returns a set of possible values of a symbol, K is a set of constants, F is a set of operators, F_{call} is a set of function calls, $A = A_V \cup A_O$ is a set of assignments, G is a set of conditions, L_A is a labeling function for assignments, and L_G is a labeling function for conditions. All of them have already been defined.

A controller is defined as a tuple:

$$C = (D, S, \alpha, R, P, Q)$$

where D is a datapath defined above, S is a set of control states, $\alpha \in S$ is the initial state, R is a transition relation, P represents a relation between states and assignments executed at the states, and Q is a function which returns the condition that a transition is performed. All of them have also been already defined.

The features of this definition are as follows:

- Design descriptions in RTL, behavioral level, and program code can be simply mapped to FSMDs, since the notion of symbols and expressions are introduced.
- Representable design types of this definition and Definition 5 are equivalent, since no restrictions are assumed in the above definition process.
- Control and data portions of a design can be easily separated, since the controller C and the datapath D directly represent them, respectively.

In this thesis, this definition is referred as the definition of FSMD, and the FSMD is used as an intermediate representation as shown in Figure 1.5.

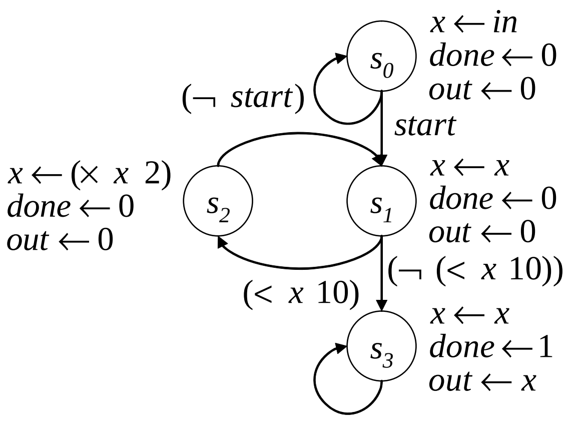


Figure 2.11: An example of FSMD with Definition 6

Example

Figure 2.11 shows an example of FSMD. It repeats doubling an input $in \in I$ while it is smaller than 10. If it becomes equivalent to or greater than 10, then the number is assigned to an output $out \in O$. The FSMD has four states ($S = \{s_0, ..., s_3\}$), and $s_0 = \alpha$ is the initial state. The other symbols in the FSMD are $I = \{in, start\}$, $V = \{x\}$, $O = \{out, done\}$, $K = \{0, 1, 2, 10\}$, and $F = \{\times, \neg, <\}$. A left arrow ($\leftarrow$) represents an assignment which is included in $A = A_V \cup A_O$. Particularly, assignments to x are included in A_V and assignments to *done* and *out* are included in A_O. They define next values of x and current values of *done* and *out*, respectively. When a state transition is performed, all assignments of the departure state are executed simultaneously. Function calls are written in a Lisp-like style, which means the first symbol in a parentheses is an operator, and the other symbols are argument expressions.

The expression described on each state transition represents its transition condition, and transition conditions which are 1(*true*) are omitted in this figure.

2.3.2 Definition of Other Notions Related to FSMD

For the FSMD defined in Definition 6, the following notions are also defined for the following chapters.

Sequence of State Transitions Let T denote a set of sequence of state transitions, defined as follows.

$$T = \{t = (s_0, s_1, \cdots s_n) | t \in S^{n+1}, \forall j, (0 \leq j \leq n-1) \rightarrow (s_j, s_{j+1}) \in R, 1 \leq n\}$$

where n is the length of the sequence of state transitions.

Symbolic Value A symbolic value represents a set of all possible values of an expression. A condition or equation with the symbolic values of expressions represents a relationship which must be satisfied among those expressions. For example, let a, b, and c are symbolic values, and an equation $a+b = c$ with them are given. When concrete values (assignments of concrete numbers) of them, such as $a = 1, b = 2$, and $c = 3$, are considered, they must satisfy the equation. Therefore, a condition or equation with symbolic values can be considered as a condition or equation which must be satisfied when those symbolic values are treated as variables.

For the FSMD definition in Definition 6, I, V, O, K, F_{call}, and E can be treated as sets of symbolic values, since they are sets of symbols. These symbolic values are used to analyze FSMD symbolically.

2.3.3 System Dependence Graph

System dependence graph for software program

Dependency between program portions is mainly researched in the software field for the purpose of program slicing. Program slicing[173, 174] is a technique which extracts program portions relating to a given portion of a program. A set of extracted portions is guaranteed to be a complete program code that can be compiled and run.

For graph based program slicing approach, dependence graph[128] of a program is proposed. Dependence graph is a graph where each node represents a statement and each edge represents dependence. Those edges are mainly classified into three types, data dependence edges, control dependence edges, and declaration dependence edges.

Data dependence is a directed relation between nodes which has a direct relation in the data flow, and represented by a data dependence edge. A data dependence

```
1 int i;
2 int a;
3 int b;
4 i = 1;
5 a = 0;
6 b = 0;
7 a = a + i;
8 if(a == 1)
9   b = a++;
```

Figure 2.12: Example program for dependence graph

edge is a directed edge drawn from an assignment node n_1 to another node n_2 if the assigned variable at n_1 can be used at n_2.

Control dependence is a directed relation between nodes which has a direct relation in the control flow, and represented by a control dependence edge. A control dependence edge is a directed edge drawn from a control point node n_1 to another node n_2 if the execution of n_2 is controlled by n_1 (e.g. conditional branch). Dependence graph can represent dependencies in a program without the notion of function.

Declaration dependence is a directed relation between a variable declaration and a portion it is referred, and represented by a declaration dependence edge. A declaration dependence edge is a directed edge drawn from a variable declaration node to another node where the variable is used.

An example program code and its dependence graph is shown in Figure 2.12 and Figure 2.13. Each node represents a program portion. Heavy edges are data dependence edges, dashed edges are control dependence edges, and solid edges are declaration dependence edges. Program slicing can be applied with traversing those dependence edges and extracting reached portions. Such a reachability analysis finishes within linear time with the design size. If the edges are traversed backwardly from the node **b = a++;** corresponds to line 9 in Figure 2.12, the portions which affect line 9 are extracted. This process is called backward slicing and lines except line 6 (**b = 0;**) are extracted in this example.

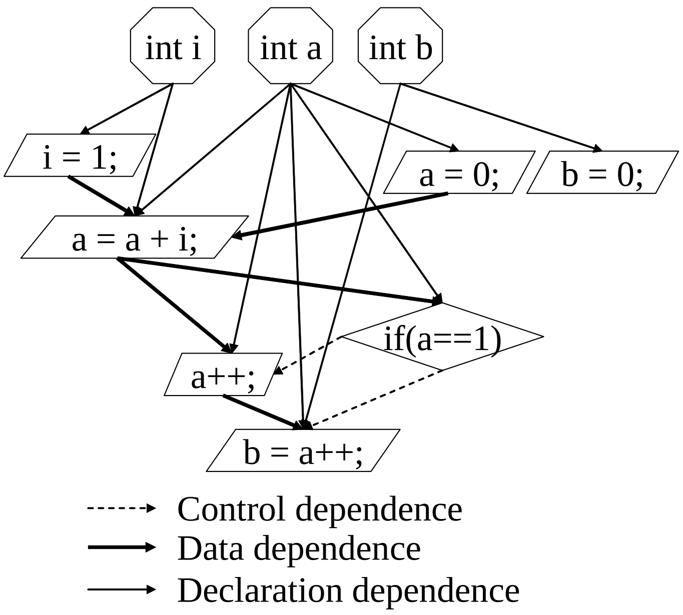


Figure 2.13: Dependence graph of the code in Figure 2.12

To handle programs include multiple procedures, such as functions, System Dependence Graph (SDG)[75] is proposed. SDG is an extension of the dependence graph[128], which contains multiple Procedure Dependence Graphs (PDGs), and expresses dependences among those procedures. Each PDG corresponds to a single procedure. To describe dependences between procedures, additional nodes and edges, such as call-site nodes, formal-in/out nodes, actual-in/out nodes, call edges, parameter-in/out edges, and interprocedural dependence edges (summary edges) are introduced.

A call-cite node represents a procedure call, formal-in and formal-out nodes represent arguments and return values of a procedure, respectively, actual-in and actual out nodes represent a given argument and taken return value of a procedure call, respectively. A call edge is drawn from an expression node which represents a function call expression to the corresponding call-site node, and it represents the control dependence of the procedure call. A parameter in edge is drawn from an actual-in node to the corresponding formal-in node, and it represents the data dependence through the argument. A parameter out edge is drawn from a formal-out node to an actual-out node, and it represents the data dependence through the return value. Only with those edges, because of the lack of the information about correspondences between arguments and return values of a same procedure call, program slicing is inaccurate when one procedure is called from multiple places (False data dependence among those places exist). Interprocedural dependence edge, also called summary edge, is proposed to solve the problem. A summary edge is drawn from an actual-in node to an actual-out node for a same procedure call, and it represents the data dependence from an argument to a return value of the same procedure call. Such SDG and interprocedural program slicing methods are implemented in a commercial program slicing tool Codesurfer[37].

An SDG of the example code in Figure 2.14 is shown in Figure 2.15. In Figure 2.15, pentagon nodes are call-site nodes, and ellipse nodes are formal-in/out and actual-in/out nodes. Since a call edge represents control dependence and parameter-in/out edges represent data dependence, they are shown in the same types of arrows as control dependence edge and data dependence edge, respectively. If the data-dependence and parameter-in/out edges are backwardly traversed from the node **d = sub(b);** which corresponds to line 13 in Figure 2.14, the node **a = 0;** which corresponds to line 10 is reached. However, there is no data dependence between them. This is an example of false data dependence mentioned above.

Then, interprocedural dependence edges represented by heavy dashed edges are inserted, and a two level slicing method is applied to make slicing results more accurate. In the first step, edges except parameter-out edges are traversed. In the second step, edges except parameter-in edges and call edges are traversed. In this approach, since both parameter-in and parameter-out edges are not traversed at once, the problem of false-data dependence can be avoided.

In [101], program slicing method for concurrent program is proposed with introducing three additional types of edges, such as parallel edge, communication dependence edge, and interference dependence edge. Parallel edge represents dependence about starting concurrent executions. A parallel edge is drawn from a node which starts concurrent executions to the first node of a concurrent process. Communication dependence edge represents synchronization. A communication dependence edge is drawn to a node which stops the process until a corresponding

```
int sub(int in){
  return in;
}

main(){
  int a;
  int b;
  int c;
  int d;
  a = 0;
  b = 0;
  c = sub(a);
  d = sub(b);
}
```

Figure 2.14: Example program for SDG

signal is sent as a trigger from a node which sends the signal. Those nodes must be in different concurrent processes. Interference dependence edge represents data dependence between concurrent processes. An interference dependence edge is drawn from an assignment node n_1 to another node n_2 if the assigned variable at n_1 can be used at n_2 and those nodes are in different concurrent processes.

Figure 2.17 is an SDG of the concurrent code in Figure 2.16. Declaration nodes and declaration dependence edges are abbreviated in Figure 2.17. In Figure 2.16, **par** statement in line 7 executes concurrent processes under the statement. In this code, two processes, **proc1** and **proc2** are executed concurrently. In **proc2**, a **wait** statement in line 14 stops the process execution until receiving the argument trigger event signal **e**. In **proc1**, **notify** statement in line 11, sends the argument trigger event signal **e**. Parallel dependence edges are drawn from the **par** node to the concurrent process nodes in Figure 2.17. A communication dependence edge is also drawn from the **notify** node to the **wait** node. Interference dependence edges are drawn from the assignment nodes corresponds to line 10 and line 15 to the assignment node corresponds to line 15 and line 9, respectively, since they are in the different concurrent processes, and they have data dependence of the variable **a**

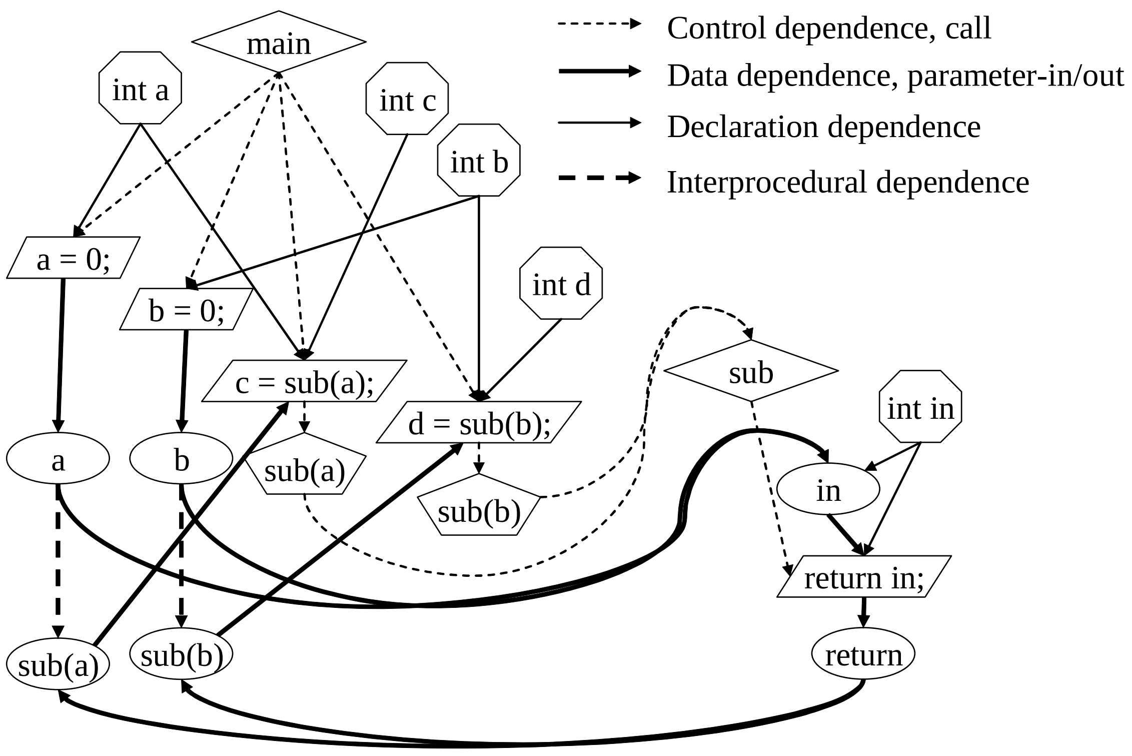


Figure 2.15: SDG of the code in Figure 2.14

and **b**, respectively.

When applying straightforward slicing approach to concurrent program, false data dependences among different concurrent processes are detected. For example, in Figure 2.17, when the program backwardly sliced from the node **c = b;** which corresponds to line 9 in Figure 2.16, the node **a = 1;** which corresponds to line 10 is reached. However, since **a = 1;** is always executed only after **c = b;** in Figure 2.16, this result is not accurate. Therefore, the slicing method proposed in [101] uses Control Flow Graph (CFG) as well as SDG to consider the execution order in each concurrent process. The computation amount of this algorithm is linear to the product of the number of nodes and the number of concurrent processes.

```
int a;
int b;
int c;
event e;
a = 0;
b = 0;
par{
  proc1{
     c = b;
     a = 1;
     notify(e);
  }
  proc2{
     wait(e);
     b = a;
  }
}
```

Figure 2.16: Example concurrent program for SDG

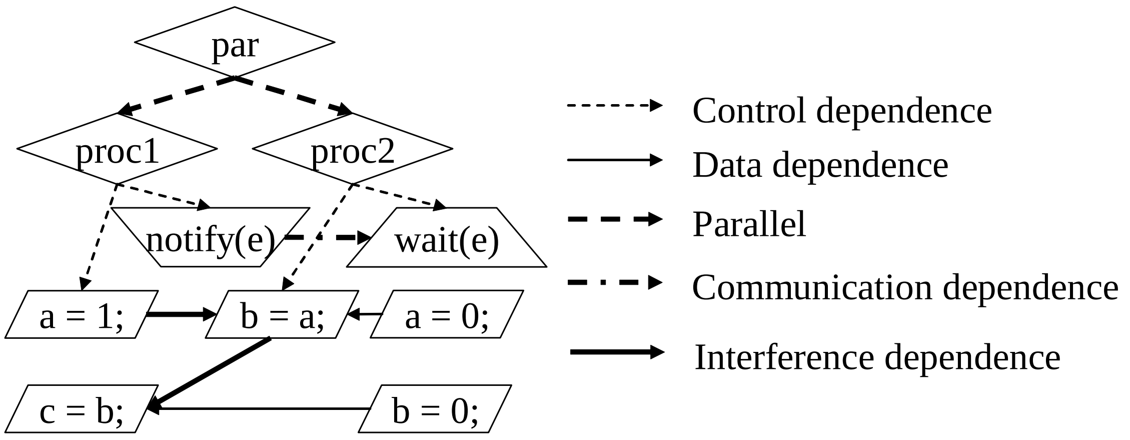


Figure 2.17: SDG of the code in Figure 2.16

System dependence graph for hardware design

Program slicing of Hardware Description Language (HDL) is firstly discussed in [124, 84]. They mentioned that the differences from software program slicing are infinite control loops, parallel executions, the notion of time including delay, and synthesizability. Infinite control loop is introduced by hardware modules run permanently. Then, hardware processes must be treated as blocks in infinite loops like *while(true)* in software program code. Parallel execution means all modules in a hardware design are executed parallely and parallel issues must be introduced to the dependence graph. The notion of time means that execution time or delay can be written in HDL code, and such statements cannot be ignored to trace dependence. Synthesizability means that although a sliced code is complete, there can be a case that it is not synthesizable. They proposed a program slicing method of VHDL in [81, 80] as the result of such a discussion. Each hardware module (process) is described as a process dependence graph. They introduced signal dependence which represents data dependence between parallel hardware processes through a shared signal.

Another program slicing method for VHDL is proposed in [35, 36]. Instead of creating dependence graphs directly represent VHDL codes, [35, 36] generate a program code whose dependencies among code portions are equivalent as that of an original VHDL circuit structure as a preprocess, and use software program slicing technique as the back-end.

In the above slicing methods, CFG is not required as the concurrent software slicing method proposed in [101], since hardware modules run iteratively, and do not have the problem about the false data dependences as shown with Figure 2.17.

System dependence graph for system level design

For system level design, SDG for SpecC description is proposed in [162, 161]. Here, SpecC language is introduced before the SDG.

SpecC[54] is a system level language which extends ANSI-C to describe hardware elements, such as hierarchical structures, concurrencies, synchronizations, bit-level variables, signals, registers, the notion of execution time.

SpecC has three types of classes, **behavior**, **channel**, and **interface**. Behavior is a unit of a behavioral procedure which corresponds to a class in software program or a functional module in hardware. Behavior has **port**s which represents a connection with other behaviors. Instead of **port**s, **interface**s can be used as connections with other behaviors. **interface** is an interface class which declares the functionality of an interface portion of a design, and a **channel** implements an **interface**. Interface portions in a design can be encapsulated with **channel**s and **interface**s.

Concurrency is represented by **par** statement. Processes under a **par** statement are executed concurrently. Those processes start at the same timing and also end at the same timing (co-start and co-end).

Synchronization is represented by **wait** statement, **notify** statement, and **event** variable. A **wait** statement stops the process execution until receiving a trigger of an argument event variable. A **notify** statement triggers an argument event variable.

Figure 2.18 shows an example SpecC code including hierarchical structure, concurrency, and synchronization. It has two interfaces **Receiver** and **Sender**, a channel **Ch** which implements those interfaces, and three behaviors **Sub1**, **Sub2**, and **Main**. As shown in Figure 2.19, **Sub1** and **Sub2** are connected with **Ch** through **Sender** and **Receiver**. Variables **a** and **b** are connected to the ports **i** and **o** in **Sub1** and **Sub2**, respectively. Main functions of **Sub1** and **Sub2** are executed concurrently with the **par** statement in **Main** behavior. Those main functions call the member functions of **Ch**, and **wait** and **notify** statements are executed in those functions for synchronization. After executing this example code, the value of variable **b** becomes equivalent to that of variable **a**.

In SpecC, bit-level variables, signal variables, and register variables are represented by **bit**, **signal**, and **buffered** declarator, respectively. The notion of time is also introduced with **waitfor** statements that put the time forward for the argument amounts of unit times.

In [162, 161], a C++ code whose dependency among code portions are equivalent to that of the original SpecC code is generated in a preprocess, and an ANSI-C/C++ program slicer CodeSurfer[37] is used as the back-end. In the generated SpecC SDG, control dependence edges are drawn from each **par** node to all process entries under the **par** node, from each **notify** node to corresponding **wait** nodes. Data dependence edges are also drawn for interprocedural data accesses. Those edges can be considered as call edges, communication dependence edges, and interference dependence edges in [101]. However, SDGs directly generated from Codesurfer are not accurate, since dependencies of concurrent codes cannot be represented equivalently

```
interface Receiver{
  int receive();
};

interface Sender{
  void send(int x);
};

channel Ch implements
  Receiver,Sender{
  event e;
  int data;
  int receive(){
    wait(e);
    return data;
  }
  int send(int x){
    data = x;
    notify(e);
  }
};
```

```
behavior Sub1(
  in int i, Sender s){
  void main(){
    s.send(i);
  }
};

behavior Sub2(
  out int o, Receiver r){
  void main(){
    o = r.receive();
  }
};

behavior Main(){
  Ch ch;
  int a, b;
  Sub1 sub1(a, ch);
  Sub2 sub2(b, ch);
  int main(){
    a = 1;
    par{
      sub1.main();
      sub2.main();
    }
  }
};
```

Figure 2.18: Example SpecC code

with sequential codes. Then, [162, 161] apply modifications after generating SDGs with Codesurfer. However, since data dependence edges among concurrent processes which correspond to interference dependence edges must be re-drawn, it is not efficient to use CodeSurfer as the back-end, and the method for concurrent programs proposed in [101] is better for this purpose.

In this thesis, an unified SDG for system level, behavioral level, and RTL code is proposed in Chapter 6 which is based on the method proposed in [101].

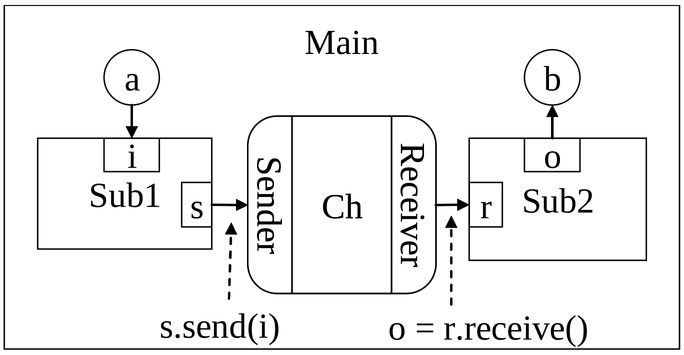


Figure 2.19: Structure of the code in Figure 2.18

2.4 Symbolic Simulation

Symbolic simulation[96] is a simulation where values of variables and meanings of operations are not interpreted. Then, exhaustive analyses can be performed, since one symbol can express all values of a variable.

Basic symbolic simulation process is performed by iteratively replacing each variable in expressions with an equivalent expression derived from previously executed assignments. The result of symbolic simulation is a set of equations each of which represents a relation between latest value of a variable and primary inputs. Since symbolic simulation is applied to FSMD in this thesis, the basic symbolic simulation method is introduced with FSMD.

Symbolic simulation for FSMD is performed for a sequence of state transitions from the first state at cycle 0. Let $(s_0 \cdot \cdot s_k) \in T$ be a sequence of state transitions, where s_j represents a state at cycle j, and k is the length of the sequence. For each symbolic simulation cycle, conditions among multiple cycles are generated. To express such conditions, the notion of timed symbolic value is required.

Although symbolic value has been introduced in Section 2.3.2, it is not enough for symbolic simulation, since symbolic simulation of FSMD has the notion of cycles. A same symbolic value can represent different values among multiple cycles. Then, symbolic value for each cycle, timed symbolic value, is introduced. Let I_j, V_j, and O_j denote set of timed symbolic values at cycle j of inputs, data registers, and outputs, respectively. Those timed symbolic values are defined for each cycle. Set of all timed symbolic values are denoted by I_T, V_T, O_T, respectively, and defined as follows.

$$I_T = \bigcup_{j|j\geq 0} I_j$$
$$V_T = \bigcup_{j|j\geq 0} V_j$$
$$O_T = \bigcup_{j|j\geq 0} O_j$$

Since K is a set of constants and they does not change over cycles, K can also be considered as timed symbolic values. Then, a set of function calls represent timed symbolic values is defined as follows.

$$F_{callT} = \{(f, e_1, e_2, ..., e_n) \mid f \in F, e_1, e_2, ..., e_n \in E_T\}$$

where $E_T = I_T \cup V_T \cup O_T \cup K \cup F_{callT}$ is a set of expressions represent timed symbolic values.

Then, symbolic simulation at cycle j, where $0 \leq j \leq k$, generates the following equation.

$$\begin{aligned} N_{Dj} \quad = \quad & \left(\bigwedge_{v\in V, a_v\in A_V, e\in E \ \mid \ L_A(a_v)=(v,e), (s_j,a_v)\in P} v_{j+1} = e_j \right) \wedge \\ & \left(\bigwedge_{o\in O, a_o\in A_O, e\in E \ \mid \ L_A(a_o)=(o,e), (s_j,a_o)\in P} o_j = e_j \right) \end{aligned} \tag{2.2}$$

where, $e_j \in E_j$ $v_j \in V_j$, and $o_j \in O_j$ represent the timed symbolic values of e, v, and o at cycle j, respectively. The first term of Equation 2.2 shows that the next value of the data register in the left hand side of an assignment is equivalent to the current value of the right hand side expression. The second term shows that the current value of the output in the left hand side of an assignment is equivalent to the current value of the right hand side expression. The conjunction of the generated equations is called data condition in this thesis since it is derived from the datapath assignments. Let $N_D : T \rightarrow E_T$ denote a function which returns the data condition of a given sequence of state transitions, where the returned expression is as follows.

$$\bigwedge_{j|0\leq j\leq k} N_{Dj}$$

The following equation can also be generated from the symbolic simulation at cycle j.

$$N_{Cj} = L_G(Q((s_j, s_{j+1})))$$

This equation simply represents that the transition condition of a transition on the sequence must be *true*. The conjunction of these generated equations is called control condition in this thesis since it is derived from the controller state transition conditions. Let $N_C : T \rightarrow E_T$ denote a function which returns the control condition of a given sequence of state transitions, where the returned expression is as follows.

$$\bigwedge_{j|0 \leq j \leq k-1} N_{Cj}$$

Figure 2.20 shows the equations generated with symbolic simulation for the sequence of state transitions $(s_0, s_1, s_2, s_1) \in T$ in Figure 2.11. All equations in the figure must be true when the FSMD is assumed to transit through the path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1$. From these equations, the symbolic value of arbitrary expression at arbitrary cycle which is represented only with primary inputs can be generated. For example, the symbolic value of x at cycle 3 can be calculated as follows.

$$\begin{aligned} x_3 &= x_2 \times 10 \\ &= x_1 \times 10 \\ &= in_0 \end{aligned}$$

Such symbolic simulation techniques are widely used in the verification field. [13, 14] apply symbolic simulation to check pre-defined assertions statically which can also be used for model checking. In their methods, an efficient data structure to contain symbolic expressions, Maximally-Shared Graph which is originally proposed in [97], is used. In Maximally-Shared Graph, common sub-expressions are shared. [71] uses symbolic simulation with random simulation and bounded model checking to search state space efficiently by switching those engines with the conditions about state coverage and execution time. [90, 91] apply symbolic simulation for equivalence checking between two FSMDs. Symbolic simulation is performed for each path between conditional branches in the FSMDs, and result

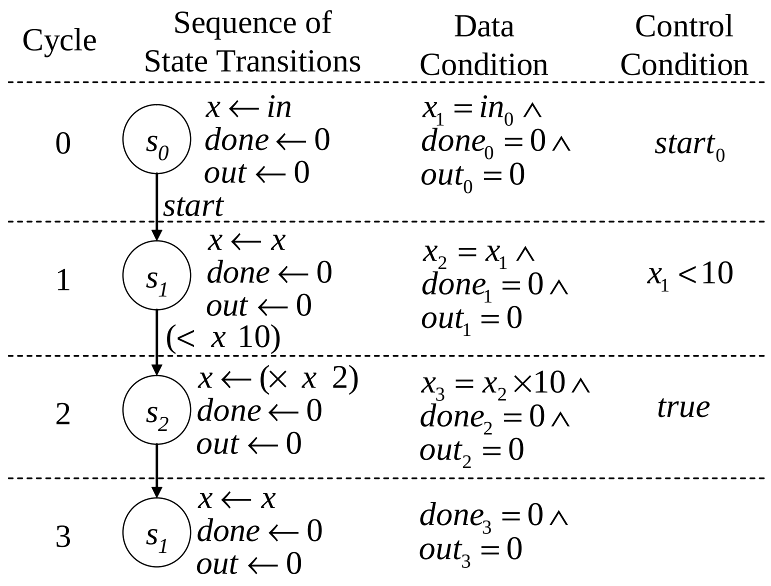


Figure 2.20: Symbolic simulation result for the path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1$ in Figure 2.11

symbolic expressions of corresponding paths are compared to prove the equivalence. In [139, 140, 137, 138, 143, 113], equivalence class based equivalence checking methods are proposed. Equivalence class is a set where all member expressions are equivalent, and equivalence class based symbolic simulation is performed by iteratively allocating new expressions to equivalence classes. [139, 140, 137, 138] target on hardware designs in RTL and gate level, and designs are unrolled for a given number of cycles as a preprocess.

[143, 113] target on ANSI-C based system-level and behavioral level designs. Since the equivalence checking method used in Chapter 4 is similar to their methods, details of them are introduced here. Before the verification, all loops must be unrolled. First, each pair of corresponding inputs is assigned to a same equivalence class. Verification is performed for each execution path based on the equivalences of inputs. For each assignment on the path, since its left hand side and right hand side are equivalent, those expressions are assigned to a same equivalence class. Finally, if every pair of corresponding outputs is in a same equivalence class, the two designs are proved to be equivalent.

In this thesis, symbolic simulation is used as a verification engine in equivalence checking, or a method to generate formulae which corresponds to a set of states at the boundary of bounded model checking results.

Chapter 3

Multi-Level Bounded Model Checking

3.1 Introduction

As discussed in Section 2.1.1, although bounded model checking[22] is a strong technique to model check large designs, it is difficult to verify long bounds since the worst complexity increases exponentially with the bound[32]. Finite bounds also mean that the results are not complete since counter examples can be found with longer bounds. Therefore, there are researches which prove the completeness of model checking with the result of a finite bound. [32] mentions completeness threshold which is the minimum bound that completeness of the model checking can be proved when no counter-examples are found by bounded model checking. That threshold can be considered as a minimum fixed point. However, since the threshold increases exponentially with the number of state variables, computation amount of complete bounded model checking is doubly exponential to the verified design size. [32, 12] calculates such completeness thresholds using Büchi automata. Induction is also used to prove the completeness[153, 41, 8].

On the other hand, though BDD based symbolic model checking can verify infinite bounds, it can verify only much smaller designs compared to bounded model checking. Then, [65] uses SAT instead of BDD for image computation. [118] also eliminates universal quantifiers from the Boolean formula so that it can be solved by SAT solvers. Their experimental results showed that SAT and BDD can be complementary relation since SAT is better for some examples and BDD is better for the others.

Though those methods try to verify model checking problems without any bound, it cannot reduce the computation amount itself. Then methods which can improve the model checking performance are strongly required.

Incremental SAT solving method[182, 159, 175, 89, 18, 19] is one of them. It learns clauses from the previous related problems, and can improve the solution time for the next problem. Since bounded model checking instances of bounds k and $k+1$ are similar, it is a good application of incremental SAT solving. [44, 68] combine incremental SAT solving with the induction based method.

[119] proposes a method which uses interpolants from unsatisfiability proof of counter examples to over-approximate reachability. The result of this method is

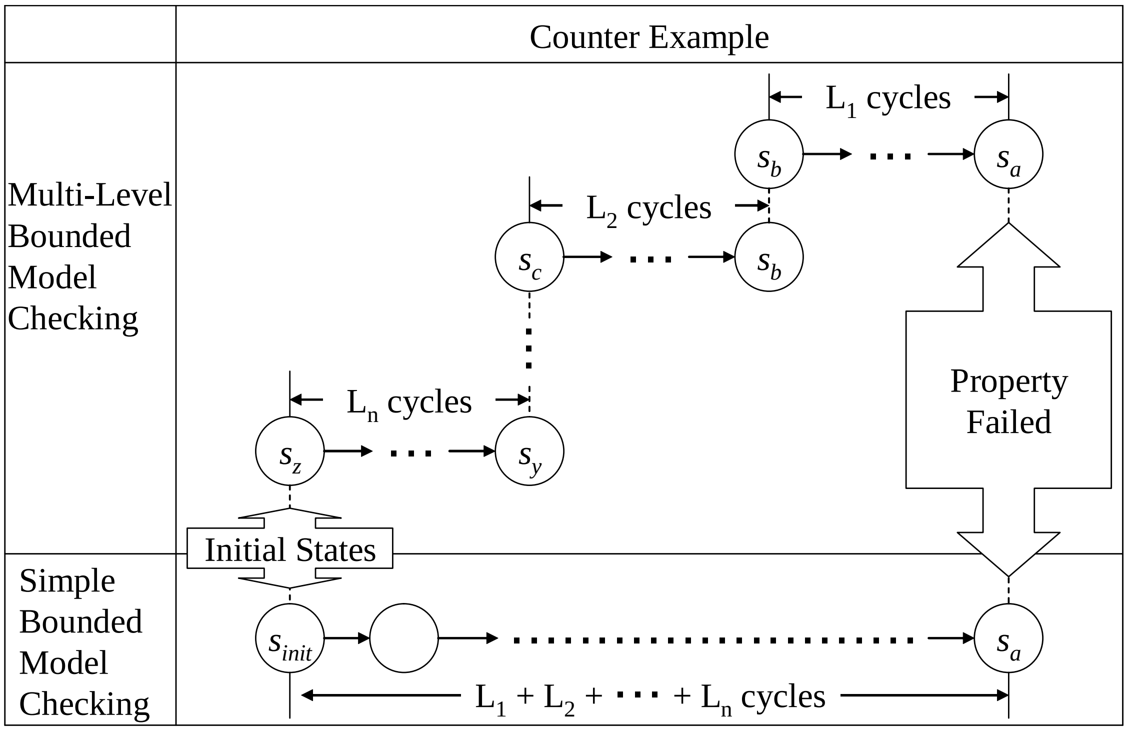


Figure 3.1: Multi-Level Bounded Model Checking

also complete, and it can improve bounded model checking performance as the other heuristics.

Abstraction (and refinement) methods which have been introduced in Section 2.1.1 can also used as heuristics to improve the performance of bounded model checking.

In this section, a new heuristic for bounded model checking which concatenates multiple bounded model checking results by inductive approach and symbolic simulation is proposed. During the verification flow, verification is decomposed to divide one long counter example into multiple ones as shown in Figure 3.1.

The basic flow of the proposed method is as follows. The first decomposed bounded model checking checks the original property without considering the initial states (all states are assumed as initial states). If the property is proved to be correct, then the verification ends since those assumed initial states include the original initial states. Otherwise, a counter example is generated. The second bounded model checking checks the reachability from the initial states to the first state of the counter example with the property that “The first state of the counter example is an error state”. If this property fails and another counter example is generated, then the first counter example is guaranteed to be correct, and the original property is proved not to be satisfied. A complete counter example can be generated by concatenating those two counter examples. The second bounded model checking can also be recursively decomposed by the same way.

Obviously, verification bounds of the decomposed bounded model checking can be smaller than that of the original bounded model checking. The computation amount of each decomposed bounded model checking is expected to be much smaller than that of the original one since the computation amount is exponent with the bound. Then, the method can handle larger bounds when all decomposed bounded model checking generates counter examples. However, if a bounded model checking does not generate a counter example, the bounded model checking one step earlier must be retried with a refined initial state condition not to generate the same counter example again.

In the proposed method, symbolic simulation is used to expand each connection state (the first state of the counter example) to a set of states by generating a more general condition. Each bounded model checking is also accelerated with this technique. When applying symbolic simulation, control of the design is considered.

Though the proposed method is a heuristic, since it does not modify the bounded model checking algorithm itself, it can be applied with other methods together, such as incremental SAT solving, interpolation based methods, and abstraction/refinement methods which have already been introduced in this section.

The target designs are FSMDs which have been introduced in Section 2.3.1.

The organization of this section is as follows. Section 3.2 introduces definition of bounded model checking used in this section. Section 3.3 proposes a simple algorithm to concatenate two bounded model checking result, and that is the basic idea of the two level bounded model checking method proposed in Section 3.4. Section 3.5 extends this approach to multi level more than two. Experimental results with a couple of examples are reported in Section 3.6. In Section 3.7, this section is concluded with future directions.

3.2 Basic Bounded Model Checking Algorithm

General bounded model checking[22] is performed for an FSM by translating the model checking problem into satisfiability problem. For that purpose, the FSM is unfolded from the initial states for a given bound as follows. Let $M_{FSM} \models_k \varphi$, where M_{FSM} is an FSM, k is a positive integer number, and φ is a property, denote M_{FSM} satisfies φ within k bounds from the initial states. Such the relation is represented by the following formula.

$$\begin{aligned}&M_{FSM} \models_k \varphi \Longleftrightarrow \forall s_0 \in S, \forall s_1 \in S, \cdots, \forall s_k \in S,\\&Cond_{init} \wedge Cond_{trans} \rightarrow Cond_{prop}\end{aligned} \tag{3.1}$$

where

$$\begin{aligned}Cond_{init} &:= s_0 \in S_0\\Cond_{trans} &:= \bigwedge_{j=0}^{k-1} R(s_j, s_{j+1})\\Cond_{prop} &:= \bigwedge_{j=0}^{k} P_{\varphi}(s_j)\end{aligned} \tag{3.2}$$

Here, $Cond_{init}$ is the condition of the initial states, $Cond_{trans}$ shows the unfolding of the transition relation, and $Cond_{prop}$ represents the condition that the property is satisfied in every sequence. To check this formula, the following formula is translated into conjunctive normal form (CNF), and solved by SAT solvers with treating $s_0, s_1, \cdots, s_k \in S$ as variables.

$$Cond_{init} \wedge Cond_{trans} \wedge \neg Cond_{prop} \tag{3.3}$$

When a positive number n is given as a bound, Formula 3.3 is iteratively checked with incrementing k from 0 to $n-1$. When there is no set of $s_0, s_1, \cdots, s_k \in S$ which makes Formula 3.3 true for all $0 \leq k \leq n-1$, the property is satisfied within the bound n. On the other hand, there is a set of $s_0, s_1, \cdots, s_k \in S$ which makes Formula 3.3 true with a number k, which is a counter example. Then, a counter example of FSM CE_{FSM} is defined as follows.

$$CE_{FSM} = (s_0, s_1, \cdots, s_k) \in S^{k+1} \mid \bigwedge_{j=0}^{k-1} R(s_j, s_{j+1})$$

where k is the length of the counter example. Such counter examples do not satisfy the property.

Similarly, bounded model checking formula for DFSM is defined as follows.

$$\begin{aligned}
&M_{DFSM} \models_k \varphi \Longleftrightarrow \\
&\forall s_0 \in S, \forall s_1 \in S, \cdots, \forall s_k \in S, \forall i_0 \in I_V, \forall i_1 \in I_V, \cdots, \forall i_{k-1} \in I_V \\
&Cond_{init} \wedge Cond_{trans} \rightarrow Cond_{prop}
\end{aligned} \tag{3.4}$$

where

$$\begin{aligned}
Cond_{init} &:= s_0 = \alpha \\
Cond_{trans} &:= \bigwedge_{j=0}^{k-1} s_{j+1} = T(s_j, i_j) \\
Cond_{prop} &:= \bigwedge_{j=0}^{k} P_\varphi(s_j)
\end{aligned} \tag{3.5}$$

The differences from Formula 3.1 and Formula 3.4 are as follows.

- Not only states but also inputs are considered with the universal quantifiers.
- The initial state is fixed since an FSMD has only one initial state.
- Transition function is used instead of transition relation.

To check Formula 3.4, Formula 3.3 is translated into CNF with $Cond_{init}$, $Cond_{trans}$, and $Cond_{prop}$ defined in Formula 3.5, and $s_0, s_1, \cdots, s_k \in S$, $i_0, i_1, \cdots, i_{k-1} \in I_V$ are treated as variables. A counter example of DFSM CE_{DFSM} is defined as follows.

$$CE_{DFSM} = ((s_0, s_1, \cdots s_k) \in S^{k+1}, (i_0, i_1, \cdots, i_{k-1}) \in {I_V}^k) \mid \bigwedge_{j=0}^{k-1} s_{j+1} = T(s_j, i_j)$$

Note that the minimum information required to specify the counter example is only the input sequence $(i_0, i_i, ..., i_{k-1})$ since sequence of states can be obtained by executing the DFSM with the input sequence.

Bounded model checking formula for FSMD in Definition 5 is also defined as follows.

$$M_{FSMD} \models_k \varphi \Longleftrightarrow$$

$$\begin{aligned}
&\forall s_0 \in S, \forall s_1 \in S, \cdots, \forall s_k \in S, \forall i_0 \in I_V, \forall i_1 \in I_V, \cdots, \forall i_k \in I_V,\\
&\forall v_0 \in V_V, \forall v_1 \in V_V, \cdots, \forall v_k \in V_V, \forall o_0 \in O_V, \forall o_1 \in O_V, \cdots, \forall o_k \in O_V,\\
&Cond_{init} \wedge Cond_{trans} \rightarrow Cond_{prop} \qquad (3.6)
\end{aligned}$$

where

$$\begin{aligned}
Cond_{init} &:= s_0 = \alpha \wedge v_0 = \beta\\
Cond_{trans} &:= \bigwedge_{j=0}^{k-1}(s_{j+1} = fs(s_j, i_j, v_j)) \wedge \bigwedge_{j=0}^{k-1}(v_{j+1} = fv(s_j, i_j, v_j)) \wedge \bigwedge_{j=0}^{k} o_j = f_o(s_j, i_j, v_j))\\
Cond_{prop} &:= \bigwedge_{j=0}^{k} P_\varphi(s_j, v_j, o_j) \qquad (3.7)
\end{aligned}$$

Additional elements from Formula 3.4 and Formula 3.5 are as follows.

- Not only states and inputs, but also data registers and outputs are considered with the universal quantifiers.
- The initial state condition is separated into the initial control state condition and initial data register value condition.
- Not only transition function but also next register value function and output function are used.
- Property is evaluated with states, register values, and outputs, instead of only states.

To check Formula 3.6, Formula 3.3 is translated into CNF with $Cond_{init}$, $Cond_{trans}$, and $Cond_{prop}$ defined in Formula 3.7, and $s_0, s_1, \cdots, s_k \in S$, $i_0, i_1, ..., i_k \in I_V$, $v_0, v_1, \cdots, v_k \in V_V$, $o_0, o_1, \cdots, o_k \in O_V$ are treated as variables. A counter example is defined with the following formula.

$$\begin{aligned}
CE_{FSMD} = ((s_0, s_1, \cdots, s_k) \in S^{k+1}, (i_0, i_1, \cdots, i_k) \in {I_V}^{k+1},\\
(v_0, v_1, \cdots, v_k) \in {V_V}^{k+1}, (o_0, o_1, \cdots, o_k) \in {O_V}^{k+1}))\\
\mid \bigwedge_{j=0}^{k-1} s_{j+1} = fs(s_j, i_j), \bigwedge_{j=0}^{k-1} v_{j+1} = fv(v_j, i_j), \bigwedge_{j=0}^{k} o_j = fo(s_j, i_j) \qquad (3.8)
\end{aligned}$$

The minimum information to specify the counter example is the first state s_0 and data register value v_0 and an input sequence $(i_0, i_1, \cdots i_k)$ since other parameters can be obtained by executing the FSMD with the input sequence.

Formula 3.7 can be replaced with the following formula for Definition 6.

$$
\begin{aligned}
Cond_{init} &:= s_0 = \alpha \wedge v_0 = \beta \\
Cond_{trans} &:= \bigwedge_{j=0}^{k-1} ((s_j, s_{j+1}) \in R \wedge (i_j, v_j) \in Q((s_j, s_{j+1}))) \wedge \\
&\quad\ \bigwedge_{j=0}^{k-1} \left(v_{j+1} = \prod_{a_v \ \mid \ a_v \in A_V, (s_j, a_v) \in P} a_v(i_j, v_j) \right) \wedge \\
&\quad\ \bigwedge_{j=0}^{k} \left(o_j = \prod_{a_o \ \mid \ a_o \in A_O, (s_j, a_o) \in P} a_o(i_j, v_j) \right) \\
Cond_{prop} &:= \bigwedge_{j=1}^{k} ((s_j, v_j, o_j) \in P) \qquad (3.9)
\end{aligned}
$$

where

$$
\begin{aligned}
I_V &= \prod_{i \ \mid \ i \in I} \sigma(i) \\
O_V &= \prod_{o \ \mid \ o \in O} \sigma(o) \\
V_V &= \prod_{v \ \mid \ v \in V} \sigma(v)
\end{aligned}
$$

The notions of symbol, assignment, transition condition are additionally introduced in Formula 3.9. The counter example definition is also modified from Formula 3.8 as follows.

$$
\begin{aligned}
CE_{FSMD} = ((s_0, s_1, \cdots, s_k) \in S^{k+1}, (i_0, i_1, \cdots, i_k) \in {I_V}^{k+1}, \\
(v_0, v_1, \cdots v_k) \in {V_V}^{k+1}, (o_0, o_1, \cdots, o_k) \in {O_V}^{k+1})) \\
\mid \bigwedge_{j=0}^{k-1} \left((i_j, v_j) \in Q((s_j, s_{j+1})) \wedge v_{j+1} = \prod_{a_v \in A_V \ \mid \ (s_j, a_v) \in P} a_v(i_j, v_j) \right), \\
\bigwedge_{j=0}^{k} \left(o_j = \prod_{a_o \in A_O \ \mid \ (s_j, a_o) \in P} a_o(i_j, v_j) \right) \qquad (3.10)
\end{aligned}
$$

In this thesis, the bounded model checking definition for FSMD, Formula 3.6 and Formula 3.3 with Formula 3.9, are used since the method proposed in this chapter targets on FSMD defined in Definition 6. Formula 3.10 is also used as the definition of counter example. However, similar methods can be applied to other representations if they can be translated to FSM or DFSM, since $Cond_{init}$, $Cond_{trans}$, and $Cond_{prop}$ in all of the above four definitions corresponding to each other.

3.3 Concatenation of Two Bounded Model Checking Results

Since computation amount of bounded model checking increases exponentially with the given bound, one simple idea to model check with a large bound is to separate the bound into two shorter bounds, and model check with each bound separately. If the given bound is assumed to be n and both the separated shorter bounds are $n/2$. The complexity can be considered to be refined from $O(e^n)$ to $O(2e^{n/2})$. Then the complexity becomes $2/e^{n/2}$ in successful cases. Figure 3.2 shows the flow of such a concatenation of two bounded model checking results.

The inputs of this flow are an FSMD design which is the verification target and a property to be checked. The property can be in any representation while it can be translated to the property condition formula $Cond_{prop}$ in Formula 3.3. This flow can be divided to the following two steps.

- Bounded model checking without the initial state condition
- Reachability analysis to the first state of the counter example, and initial condition refinement

Bounds for both steps (bounded model checkings) are given by users. Details of those steps are explained in the following sections.

3.3.1 Bounded Model Checking without the Initial State Condition

The first step is simply performed by replacing the term of the initial state condition in Formula 3.3 with *true*. The formula to be checked with SAT solvers becomes as

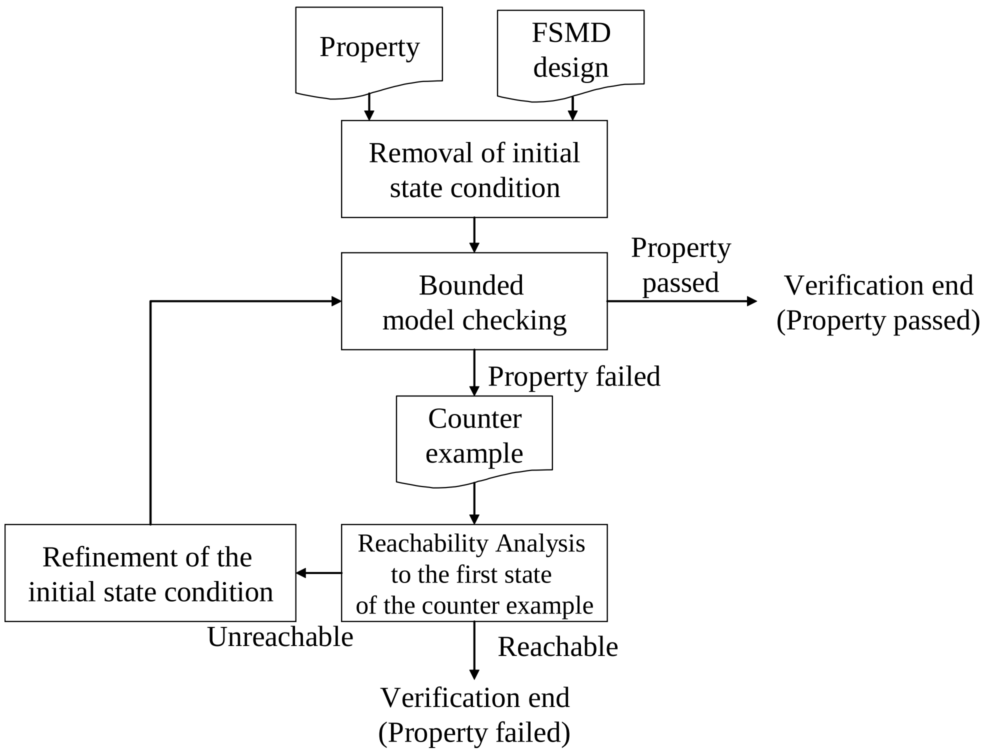

Property
FSMD design
Removal of initial state condition
Bounded model checking
Property passed
Verification end (Property passed)
Property failed
Counter example
Reachability Analysis to the first state of the counter example
Refinement of the initial state condition
Unreachable
Reachable
Verification end (Property failed)

Figure 3.2: Concatenation of two bounded model checking results

follows.

$$true \wedge Cond_{trans} \wedge \neg Cond_{prop} \tag{3.11}$$

In Formula 3.11, $Cond_{init}$ in Formula 3.9 defining the initial state and initial data register value are ignored. This corresponds to treating all states in $S \times V_V$ as initial states.

Formula 3.11 is checked with the number of unfolding steps k incremented from 0 to a given bound as follows.

- When there are no variable assignments which make Formula 3.11 true with $k = 0$, there are no states which violate the property. Then Formula 3.3 also cannot be true. In the case, the result is decided without the second step.

- When there is a variable assignment which makes Formula 3.11 true with $k = j$, where j is less than the bound, there is a counter example which violates the property, such as

$$CE_{FSMD1} = ((s_0, \cdot\cdot, s_j), (i_0, \cdot\cdot, i_j), (v_0, \cdot\cdot, v_j), (o_0, \cdot\cdot, o_j))$$

It can be generated from the result of SAT solvers. In this case, k is incremented to find a longer counter example which can step closer to the initial state.

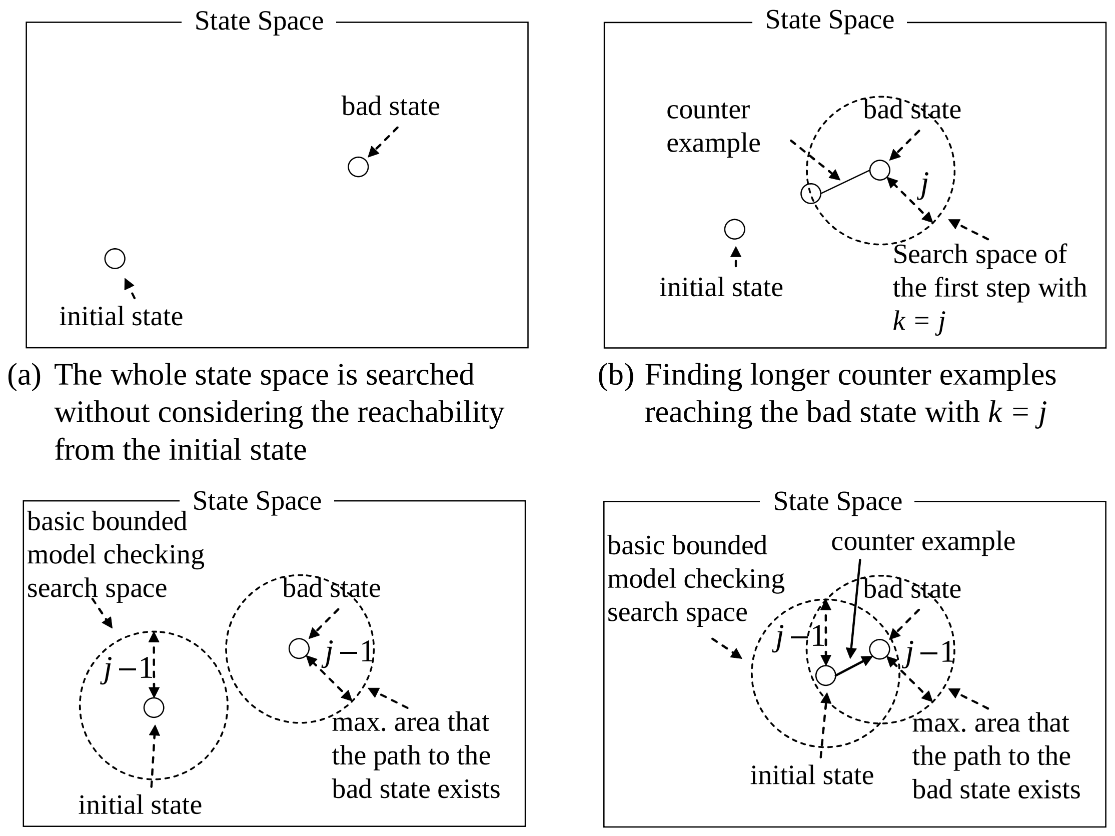
- When there is a variable assignment which makes Formula 3.11 true with $k = j - 1$ and not with $k = j$, the maximum k which make Formula 3.11 true is guaranteed to be $j - 1$. It is also valid for Formula 3.3 since any assignments (counter examples) which make Formula 3.3 true also make Formula 3.11 true. Therefore, basic bounded model checking with Formula 3.3 can be applied with the bound $j - 1$. If no counter examples are found with Formula 3.3 and the bound $j - 1$, the property is proved to be true for any bounds.

- When k becomes equal to the given bound, this step is finished and the next step is applied.

The first step is introduced more intuitively with Figure 3.3. In the figure, the state space corresponds to $S \times V_V$, and assume that the distance between two states represents minimum number of state transitions required to move between those two states.

Firstly, the property is checked without considering the initial state by checking Formula 3.11 with $k = 0$ as shown in Figure 3.3 (a). Here, all states in the state space is searched without considering the reachability from the initial state. When no counter examples are found, it means that there are no states which violate the property in the state space. Then, the original bounded model checking problem in Formula 3.3 also succeeds with any bounds. On the other hand, when there is a state which violates the property (bad state), k is incremented and Formula 3.11 is checked again to find a path to the bad state.

The picture of checking Formula 3.11 with $k = j$ is shown in Figure 3.3 (b). The area in the dotted circle with radius j is the searched area with the check. If a counter example is found, its length must be j as shown in the figure. If no counter


State Space
bad state
initial state
State Space
counter example
bad state
j
initial state
Search space of the first step with k = j
State Space
basic bounded model checking search space
bad state
j −1
j −1
initial state
max. area that the path to the bad state exists
State Space
basic bounded model checking search space
counter example
bad state
j −1
j −1
initial state
max. area that the path to the bad state exists

(a) The whole state space is searched without considering the reachability from the initial state

(b) Finding longer counter examples reaching the bad state with $k = j$

(c) Bad state is unreachable from the initial state when the initial state is not in the maximum area that the counter example to the bad state exists

(d) When the initial state is in the maximum area that the counter example to the bad state exists, basic bounded model checking can find a counter example

Figure 3.3: State space image of the first step of two bounded model checking results concatenation

examples are found, it means that there are no states on the circumference of the circle. While counter examples whose length is k exist, the check is continued with incrementing k until it becomes equal to the given bound. The reason to increment k is to find a longer feasible path which reaches a bad state. The first state of a long counter example path can be closer to the initial state than bad states as shown in Figure 3.3 (b). In such a case, reachability analysis to the state can be easier than that to the bad states.

Let's consider the case that a counter example is found with $k = j - 1$ but not with $k = j$. This situation guarantees that the maximum path length to reach bad states is $j-1$. Then the area includes states reachable to a bad state is in the circle with radius $j-1$ whose center is the bad state as shown in Figure 3.3 (c) and (d). Then, what should be checked is that the initial state is within the area. The result is identical to that of the basic bounded model checking from the initial state with Formula 3.3 and bound $j-1$ since the only difference is the directions of reachability analyses as shown in Figure 3.3 (c) and (d). If no counter examples from the initial state are found as shown in Figure 3.3 (c), it means that the distances between the initial state and bad states are more than $j-1$. Since there are no paths reach the bad states whose lengths are more than $j-1$, the bad states are absolutely unreachable from the initial state, and the property is proved to be true. On the other hand, if a counter example is found as shown in Figure 3.3(a), that counter example is a complete counter example from the initial state to one of the bad states which violates the property. Therefore, a complete result can be got without the second step.

The only case the second step is applied is when k becomes equal to the given bound.

3.3.2 Reachability Analysis to the Counter Example

When k becomes equal to a given bound k_1 in the first step, there must be a counter example which makes the formula true, such as

$$CE_{FSMD1} = ((s_0^1, \cdot\cdot, s_{k_1}^1), (i_0^1, \cdot\cdot, i_{k_1}^1), (v_0^1, \cdot\cdot, v_{k_1}^1), (o_0^1, \cdot\cdot, o_{k_1}^1))$$

The next step is to check the reachability to the counter example by checking the reachability to the first state of the counter example. The first state is a pair (s_0^1, v_0^1). The other information, such as remaining states and data register values and input sequence, is not used since the counter example sequence can be specified only by (s_0^1, v_0^1) and the input sequence.

The reachability analysis is performed as a bounded model checking with replacing the negation of the property condition in Formula 3.3 with a reachability condition, such that

$$Cond_{reach} := s_k = s_0^1 \wedge v_k = v_0^1$$

Then, the formula to be checked becomes

$$Cond_{init} \wedge Cond_{trans} \wedge Cond_{reach} \tag{3.12}$$

In this second step, k is incremented from 0 to a given bound as the original bounded model checking.

If there is an assignment which makes Formula 3.12 true with $k = k_2$, a counter example which makes Formula 3.12 true, such as

$$CE_{FSMD2} = ((s_0^2, \cdot\cdot, s_{k2}^2), (i_0^2, \cdot\cdot, i_{k2}^2), (v_0^2, \cdot\cdot, v_{k2}^2), (o_0^1, \cdot\cdot, o_{k2}^2))$$

, can be generated.

Then, a complete counter example from the initial state to a bad state is generated by concatenating those two counter examples as follows.

$$\begin{aligned} CE_{FSMD12} = \quad & (s_0^2, \cdot\cdot, s_{k_2}^2 = s_0^1, \cdot\cdot, s_{k_1}^1) \in S^{k_1+k_2+1}, \\ & (i_0^2, \cdot\cdot, i_{k_2-1}^2, i_0^1, \cdot\cdot i_{k_1}^1) \in {I_V}^{k_1+k_2+1}, \\ & (v_0^2, \cdot\cdot, v_{k_2}^2 = v_0^1, \cdot\cdot v_{k_1}^1) \in {V_V}^{k_1+k_2+1}, \\ & (o_0^2, \cdot\cdot, o_{k_2-1}^2, o_0^1, \cdot\cdot o_{k_1}^1) \in {O_V}^{k_1+k_2+1}) \end{aligned}$$

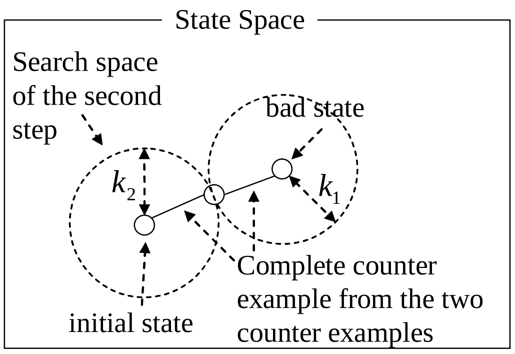
On the other hand, when there are no assignments which make the formula true, the state (s_0^1, v_0^1) is unreachable from the initial state within the given bound. In this case, the two steps are retried from the first step. The initial state condition is refined per each trial as follows to avoid generating same counter examples.

$$Cond'_{init} := Cond'_{init} \wedge \neg Cond_{reach}$$

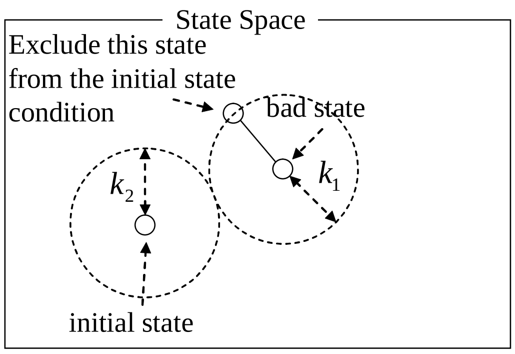
where, the initial $Cond'_{init}$ is $true$. Then the following formula is checked as the first step of the next trial instead of Formula 3.11.

$$Cond'_{init} \wedge Cond_{trans} \wedge \neg Cond_{prop} \tag{3.13}$$

The same counter examples cannot be generated with Formula 3.13 since the initial state condition $Cond'_{init}$ excludes counter examples from the state (s_0^1, v_0^1) which is


State Space
Search space of the second step
bad state
k_2
k_1
Complete counter example from the two counter examples
initial state

(a) Checking reachability from the initial state to the first state of the counter example as the second step



(b) Initial state condition is refined when the second step fails

Figure 3.4: State space image of the second step of two bounded model checking results concatenation

the first state of the previous counter example. Such the refinement is repeated until a complete counter example is found or the refined initial state condition $Cond'_{init}$ becomes (α, β) which means all states except the original initial state are excluded.

After at least one refinement, note that the property is guaranteed to be true only within the sum of the given bounds $k_1 + k_2$ when no counter examples are found with $k = i$ where $0 \leq i \leq k_1$ at the first step. It is because there can be counter examples through excluded states at the refinements which can be found with larger k_2.

Here, the second step is also introduced more intuitively with the state space images in Figure 3.4. Let's assume that a bound given for the first step is k_1, and a counter example is found in $k = k_1$. In such a case, the length of the counter example is k_1. The first states of such counter examples are on the circumference of the circle representing the search area of the first step bounded model checking in Figure 3.4 (a) and (b).

In the second step, reachability from the initial state to such a state is checked with Formula 3.12. Let's assume that a given bound of this step is k_2. The search area of this step is shown in the circle whose center is the initial state and radius is k_2 as shown in Figure 3.4 (a) and (b).

When the first state of the counter example is in the area, a counter example from the initial state to the state is found as shown in Figure 3.4 (a). In such a case, a complete counter example is generated by concatenating the two counter examples from the first step and the second step. Such a complete counter example is from the initial state to the bad state, and it is a proof of the property violation.

On the other hand, when the first state of the counter example is not in the area as shown in Figure 3.4 (b). The second step fails. In such a case, the first state of the counter example is excluded from the initial state condition $Cond'_{init}$ which initially includes all states not to generate the same counter example again. Then, the verification is retried from the first step with the refined initial state condition.

3.3.3 Algorithm

In this section, the algorithm which formally defines the method explained in this section is shown.

Algorithms of the first step and the second step are shown in Algorithm 1 and Algorithm 2, respectively. The main algorithm is defined in Algorithm 3, and it uses both Algorithm 1 and Algorithm 2 as subroutines.

Note that Algorithm 3 is clearly not efficient since the number of refinements can be huge. In the best case, no refinements are required. However, in the other cases, refinements must be applied iteratively while counter examples are not found, and only a single state $(s, v) \in S \times V_V$ is excluded from the initial state space per each refinement. In the worst case, it is continued until all states except the original initial state (α, β) are excluded. Then, when the number of states are n and the number of data register values are m, the maximum number of refinements can be $m \times n - 1$. This number is huge since n and m increase exponentially with the design size. A refined method which overcomes the problem is proposed in the next section.

3.3.4 Example

Figure 3.5 shows a simple example and a property for the method proposed in this section. $s_a, s_b, \cdots, s_f \in S$ are states where $\alpha = s_a$. $in \in I$ is an input where $I_V =$

Algorithm 1 *FirstStep*: Bounded model checking without initial state condition

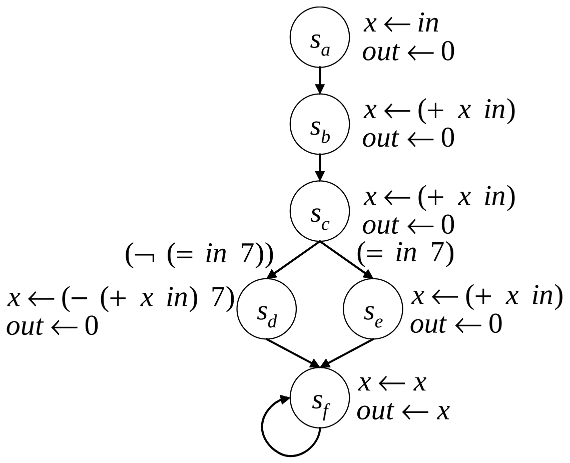
```
Require: $cond_{prop}$, $cond'_{init}$ {given initial state condition}, $k_1 \geq 0$ {bound of the first step}
 1: for $k = 0$ to $k_1$ do
 2:   $ce_1 \leftarrow BMC(cond'_{init}, \neg cond_{prop}, k)$ {Apply bounded model checking to $FSMD$ with $cond'_{init}$, $\neg cond_{prop}$, and unfolding step $k$. It returns a counter example or $NULL$ when no counter example is found}
 3:   if $ce_1 = NULL$ then
 4:     if $k = 0$ then
 5:       return $NULL$ {Property is true within the bound}
 6:     else
 7:       $cond_{init} \leftarrow ((s_0, v_0) = (\alpha, \beta))$ {General initial state condition}
 8:       $ce \leftarrow BMC(cond_{init}, \neg cond_{prop}, k)$ {Apply basic bounded model checking}
 9:       if $ce = NULL$ then
10:         return $NULL$ {Property is true within the bound}
11:       else
12:         return $ce$ {Property is violated with $ce$}
13:       end if
14:     end if
15:   end if
16: end for
17: return $ce$ {Go to the second step}
```

Algorithm 2 *SecondStep*: Reachability analysis from the initial state

```
Require: $cond_{reach}$ {reachability condition}, $k_2 \geq 0$ {bound for the second step}
 1: $cond_{init} \leftarrow ((s_0, v_0) = (\alpha, \beta))$ {General initial state condition}
 2: for $k = 0$ to $k_2$ do
 3:   $ce_2 \leftarrow BMC(cond_{init}, cond_{reach}, k)$ {Apply bounded model checking}
 4:   if $ce_2 \neq NULL$ then
 5:     return $ce2$ {A counter example from the initial state which makes $cond_{reach}$ true}
 6:   end if
 7: end for
 8: return $NULL$ {$cond_{reach}$ cannot be true within the bound $k$}
```

Algorithm 3 Concatenating two bounded model checking results

```
Require: $cond_{prop}$, $k_1 \geq 0$ {bound for the first step}, $k_2 \geq 0$ {bound for the second step}
 1: $cond'_{init} \leftarrow true$ {Initialization of initial state condition}
 2: loop
 3:   $ce_1 \leftarrow FirstStep(cond_{prop}, cond'_{init}, k_1)$ {Algorithm in Figure 1 is applied}
 4:   if $ce_1 = NULL$ then
 5:     return true {Property is true within the bound}
 6:   else if $ce_1 = (\alpha, \beta)$ then
 7:     return $ce_1$ {Counter example is generated without the second step}
 8:   else
 9:     $cond_{reach} \leftarrow ((s_0, v_0) = GetFirstState(ce_1))$ {Get first state condition}
10:     $ce_2 \leftarrow SecondStep(cond_{reach}, k_2)$ {Algorithm in Figure 2 is applied}
11:   end if
12:   if $ce_2 \neq NULL$ then
13:     return $(ce_1, ce_2)$ {Counter example is the concatenation of $ce_1$ and $ce_2$}
14:   end if
15:   $cond'_{init} \leftarrow cond'_{init} \wedge (s_0, w_0) \neq GetFirstState(ce_1)$ {Initial state condition refinement}
16:   if $cond'_{init} = (\alpha, \beta)$ then
17:     return true {Property is true within the bound}
18:   end if
19: end loop
```



Property: $G(out < 28)$

Figure 3.5: Toy FSMD example and property

$(\sigma(in)) = (\{0, \cdots, 7\})$. $x \in V$ is a data register where $V_V = (\sigma(x)) = (\{0, \cdots, 31\})$ and $\beta = (0)$. $out \in O$ is an output where $O_V = \sigma(out) = (\{0, \cdots, 31\})$. The property is written in LTL, and it means $out < 28$ is globally $true$.

The shortest counter example from the initial state is as follows.

$$((s_a, s_b, s_c, s_e, s_f) \in S^5, (7, 7, 7, 7, *) \in {I_V}^5,$$
$$(0, 7, 14, 21, 28) \in {V_V}^5, (0, 0, 0, 0, 28) \in {O_V}^5)$$

where $*$ represents don't-care. As shown in the counter example, the property is violated only when 7 is input at the first four cycles. To find such a counter example with basic bounded model checking, the bound must be at least 4.

Let's consider to apply the proposed method with $k_1 = 2$ and $k_2 = 2$.

As the begging of the first step, Formula 3.11 is checked with $k = 0$. A counter example whose length is 0, such as $((s_f), (0), (28), (28))$, is generated. Then k is incremented and checked again with $k = 1$ and also $k = 2$. Finally, a counter example whose length is 2 is generated. Here, assume that the following counter example is generated.

$$((s_f, s_f, s_f), (0, 0, 0), (28, 28, 28), (28, 28, 28))$$

In the second step, the reachability from the initial state to the first state of the counter example, $(s_f, 28) \in S \times V_V$, is checked with Formula 3.12. However, since the minimum bounds required to reach s_5 is 4, the second step fails with the bound 2. Then, the state $(s_f, 28)$ is excluded from the initial state condition. Then the initial state condition $Cond'_{Init}$ becomes as follows.

$$Cond'_{init} = (s_0 \neq s_f \wedge v_0 \neq 28) \tag{3.14}$$

Then the first step is applied with the refined formula shown in Formula 3.13. Similarly to the previous trial, k is incremented to $k_1 = 2$, and assume that the following counter example is generated.

$$((s_c, s_e, s_f), (7, 7, 0), (14, 21, 28), (0, 0, 28))$$

In the second step, the reachability from the initial state to the state $(s_c, 14)$ is checked with Formula 3.12. In this case, there is a counter example from the initial state to the state $(s_c, 14)$ within length 2, such as

$$((s_a, s_b, s_c), (7, 7, 7), (0, 7, 14), (0, 0, 0))$$

Then a complete counter example can be obtained by concatenating the two counter example as follows.

$$((s_a, s_b, s_c, s_e, s_f), (7, 7, 7, 7, 0), (0, 7, 14, 21, 28), (0, 0, 0, 0, 28))$$

This counter example is included in the shortest counter example shown at the beginning of this paragraph.

Here, the number of maximum refinements of this example is counted to show that the inefficiency of this method. Possible first states for each counter example state transition path are as follows.

- $(s_0, s_1, s_2) = (s_f, s_f, s_f)$: Possible v_0 is $28, 29, 30, 31$.
- $(s_0, s_1, s_2) = (s_c, s_e, s_f)$: Possible v_0 is $21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31$.
- $(s_0, s_1, s_2) = (s_c, s_d, s_f)$: Possible v_0 is $22, 23, 24, 25, 26, 27, 28, 29, 30, 31$.

Since the first state of the second and third paths are same, overlapping v_0 value represents the same state. Then the number of possible first states are $4 + 11 = 15$. Then the maximum number of refinements is $15 - 1 = 14$.

3.4 Two Level Bounded Model Checking

To refine the method proposed in Section 3.3, symbolic technique is applied. The main idea of this method is to gather multiple counter examples at the first bounded model checking step instead of just one counter example. Then the following advantages can be obtained.

- Since reachability to the set of multiple counter examples instead of a single counter example is checked at the second bounded model checking step, counter example paths from the initial state can be found easier in the second step.
- Because of the first item, shorter counter-examples may be found.
- Since multiple states instead of a single state are excluded from the initial state space per each refinement, the number of refinements become much smaller.

Figure 3.6 is the entire flow of the two level bounded checking proposed in this section. The difference from Figure 3.2 is the insertion of symbolic simulation step.

Figure 3.7 shows an image of the method in the state space. As already explained above, the main feature is to handle multiple counter examples at once. As shown in Figure 3.7(a), the second step reachability analysis becomes much easier since it checks reachability to multiple states instead of a single state. Also multiple states can be removed at the initial condition update at once when the second step reachability analysis fails.

3.4.1 Generation of Condition to Enter Counter Example Path and Violate Property

As explained in Section 2.4, symbolic simulation is a technique that generates a condition satisfied when passing through a given control path. In the proposed method, symbolic simulation is applied to the control path of a counter example generated at the first step of the method explained in Section 3.3. From the symbolic simulation result and the property condition, the condition which is possible to pass through the control path same as the counter example and violate the property is

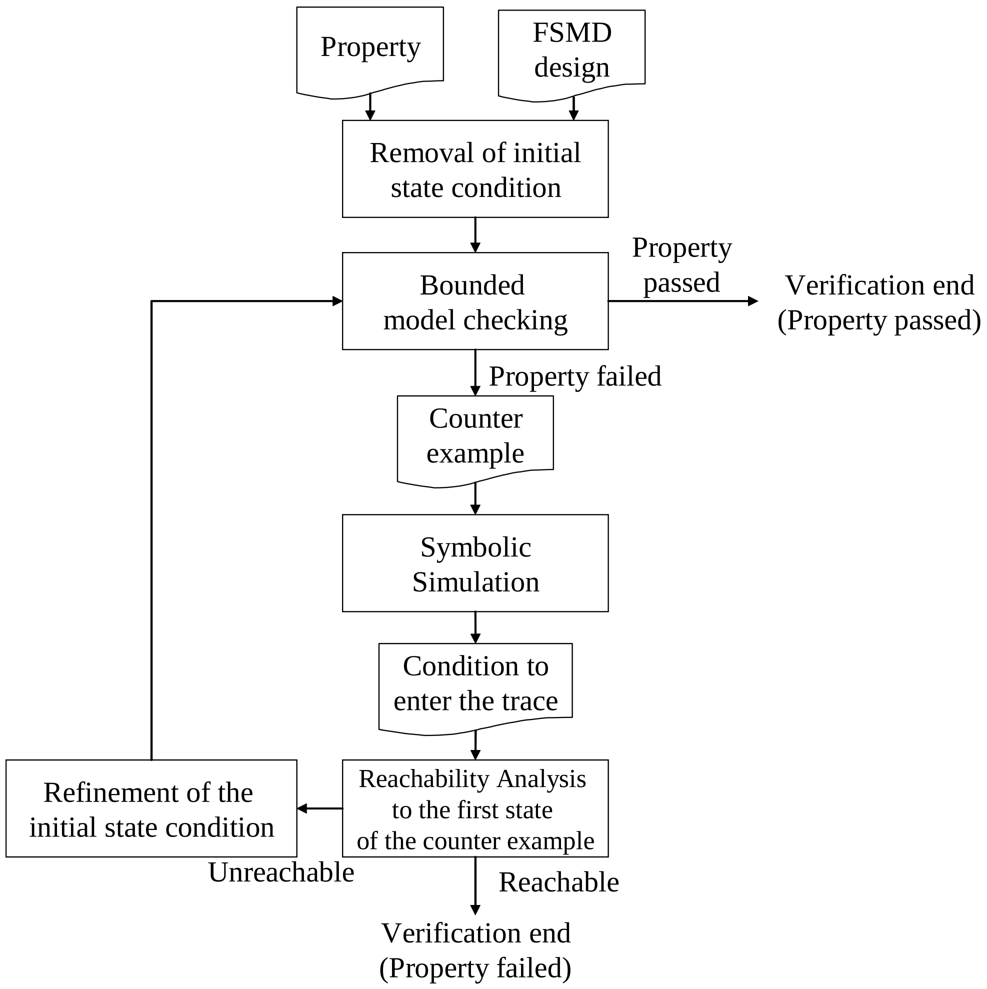

Property
FSMD design
Removal of initial state condition
Bounded model checking
Property passed
Verification end (Property passed)
Property failed
Counter example
Symbolic Simulation
Condition to enter the trace
Reachability Analysis to the first state of the counter example
Refinement of the initial state condition
Unreachable
Reachable
Verification end (Property failed)

Figure 3.6: Two level bounded model checking flow

generated. The condition corresponds to the set of multiple counter examples since the condition includes all counter examples pass through the control path. The advantage of this generation method is that it can generate such a set of multiple counter examples quite easily.

Symbolic simulation of a control path generates two conditions such as data condition and control condition as explained in Section 2.4. Let $Cond'_{prop} \in E$ denote a property represented by symbols. Then, when

$$CE_{FSMD1} = ((s^1_0, \cdot\cdot, s^1_{k_1}), (i^1_0, \cdot\cdot, i^1_{k_1}), (v^1_0, \cdot\cdot, v^1_{k_1}), (o^1_0, \cdot\cdot, o^1_{k_1}))$$

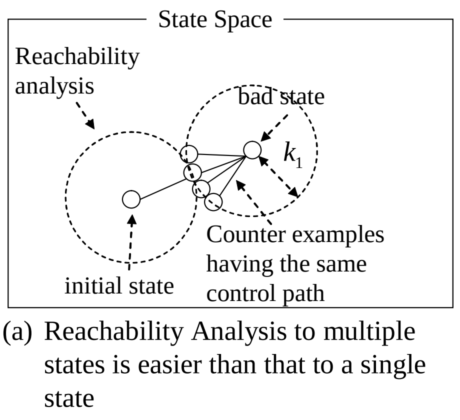

State Space
Reachability analysis
bad state
k_1
Counter examples having the same control path
initial state
(a) Reachability Analysis to multiple states is easier than that to a single state
State Space
Multiple states can be excluded at once
bad state
k_2
k_1
Counter examples having the same control path
initial state
(b) Multiple states can be excluded from the Initial state condition when the second step fails

Figure 3.7: State space image of two level bounded model checking

is the counter example generated by the first step of the method presented in Section 2.4, the necessary condition to pass through the control path same as the counter example and violate the property is as follows.

$$(s = s_0^1) \wedge N_C(t^1) \wedge \neg Cond'_{prop} \tag{3.15}$$

where s represents the current state, $t^1 = (s_0^1, \cdot\cdot, s_{k_1}^1)$. Let $Cond_{path}$ denote this condition. The first term represents the condition of the first state of the counter example, the second term expresses the control condition to pass through the counter example path, and the last term shows the condition to violate the property.

3.4.2 Replacement of Timed Symbolic Values

Although $Cond_{path}$ includes timed symbolic values at the cycles from 0 to k^1, only data register values at cycle 0 is required since the second step checks the reachability to the first states of the counter example set, and such first states are determined only with the control state and data register values at cycle 0. Then, timed symbolic values included in $I_0, I_1, \cdots, I_{k_1}$, $V_1, V_2, \cdots, V_{k_1}$, $O_0, O_1, \cdots, O_{k_1}$ are not required, and only timed symbolic values in V_0 are necessary. The timed symbolic values not required are replaced or removed from $Cond_{path}$ by the following procedure.

To remove the symbolic values in $V_1, V_2, \cdots, V_{k_1}$, and $O_0, O_1, \cdots, O_{k_1}$, the symbolic simulation result such that $N_D(t^1)$ can be used. As shown in Equation 2.2, $N_D(t^1)$

is a conjunction of multiple equations derived from assignments. Right hand sides of the equations are symbolic values at previous cycles of the left hand side symbolic values. Then timed symbolic values in $Cond_{path}$ which are included in the left hand side of the equations are replaced with the right hand side symbolic values. Since the left hand sides cover all timed symbolic values in $V_1, V_2, \cdots, V_{k_1}$, and $O_0, O_1, \cdots, O_{k_1}$, all of those symbolic values can be removed by applying such replacements incrementally. Here, that conversion is represented by a function $Replace_D : E_T \rightarrow E_T$.

Another conversion is applied to remove $I_0, I_1, \cdots, I_{k_1}$. Those timed symbolic values are not necessary since conditions of them are for after entering the counter example paths. Those timed symbolic values can be considered to take arbitrary values when checking the reachability to the set of the first state of the counter example paths. Then, each $i_T \in I_T$ in $Cond_{path}$ can be removed by assigning adequate value which satisfies necessary condition to keep $Cond_{path}$ true. Expressions including i_T which can also take arbitrary values with the values of i_T can be replaced with adequate values which satisfies necessary condition to make $Cond_{path}$ true. For example when $i_3 < 5$ must be true if $Cond_{path}$ is true and $i_3 \in I_T$, then i_3 can be replaced with 2. Also, whole the expression can be replaced with $true$, since it can take both $true$ and $false$ value by assigning values less than 5 and equal to or more than 5, respectively. Here, that conversion is represented by a function $Replace_I : E_T \rightarrow E_T$

With applying the above two conversions, $Cond_{path}$ is expressed only with timed symbolic values in V_0, F_{callT}, K, and the condition of the control state such that $s = s_0$. Here, since no timed symbolic values are at cycles more than 0, all timed symbolic values can be treated as symbolic values. Therefore, the condition $Cond_{reach} = Replace_I(Replace_D(Cond_{path}))$ is the condition that reachability to it is checked at the second step.

Since this condition represents the set of the first states of counter examples which pass through the same control path with the first counter example and violate the property. Then, when no counter example is found in the second step, the refinement of initial state condition is as follows.

$$Cond'_{init} := Cond'_{init} \wedge \neg Cond_{reach}$$

Since this refinement removes multiple states from the initial state condition at once, the number of refinement times can be dramatically reduced.

3.4.3 Construction of Complete Counter Example

In the case that a counter example is found in the second step which reaches to the condition generated in the previous section, a complete counter example is generated from the initial state which violates the property. It cannot directly be accomplished by concatenating two counter examples since the second step checked the reachability to the set of counter examples instead of the single counter example generated in the first step. The last state of the second counter example can be different from the first state of the first counter example. A complete counter example is generated with the following procedure.

To determine a counter example which can be concatenated with the second counter example, an input sequence must be determined. An additional counter example can be generated by running simulation from the last state of the second counter example with the input sequence. Then, if there is a sufficient condition of inputs to violate the property, a counter example which violates the property can be generated with an input sequence satisfying the condition.

The input condition to violate the property is generated from $Cond_{path}$ shown in Formula 3.15. Let CE_{FSMD2} denote the counter example generated in the second step.

$$CE_{FSMD2} = ((s_0^2, \cdot\cdot, s_{k_2}^2), (i_0^2, \cdot\cdot, i_{k_2}^2), (v_0^2, \cdot\cdot, v_{k_2}^2), (o_0^2, \cdot\cdot, o_{k_2}^2))$$

Since the subsequent counter example must start from the last state of CE_{FSMD2}, s must be $s_{k_2}^2$ and values of V_0 must be $v_{k_2}^2$ in $Cond_{path}$. Then, the first term of $Cond_{path}$, $s = s_0$, becomes true since s_0 must be $s_{k_2}^2$, and it can be removed. Concrete values are also assigned to the timed symbolic values in V_0. Here, this conversion is represented by a function $Replace_0$.

$Replace_0(Cond_{path})$ includes timed symbolic values in $I_0, I_1, \cdots, I_{k_1}, V_1, V_2, \cdots, V_{k_1}$, and $O_0, O_1, \cdots, O_{k_1}$ some of which are removed with the conversions introduced in the previous section. Here, since the input condition is required, only $V_1, V_2, \cdots, V_{k_1}$ and $O_0, O_1, \cdots, O_{k_1}$ should be removed. This removal can be performed with applying the function $\mathit{Replace}_D$ which is introduced in the previous section. Then, $\mathit{Replace}_D(\mathit{Repalce}_0(\mathit{Cond}_{path}))$ is the condition expressed only with $I_0, I_1, \cdots, I_{k_1}$, and a sufficient condition to violate the property.

There can be multiple input sequences which satisfy $\mathit{Replace}_D(\mathit{Repalce}_0(\mathit{Cond}_{path}))$, and any of them is a counter example which violates the property and which can be concatenated with the second counter example CE_{FSMD2}. One of such input sequences can be found by generating assignments to timed input symbolic values in $\mathit{Replace}_D(\mathit{Repalce}_0(\mathit{Cond}_{path}))$ by decision procedure. For example, SMT(Satisfiability Modulo Theories) solvers such as CVC3[40] and Boolector[24] can be used. Then, a complete counter example is generated by concatenating the second counter example and the generated assignment by decision procedure.

3.4.4 Algorithm

By applying the symbolic simulation method, algorithm shown in Algorithm 3 is modified to Algorithm 4.

Modified portions from Algorithm 3 are Line 9, 10, 14, 17. Line 9 shows the method explained in Section 3.4.1. Line 10 and 17 represents the method explained in Section 3.4.2. Line 14 corresponds to the method explained in Section 3.4.3.

3.4.5 Example

In this section, the effect of the application of symbolic simulation introduced in this section is shown with the example in Figure 3.5.

Let's consider applying the proposed method with $k_1 = 2$ and $k_2 = 2$. Assume that the following counter example is generated by the first step which is the same as that of Section 3.3.4.

$$((s_f, s_f, s_f), (0, 0, 0), (28, 28, 28), (28, 28, 28))$$

The sequence of state transitions of this counter example is $t_1 = (s_f, s_f, s_f)$, and the symbolic simulation result on the path is shown in Figure 3.8.

Firstly, the condition to pass through t_1 and violates the property is generated as explained in Section 3.4.1 as follows.

$$\mathit{Cond}_{path} : (s = s_f) \wedge \mathit{true} \wedge \neg(\mathit{out}_0 < 28 \wedge \mathit{out}_1 < 28 \wedge \mathit{out}_2 < 28)$$

Algorithm 4 Two level bounded model checking

```
Require: $cond_{prop}$, $k_1 \geq 0$ {bound of the first step}, $k_2 \geq 0$ {bound for the second step}
 1: $cond'_{init} \leftarrow true$ {Initialization of initial state condition}
 2: loop
 3:   $ce_1 \leftarrow FirstStep(cond_{prop}, cond'_{init}, k_1)$ {Algorithm in Figure 1 is applied}
 4:   if $ce_1 = NULL$ then
 5:     return true {Property is true within the bound}
 6:   else if $GetFirstState(ce_1) = (\alpha, \beta)$ then
 7:     return $ce_1$ {Counter example is generated without the second step}
 8:   else
 9:     $cond_{path} \leftarrow GetControlCondition(ce_1) \wedge \neg cond_{prop}$ {Get the condition to pass
        through the control path of the counter example and violate the property}
10:     $cond_{reach} \leftarrow Replace_I(Replace_D(cond_{path}))$ {Remove timed symbolic variables
        not necessary}
11:     $ce_2 \leftarrow SecondStep(cond_{reach}, k_2)$ {Algorithm in Figure 2 is applied }
12:   end if
13:   if $ce_2 \neq NULL$ then
14:     $ce'_1 \leftarrow Solve(Replace_D(Replace_O(cond_{path}, ce_2)))$ {Get the input sequence which
        violates the property and can be concatenated with $ce_2$}
15:     return $Concatenate(ce'_1, ce_2)$
16:   end if
17:   $cond'_{init} \leftarrow cond'_{init} \wedge \neg cond_{reach}$ {Refine the initial state condition}
18:   if $cond'_{init} = (\alpha, \beta)$ then
19:     return true {Property is true within the bound}
20:   end if
21: end loop
```

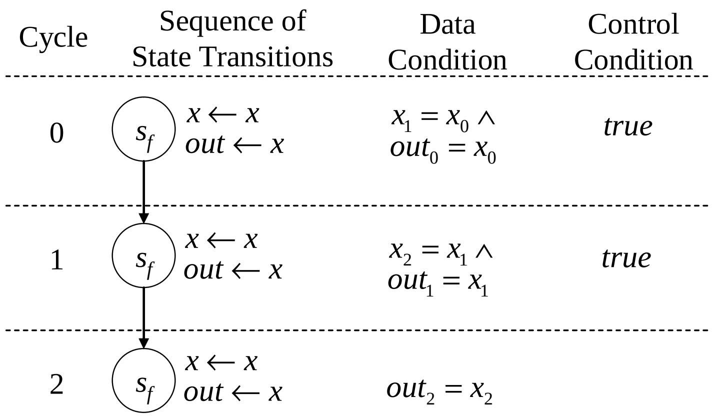


Figure 3.8: Symbolic simulation result on the path $s_f \rightarrow s_f \rightarrow s_f$ at the design in Figure 3.5

where out_j represents timed symbolic values of out at cycle j. The control condition of the sequence of state transitions, $N_C(t_1)$, is $true$, since there are no conditional branches. Property condition is simply the negation of the property $G(out < 28)$ extracted for all cycles.

Secondly, timed symbolic variables out_0, out_1, out_2 are removed from $Cond_{path}$ with the data condition shown in Figure 3.8 as explained in Section 3.4.2. Data condition $N_D(t_1)$ is as follows.

$$(x_1 = x_0 \wedge out_0 = x_0) \wedge (x_2 = x_1 \wedge out_1 = x_1) \wedge (out_2 = x_2)$$

With assuming this condition $true$, $out_0, out1, out_2$ in $Cond_{path}$ is transformed as follows.

$$out_2 = x_2 = x_1 = x_0$$

$$out_1 = x_1 = x_0$$

$$out_0 = x_0$$

Then $Cond_{path}$ is transformed as follows.

$$(s = s_f) \wedge \neg(x_0 < 28)$$

Then, reachability is checked to the above condition in the second step. However, no counter examples from the initial state to the condition exist. The condition is

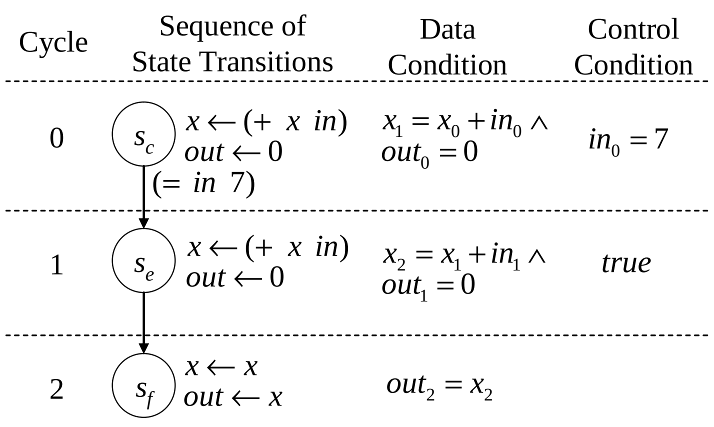


Figure 3.9: Symbolic simulation result on the path $s_c \rightarrow s_e \rightarrow s_f$ at the design in Figure 3.5

added to the initial state condition as follows.

$$Cond'_{init} = (s_0 \neq s_f \wedge \neg(v_0 = 28 \vee v_0 = 29 \vee v_0 = 30 \vee v_0 = 31))$$

Here, the condition with symbolic values has been converted into the condition with concrete values. Though Condition 3.14 excludes only one state, the above condition excludes 4 states,

In the second trial, assume that the first step generates the following counter example.

$$((s_c, s_e, s_f), (7, 7, 0), (14, 21, 28), (0, 0, 28))$$

Here, $t'_1 = (s_c, s_e, s_f)$ denotes the sequence of state transitions of the counter example, and the symbolic simulation result on the path is shown in Figure 3.9.

Since $N_C(t'_1)$ is $in_0 = 7$, the condition to transit through t'_1 is as follows.

$$Cond'_{path} = (s = s_c) \wedge (in_0 = 7) \wedge \neg(out_0 < 28 \wedge out_1 < 28 \wedge out_2 < 28)$$

Data condition $N_D(t'_1)$ is as follows.

$$(x_1 = x_0 + in_0 \wedge out_0 = 0) \wedge (x_2 = x_1 + in_1 \wedge out_1 = 0) \wedge (out_2 = x_2)$$

With assuming $N_D(t'_1) = true$, out_0, out_1, and out_2 can be transformed as follows.

$$out_2 = x_2 = x_1 + in1 = x_0 + in_0 + in_1$$

$$out_1 = 0$$
$$out_0 = 0$$

Then, $Cond'_{path}$ is transformed as follows after meaningless portions are removed.

$$(s = s_c) \wedge (in_0 = 7) \wedge \neg(x_0 + in_0 + in_1 < 28)$$

Next, the timed symbolic input values are removed. To make the condition true, in_0 must be 7. To also make the last term true, $in_1 = 7$ is most sufficient. Then, $Cond'_{path}$ is updated as follows.

$$(s = s_c) \wedge \neg(x_0 < 14)$$

Since this condition includes more than one state such that $(s_c, 14), (s_c, 15), \cdot\cdot\cdot(s_c, 31)$, it is more general than just one state $(s_c, 14)$.

The second step reachability analysis is applied to the condition $Cond'_{path}$, and it generates the following counter example.

$$((s_a, s_b, s_c), (7, 7, 7), (0, 7, 14), (0, 0, 0))$$

To generate an additional counter example to the above one, $s = s_c$ and out_0, out_1, out_2 are removed from the original $Cond'_{path}$ with $N_D(t'_1)$, and the last state of the second counter example such that $(s_c, 14)$. Then, $Cond'_{path}$ is updated as follows.

$$Cond'_{path} = (in_0 = 7) \wedge \neg(14 + in_0 + in_1 < 28)$$

Decision procedures can generate assignments to in_0 and in_1 which makes the above condition true, such that $in_0 = 7$ and $in_1 = 7$. With this input sequence, the following complete counter example is generated.

$$((s_a, s_b, s_c, s_e, s_f), (7, 7, 7, 7, 0), (0, 7, 14, 21, 28), (0, 0, 0, 0, 28))$$

The maximum number of refinements is 2 since there are only 3 control paths which reach s_f. This is much smaller than 14 which is the maximum number of refinements by the method explained in Section 3.3.

3.5 Multi-Level Bounded Model Checking

In this section, multi-level (more than three level) bounded model checking is proposed. This method is an extension of the two-level bounded model checking proposed in the previous section.

To extend the two-level one into the multi-level one, The two-level bounded model checking is recursively applied in the reachability analysis step which has been introduced in Section 3.3.2. Since the reachability analysis is a bounded model checking, it can be performed with two-level method.

Since the extension can be recursively applied for any number of times, the method can be extended to any level.

3.5.1 Division of the Second Step

The second step which is introduced in Section 3.3.2 checks the reachability from the initial state to the first state of the first counter example (without symbolic simulation, Section 3.3.1) or a state in a set of first states of counter examples (with symbolic simulation, Section 3.4). In both cases, a condition $Cond_{reach}$ whose reachability is checked is generated. Then, the bounded model checking formula whose satisfiability is checked is as shown in Formula 3.12.

By comparing Formula 3.12 with Formula 3.3, it is clear that the second step is also a bounded model checking, and its property corresponds to $\neg Cond_{reach}$. Therefore, if $Cond_{prop}$ is considered to be $\neg Cond_{reach}$ and the two-level bounded model checking is applied, a single bounded model checking is divided to three, and it can be called three-level bounded model checking. Further division can be recursively applied, and a single bounded model checking can be divided into an arbitrary number of bounded model checkings, as the state space image shown in Figure 3.10.

The algorithm of the method is shown in Algorithm 5. A function $MultiLevel$ represents this algorithm itself, so that this is a recursive algorithm. Only the difference from the two level method shown in Algorithm 4 is line 11 $\sim$ 15 where the second step reachability analysis is performed by the multi-level method recursively.

Algorithm 5 $MultiLevel$:Multi-level bounded model checking

```
Require: $cond_{prop}$, $n$ {level}, $k_1..k_n \geq 0$ {bounds}
1: $cond'_{init} \leftarrow true$ {Initialization of initial state condition}
2: loop
3:    $ce_1 \leftarrow FirstStep(cond_{prop}, cond'_{init}, k_n)$
4:    if $ce_1 = NULL$ then
5:       return $NULL$ {Property is true within the bound}
6:    else if $GetFirstState(ce_1) = (\alpha, \beta)$ then
7:       return $ce_1$ {Property is violated with $ce_1$}
8:    else
9:       $cond_{path} \leftarrow GetControlCondition(ce_1) \wedge \neg cond_{prop}$
10:      $cond_{reach} \leftarrow Replace_I(Replace_D(cond_{path}))$
11:      if $n > 2$ then
12:         $ce_2 \leftarrow MultiLevel(\neg cond_{reach}, n-1, k_1..k_{n-1})$ {recursive call}
13:      else
14:         $ce_2 \leftarrow SecondStep(cond_{reach}, k_{n-1})$ {Algorithm in Figure 2 is applied}
15:      end if
16:   end if
17:   if $ce_2 \neq NULL$ then
18:      $ce'_1 \leftarrow Solve(Replace_D(Replace_O(cond_{path}, ce_2)))$
19:      return $Concatenate(ce'_1, ce_2)$
20:   end if
21:   $cond'_{init} \leftarrow cond'_{init} \wedge (s_0, w_0) \neq GetFirstState(ce_1)$
22:   if $cond'_{init} = true$ then
23:      return NULL
24:   end if
25: end loop
```

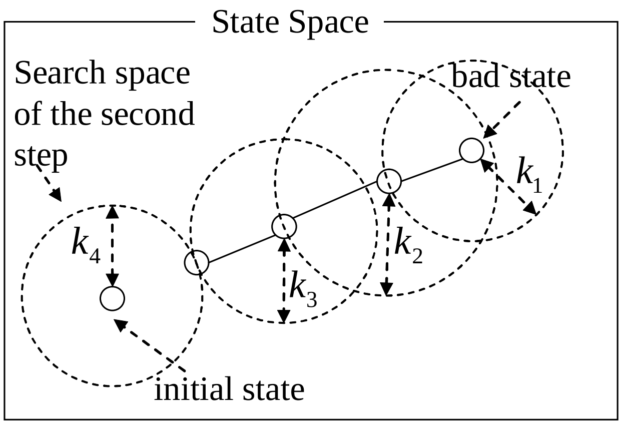


Figure 3.10: State space image of the multi-level bounded model checking (four level)

3.5.2 Deciding Levels and Bounds

When Algorithm 5 is applied, number of levels and bound of each level must be specified. Both of them increase the complexity exponentially when they increase. It is very difficult to find the best specification of them since it depends on the distances between the initial state and bad states. However, one possible strategy to specify them is shown as follows.

Verification time of a bounded model checking is greatly depends on the bound. There is a threshold of the bound that user can bear the verification time which depends on the user. Here, the bound is denoted by k_{max}, and used for each level in multi-level bounded model checking to make the verification time of each level acceptable and keep the search space large. Once the bounds are fixed, the multi-level method can be applied by incrementing the number of levels from two until a bad state is reached (the property is violated). The algorithm of this strategy is shown in Algorithm 6.

3.5.3 Combination with Simulation

In this chapter, reachability from the initial state is checked number of times. However, the same reachability can also be checked from some reachable states whose

Algorithm 6 Fixed bound approach on multi-level bounded model checking

```
Require: $cond_{prop}$, $k_{max}$ {Maximum bound that user can bear}
1: for $l \leftarrow 2$ to $\infty$ do
2:     $ce \leftarrow MultiLevel(cond_{prop}, l, k_{max}, k_{max}, ...)$
3:     if $ce \neq NULL$ then
4:         return $ce$ {Property is violated with $ce$}
5:     end if
6: end for
```

paths from the initial state are already known. Then, results of (random) simulations can be used by gathering all reached states during the simulations and regarding them as the initial states. The advantages of this method are as follows.

- Total verification time is not affected much by applying simulation since simulation finishes in much shorter time than model checking.
- Relatively far bad states can be reached since simulations can reach to states far from the initial states, and multi-level bounded model checking can be started from such states.
- Possibility to prove reachability becomes higher since this method checks reachability to multiple initial states instead of the original single state.

3.6 Experimental Results

To confirm the effectiveness of the proposed method, the experiments for two examples were conducted with the two-level approach proposed in Section 3.4. NuSMV[29] is used as a bounded model checker for both the bounded model checking and the reachability analysis. Symbolic simulations were performed with an original tool written in C++. NuSMV was also used for the comparative experiments with the standard way of bounded model checking from the initial states. These experiments were carried out on a Linux PC with 3.0GHz Core2Duo processor and 4GB memory.

3.6.1 Examples

The experiments were conducted for two examples, Double Counter and Vending Machine Controller. Both examples are designed in FSMD, and the sizes of the

Table 3.1: Size of the examples

	Num. of FSMD states	Num. of state variables
Double Counter	2	14
Vending Machine	12	33

Table 3.2: Verified properties

Example	Property	Contents
Double Counter	p1	Second counter never overflows
Vending Machine	p2	If no input is given, it eventually goes to the state “Waiting inputs”

examples are shown in Table 3.1.

3.6.2 Results

The property for each example is shown in Table 3.2. Both of them are unbounded properties. Each example has a bug that violates the property.

Verification time spent in each property by the proposed method is shown in Table 3.3. The column “BMC” shows the verification times of the bounded model checking performed in the first step. The bounds of this step are shown in the seventh column. The column “SS” shows the run times of symbolic simulation. The column “RA” shows the verification times of the reachability analysis from the initial state, and the bounds of this step are shown in the eighth column. The column “Sum” shows the total verification times of the proposed method. The column “Num of Trial” shows the numbers of trials of the reachability analyses from the initial state. The column “Simple BMC” shows the verification times of the original bounded model checking from the initial state. In the original bounded model checking, bounds were set to the sum of the seventh and eighth columns. All bugs were detected with both methods.

In the results, the proposed method could find the bugs in shorter time than the simple method in both cases. Two reasons can be found to explain that the improvement in the result of p_1 was better than that of p_2. One reason is that

Table 3.3: Results of the proposed method and simple bounded model checking

	BMC	SS	RA	Sum	Num of Trial	Bound of BMC	Bound of RA	Simple BMC
p1	52s	1s	27s	80s	99	101 cycles	100 cycles	303s
p2	2s	1s	201s	204s	1	9 cycles	24 cycles	341s

the number of the total bound in the verification of p_1 is larger than that of p_2 so that the effect of division was more strong. The other reason is that the bounds of the initial bounded model checking and the reachability analysis of p_1 are same. Equivalent bounds assignment is preferred since a step with larger bound is mostly dominant. This result supports the Algorithm 6 where the bounds are equal in all stages.

3.7 Conclusion

This chapter presented a method to extend the verification bounds of bounded model checking. It was achieved by the decompositions of a single bounded model checking and the analysis of counter-examples by symbolic simulation. Improvement of the verification performance was confirmed in the experimental results.

One problem in the multi-level method is that this method can generate a complete counter example only when all intermediate points are reachable from the initial states. Then, the number of refinement times increases dramatically with the number of levels since number of verification failures increases. Therefore, it is very important to select good intermediate states. This problem, finding good intermediate points, is also a common problem in the general state space reachability problem.

Chapter 4

Equivalence Checking with Synthesizing Designs onto Identical Datapath

4.1 Introduction

As mentioned in Section 2.1.2, equivalence checking is a powerful method to guarantee no bugs are inserted during a refinement or synthesis step.

One of the simplest equivalence checking methods is proposed in [33]. It translates designs into Boolean formulae, and checks the equivalence of those formulas with BDD or SAT. However, large designs cannot be verified, since the complexity of such a bit-level analysis increases exponentially with the size of designs.

To avoid bit-level analysis as much as possible, word-level symbolic simulation[143, 113, 90, 91] which treats each variable or operator as a symbol is applied. However, since the complexity of symbolic simulation is doubled for each conditional branch, it is still not applicable to entire designs. Loops are also not acceptable, and they must be unrolled in advance.

To solve this problem, only textually different portions of two designs are compared in [113]. This method can handle large designs when compared designs are similar. Also in [90, 91], equivalences of paths between conditional branches are checked, and the results are gathered to prove the entire equivalence. To apply this divide-and-conquer approach, correspondences of intermediate variables or registers between two designs must be known or given by users (e.g. Names of variables or registers in two designs are same).

However, in practical refinement steps, it is usual that the entire structure of a design is changed or correspondences of intermediate variables or registers are unknown (e.g. between two designs before and after automated high-level synthesis). Since bit-width or sign are not taken into account in symbolic simulation, bit-level accuracy is not considered in [143, 113, 90].

Based on the arguments above, in this chapter, a new equivalence checking method between two models before and after a refinement step, such as high-level synthesis or behavioral optimization, is proposed. This method focuses on a feature that designs after automated high-level synthesis are usually composed of controllers and datapaths. In such a design, computations of the design are executed at the datapath, and the controller determines computations executed at each clock cycle. In the proposed method, two designs are converted into RTL models which have same datapaths. Because of the advantages of abstracting computations of the datapaths, the verification can concentrate on the equivalence of the controllers. Concretely, since the datapaths are identical, the functional units in those RTL models become same. Since same control signals from the controllers represent same behaviors, the behaviors are equivalent in bit-level accuracy. Therefore, existing word-level methods, such as symbolic simulation, can be easily applied in bit-level accuracy.

However, since correspondences of intermediate variables or registers are not given in most cases, entire designs must be compared in such cases. As discussed above, symbolic simulation cannot handle designs which include large numbers of conditional branches or loops whose numbers of iterations are dependent to input values or infinite. Therefore, a new word-level method which propagates equivalences of inputs to those of outputs with pre-defined rules is proposed. Since the rules are proposed to handle conditional branches and loops, the proposed rule-based method can be used as a complement of symbolic simulation based methods.

The remainder of this chapter is organized as follows: Section 4.2 explains existing techniques used in the proposed method. Section 4.3 describes the proposed verification flows. In Section 4.4, the proposed verification algorithms used in the flows are explained. Some experimental results with realistic examples are reported in Section 4.5. In Section 4.6, a conclusion of this work is given and its future directions are shown.

4.2 Basic Notions

4.2.1 Separation of Designs' Equivalence to that of Controllers and Datapaths

The basic idea of the proposed method is proposed in [50, 48, 49]. An RTL design generated by high-level synthesis is usually composed of a controller and a datapath as shown in Figure 2.9. At each clock cycle, first, the controller sends control signals to the datapath, depending on the current state. Next, the datapath executes operations based on the control signals. Finally, the datapath returns status signals to the controller, and the controller determines the next state. Then, in [50, 48, 49], a behavioral design is mapped to a virtual controller and a virtual datapath so that

```
/* Behavior descriptions */
Input In;
Output A, B;
Variable X, Y;
X = Get(In);
Y = Get(In);
A = 0;
B = X;
While(B >= Y){
  B = B - Y;
  A = A + 1;
}
```

Figure 4.1: An example of behavioral description [50]

the equivalence can be separated into the equivalences of the (virtual) datapaths and the (virtual) controllers.

Here an example of this mapping is shown. Figure 4.1 is a behavioral description before high-level synthesis, and Figure 4.2 is an RTL description generated by high-level synthesis. These descriptions are written in a simple form to be able to understand intuitively. These descriptions are designs which execute division. The value assigned to X is divided by the value assigned to Y. The values of variables A and B after the loops show the quotient and the residual, respectively. The description in Figure 4.2 is scheduled and each **Waitfor(CL)** in the description represents that the clock proceeds for one cycle.

Figure 4.3 shows the structure of the RTL design in Figure 4.2. Since it is a design after high-level synthesis, it is composed of a controller and a datapath. The controller is a control FSM. It decides the next state with the received status signal ($dc1$), and sends the control signals ($cd1 - cd6$) correspond to a current state. The datapath is composed of data registers (x, y, a, b), multiplexers ($mux1 - mux4$), a computation unit which increments its input ($inc1$), a subtracter ($sub1$), and a comparator ($comp1$). The datapath performs different computations which depend on the value of control signals ($cd1 - cd6$), updates register values, and returns a

```
/* Scheduled (RTL) descriptions*/
Input In;
Output A, B;
Clock CL;
Variable X, Y;
Waitfor(CL);
X = Get(In);
Waitfor(CL);
Y = Get(In);
Waitfor(CL);
A = 0;
B = X;
Waitfor(CL);
While(B >= Y){
  B = B - Y;
  A = A + 1;
  Waitfor(CL);
}
```

Figure 4.2: An example of RTL description [50]

status signal ($dc1$) to the controller.

In [50], the design in Figure 4.1 is tentatively scheduled as Figure 4.2 so that the design in Figure 4.1 can be virtually mapped to the controller and the datapath shown in Figure 4.3. In this case, the mapped controller and datapath become same as that of the synthesized RTL description. Since a number of each computation units and scheduling are not decided in behavioral design, it can be mapped to the same datapath as that of the compared RTL design by applying adequate scheduling and allocation of computation units.

When the two datapaths are same, only the two controllers have to be compared. On the comparison, only the control signals from those controllers are compared without understanding the behaviors of the datapaths. Large designs can be handled by this method since the computation units and the registers in the datapath is not taken into account.

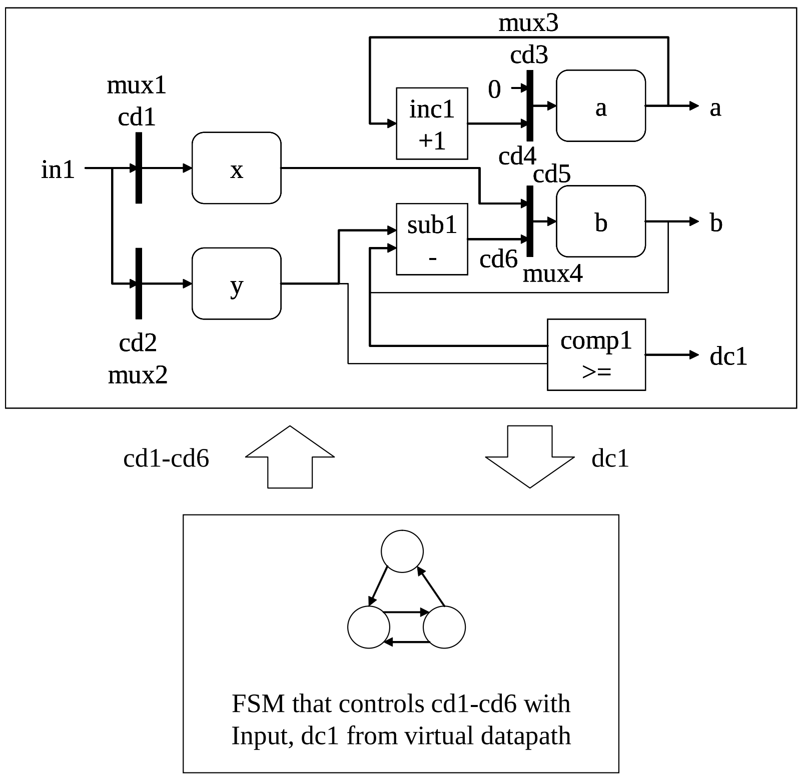


Figure 4.3: An example of controller and datapath [50]

On the other hand, when the two datapaths are not same, both of the controllers and the datapaths must be compared. The equivalence of the datapath operations must be checked under each pair of the control signals which is a candidate to be equivalent. This step might be time consuming since usually the correspondences of the control signals nor the status signals between the two datapaths are not known. If the two datapaths are same, the datapaths do not have to be compared. In addition, two controllers generated from equivalent designs can be similar since they are for the identical datapath. Then, an equivalence checking method based on the difference of controllers which is similar to [113] can be applied so that large designs can be verified.

The proposed method extends this approach, by forcibly making the datapaths of two designs same by generating controller(s) for an identical datapath which are equivalent to the original design(s). This method is described in Section 4.3. Since

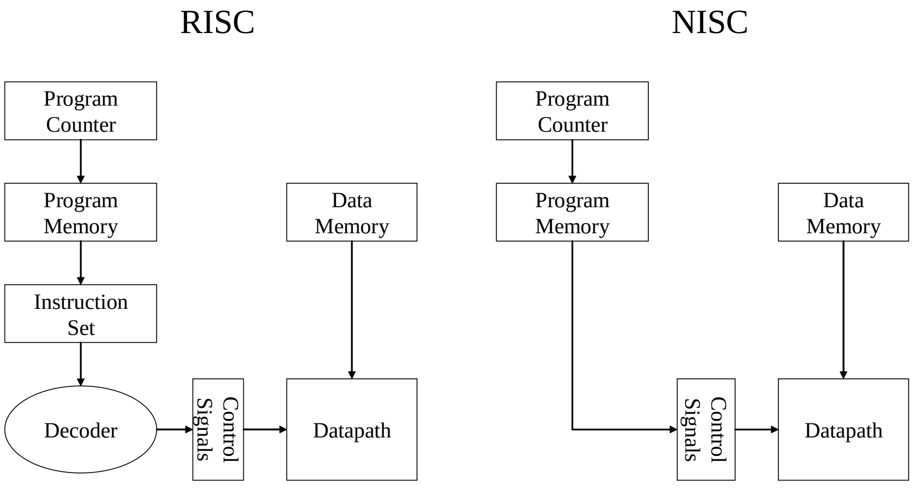


Figure 4.4: Comparison between RISC and NISC architectures [136]

only a brief approach to compare the controllers is shown in [50, 48, 49], a concrete method is also given in Section 4.4.

4.2.2 NISC(No Instruction Set Computer) Compiler

NISC[136] is a computer architecture which is composed of an arbitrary datapath and its controller. Different from the other computer architectures, NISC has no instruction sets, and a set of control signals is directly stored in a control memory instead of a set of instructions as shown in Figure 4.4. Those control signals are called "control words", and they include not only the signals which control the operations of the datapath but also the next values of the program counter. This structure enables designers to use an arbitrary datapath, since an instruction set for it does not have to be newly defined. Designers can give suitable datapaths for their requirements by specifying their structures (i.e. numbers of various computation units, registers, data memories, bus-widths, and their connections). NISC compiler can generate a set of control words for any given datapath from an ANSI-C code, if the datapath has sufficient resources to execute the code. Thus, NISC architecture can achieve both the high-performance of custom hardware and the flexibility of software.

The aim of NISC compiler is same as that of the proposed method in the point that it generates a control portion for a given datapath. In NISC compiler, this process is performed by the following steps. First, a Data Flow Graph (DFG) is created from an input ANSI-C code. Next, the DFG is traversed backwardly from the outputs, and each operation is assigned to a functional unit at a cycle in the datapath with an ALAP like scheduling. Multiple operations can be mapped to a single cycle while resources (functional units and lines of buses) are enough. At this step, delays of the functional units are considered. This avoids creating long paths of the functional units for a single cycle. Finally, control words to be stored in the control memory are generated. The control words include:

- Signals to the multiplexers in the datapath which correspond to the values of the program counter
- Next values of the program counter which can be considered as next states
- Constants used in the operations at the datapath

The above method is quite simple and reasonable. Since this chapter is focusing on verification and does not have to consider the performance, a similar (or simpler) solution can be used. The method is discussed in Section 4.3.2.

4.3 Generation of RTL Designs with Identical Datapath

4.3.1 Verification Flow

Based on the argument in Section 4.2.1, in the proposed method, the datapaths of two designs are forcibly made identical. If they are identical, the following advantages can be obtained.

- Same control signals represent that behaviors are equivalent in bit-level accuracy
- Controllers generated from equivalent designs tend to be similar since they are generated for an identical datapath.

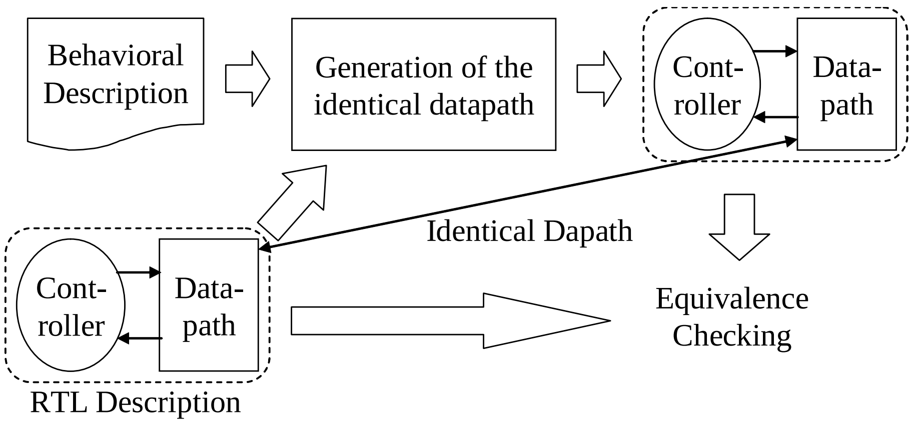


Figure 4.5: Proposed equivalence checking flow between a behavioral level design and an RTL design

Figure 4.5 shows the verification flow to check the equivalence between designs before and after high-level synthesis. One design is a behavioral design and the other is an RTL design. It is assumed that the RTL design is composed of a controller and a datapath, and easily separated into them. As mentioned in Section 4.2.1, results of high-level synthesis usually satisfy the assumption. If the assumption is not satisfied, they have to be separated by determining its state variables. Next, a controller for the datapath in the RTL design is generated from the behavioral design. The generated controller is written in RTL, and it must represent the same behavior as the behavioral design. Details of this step are described in Section 4.3.2. Then, two controllers for an identical datapath can be obtained. Comparison methods for those designs are described in Section 4.4.

A similar method can be also applied to check the equivalence between two designs before and after behavioral optimization, and its flow is shown in Figure 4.6. Input designs are both in behavioral level. The difference from the previous flow is that a new datapath has to be given to generate controllers for the datapath since neither of the input designs are RTL. The datapath should be as simple as possible since same arithmetic operations in the designs should be executed by a same set of functional units in the datapath. If same operations are executed by different sets of functional units, the equivalence between those sets must be checked

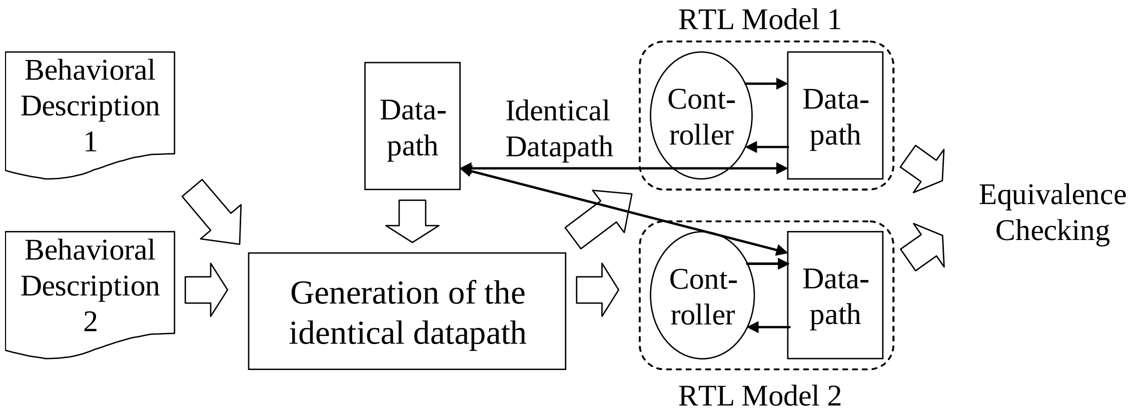


Figure 4.6: Proposed equivalence checking flow between two behavioral level designs

in bit-level.

4.3.2 Generation of a Controller for a Given Datapath

As mentioned in Section 4.2.2, a similar method to NISC compiler can be used to generate a controller for a given datapath. With the following limitations, the scheduling method of NISC compiler can be directly applied.

- Buses can only be used to transmit inputs, outputs, and status signals to the controller
- Use of datapath memories is prohibited (since it cannot be represented by FSMD)
- Delay of functional units can be neglected (since we do not have to consider the performance)

As the next step, each of generated control words is divided into signals to multiplexers and next values of the program counter. The signals to the multiplexers correspond to the control signals in Figure 3, and the next values of the program counter corresponds to the next states of the controller. Therefore, an RTL controller without control memory can be easily generated from them.

However, the above method fails in the following cases.

- Optimizations with limitations of input values, such as bit-width reduction and table-lookup division, are applied
- Precisions of variables or operations are different in two designs, such as floating value and fixed-point value.
- Operations of corresponding computations are different in two designs, such as constant multiplication, and bit-shift with addition

Since two designs are not logically equivalent in the first two cases, the proposed method cannot handle them.

For the last case, the controller generation method can be extended by giving information about equivalences of operations. When such information is appended, the correctness in bit-level must be guaranteed since it affects the accuracy of equivalence. The correctness can be checked with decision procedure (SMT solver), such as CVC3[40]. However, this solution is difficult to be applied when an external tool such as NISC compiler is used since the information about equivalences of operations must be given internally. For such a case, some circuits which perform the lacking computations to the datapath can be added, and a controller for the modified datapath is re-generated.

4.4 Equivalence Checking of RTL models which have same datapaths

4.4.1 Equivalence Checking in Bit-Level Accuracy

As shown in Figure 4.5 and Figure 4.6, inputs of the final step of the verification flows are RTL models which have identical datapaths. Since the datapaths are identical, any two same control signals represent a same behavior executed by a same set of functional units. Since operations executed by those control signals are equivalent in bit-level, word-level equivalence checking methods such as symbolic simulation can be applied with bit-level accuracy guaranteed.

However, this bit-level accuracy may be too limited to verify various designs. In such a case, equivalences among operations executed by different sets of functional units can be checked. Some candidates to be checked are listed below. All operations are written in Lisp-like style.

- Commutative law
  e.g. $(+\ a\ b) \equiv (+\ b\ a)$
- Scalar multiplication executed by addition $\times n$
  e.g. $(*\ a\ 2) \equiv (+\ a\ a)$
- Scalar multiplication executed by shift + addition
  e.g. $(*\ a\ 5) \equiv (+\ (<<\ a\ 2)\ a)$, where $<<$ represents a shifter-left operation.

Such equations can be checked by equivalence checkers for combinational circuits. Operations of a datapath are fixed with a given control signals, and the equivalence between circuit portions which are related to the operations corresponding to an equation can be checked. These portions must be combinational circuits. The details of this method is explained in Section 4.4.2.

As described in Section 4.3.2, equivalence of operations is considered in both the controller generation stage and this equivalence checking stage. If much equivalence is considered in one stage, the effort of the other stage is reduced. However this stage is required when there are multiple ways to perform an operation in a datapath (e.g. both a multiplier and a shifter exist in the datapath for constant multiplication) since the operation can be mapped differently in two designs with the method in Section 4.3.2. In such a case, verification with the method described in this section can be performed.

4.4.2 Equivalence Checking of Different Control Signals

As explained in Section 4.4.1, bit-level analysis is required to prove equivalences among different sets of control signals. Here, such equivalences are checked with existing combinational equivalence checking technique.

A control signal determines the behavior executed in a datapath at a single cycle. A behavior at a single cycle can be considered as signal transitions from the outputs of data registers to the inputs of the registers. No other registers exist between the outputs and the inputs, and the circuit portion between them is a combinational circuit. Then, comparison between computations of two control signals can be performed by combinational equivalence checking techniques. Though such combinational equivalence checkings are performed in bit-level accuracy, state-of-the-art combination equivalence checkers can handle large circuits more than million gates. Therefore, this method can also handle large datapath circuits. Even in a case that a single control signal corresponds to a sequence of control signals, this method can be applied by unrolling the circuit for the number of control signals in the sequence.

This method is composed of the following four steps.

1. The datapath circuit is unrolled for the length of each control signal sequence.
2. The multiplexer connections in the unrolled circuits are fixed with the control signals, and portions where signals are not transmitted are removed.
3. Registers are removed from the circuits to be combinational circuits.
4. The equivalences between those unrolled circuits are checked by existing combinational equivalence checkers.

Since this method based on signal transitions among registers which is the basic notion of sequential circuit, it can handle pipe-line circuits and multi-cycle operations.

Example An example of the equivalence checking between different control signals is shown with the design whose datapath is shown in Figure 4.7. $a[8], b[8], c[8]$ are 8-bit registers, $mult1[8]$, $add1[8]$, $shiftl1[8]$ are multiplier, adder, and left shifter whose inputs and outputs are 8-bit. $mux1[8] - 3[8]$ are three 8-bit multiplexers, and $cd1 - 4$ are control signals for them from a controller. When values of some control signals are 1, the corresponding multiplexer lines are connected, and signals are transmitted.

In this circuit, an expression $a[8] \times 8'h05$ where a is an integer number, can be performed by two ways of computations such that

$$mult1[8](a, 5)$$
$$add1[8](a, shiftl1[8](a, 8'h02))$$

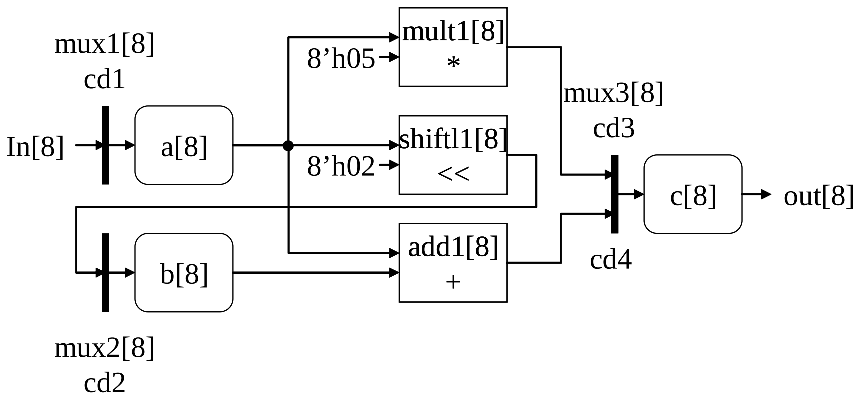


Figure 4.7: An example of datapath for bit-level analysis

This is included in the listed equivalence candidates in Section 4.4.1. The sequence of control signal values to execute each computation is shown as follows.

- $mult1[8](a[8], 8'h05) : (cd1, cd2, cd3, cd4) = (0, 0, 1, 0)$
- $add1[8](a[8], shiftl1[8](a[8], 8'h05)) : (cd1, cd2, cd3, cd4) = (0, 1, 0, 0), (0, 1, 0, 0)$

The first computation takes one cycle, and the second computation takes two cycles. The equivalence checking method explained in this section is applied to those sequences of control signal values.

Figure 4.8 shows the flow to convert the first sequence of control signal values into a combinational circuit. Since the first sequence only includes one control signal value, the datapath does not have to be unrolled. The multiplexers in Figure 4.7 are connected as the control signal values shown at the left hand side in Figure 4.8. The portions where signals are not transmitted are removed from the circuit as shown in the upper circuit of Figure 4.8. Registers are removed from the circuit, and finally the lower circuit in Figure 4.8 is generated.

Similarly, the second sequence of control signal values are converted into a combinational circuit as shown in Figure 4.9. Since the sequence takes two cycles, the datapath is unrolled for two cycles. For each cycle, multiplexers in Figure 4.7 are connected as the control signal value shown at the left hand side in Figure 4.9. After removing the portions where signals are not transmitted, the upper two circuits

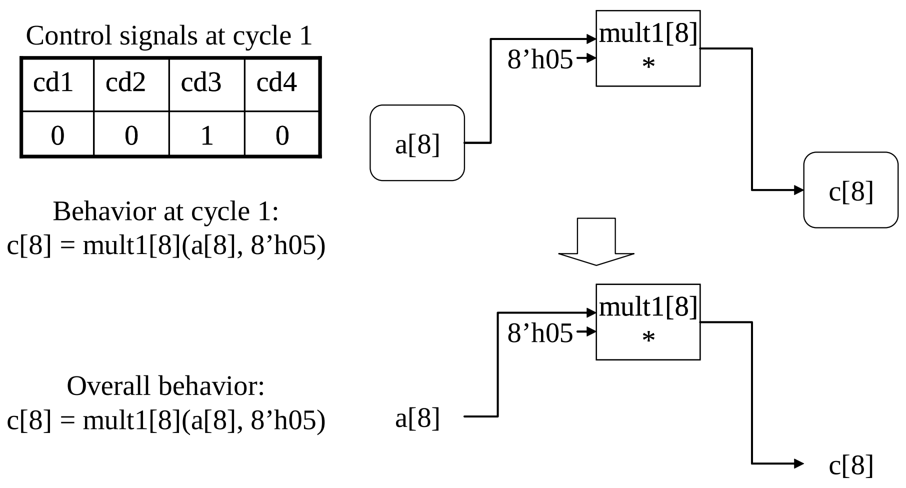

Control signals at cycle 1
cd1 cd2 cd3 cd4
0 0 1 0
Behavior at cycle 1:
c[8] = mult1[8](a[8], 8'h05)
Overall behavior:
c[8] = mult1[8](a[8], 8'h05)
mult1[8]
*
8'h05
a[8]
c[8]

Figure 4.8: A set of control signals and the generated combinational circuit

in Figure 4.9 are generated. Those circuits are connected, and the registers are removed. Finally, the lower circuit in Figure 4.9 is generated.

As the final step, those generated two combinational circuits are compared by existing combinational equivalence checkers. This equivalence checking is performed in bit-level accuracy.

4.4.3 Input of Equivalence Checking

Two RTL models which are inputs of the equivalence checking stage are represented by FSMDs. Since the RTL models have already been separated to controllers and datapaths, they can easily be described in FSMDs. For instance, an RTL design shown in Figure 4.2 having a datapath shown in Figure 4.3 can be described with an FSMD shown in Figure 4.10. A behavioral description in Figure 4.1 can be also converted to an RTL model by synthesizing the description by NISC compiler for the datapath shown in Figure 4.3. NISC compiler generates a set of control signals for the datapath whose behavior is equivalent to the behavioral description. An example of the generated control signals are shown at the upper side in Figure 4.11. This control signal set can also be represented by FSMD as shown at the lower side in Figure 4.11.

Control signals at cycle 1

cd1	cd2	cd3	cd4
0	1	0	0

Behavior at cycle 1:
b[8] = shiftl1[8](a[8], 8’h02)

Control signals at cycle 2

cd1	cd2	cd3	cd4
0	1	0	0

Behavior at cycle 2:
c[8] = add1[8](a[8], b[8])

Overall behavior:
c[8] = add1[8] (a[8], shiftl1[8](a[8], 8’h02))

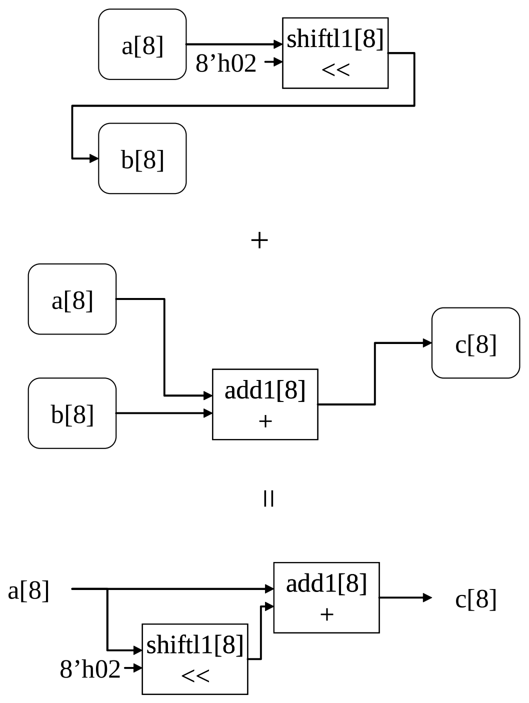


Figure 4.9: The other set of control signals and the generated combinational circuits

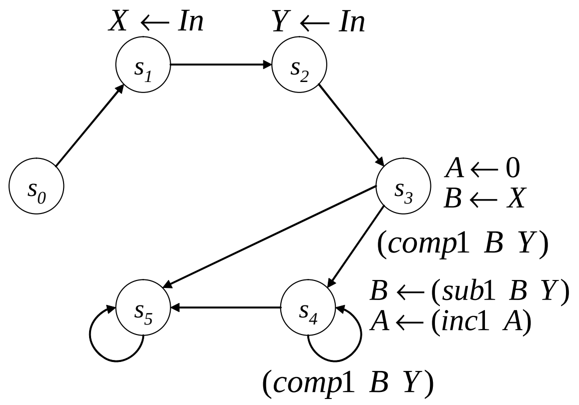


Figure 4.10: The FSMD of the design in Figure 4.2

In those FSMDs, functional units in the RTL datapaths are represented by operators in F. Some functions can be considered as identical functions if they are proved to be equivalent by the process described in Section 4.4.2. It is also as-

Program Counter	cd1	cd2	cd3	cd4	cd5	cd6	Program Counter at the next cycle
0 (s_a)	1	0	0	0	0	0	1
1 (s_b)	0	1	1	0	1	0	comp1(B, Y) ? 2 : 4
2 (s_c)	0	0	0	0	0	1	3
3 (s_e)	0	0	0	1	0	0	comp1(B, Y) ? 2 : 4
4 (s_e)	0	0	0	0	0	0	4

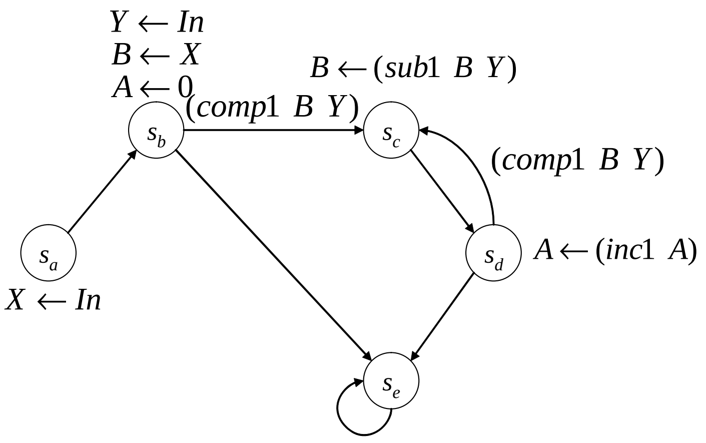


Figure 4.11: An output of NISC compiler and the corresponding FSMD

sumed that correspondences of inputs and outputs between two FSMDs are known. These correspondences are required to define the equivalence of two designs. In the proposed method, these correspondences must be given by users.

4.4.4 Definition of Equivalence

In this section, some notations and equivalences are defined. Also, equivalences of inputs and outputs which must be given by users are explained.

From this section, a symbol X_1 denotes a symbol X of the first design, and a symbol X_2 denotes a symbol X of the second design. Two FSMDs M_1 and M_2 are compared. Since the datapaths of M_1 and M_2 are same, the equivalence of the two designs' functions, $F_1 = F_2$, can be assumed and they can be described with F.

First, symbolic values of expressions are defined.

Definition 7 (**Symbolic value at state).** Let

$$Z_S \subseteq (E_1 \times S_1) \cup (E_2 \times S_2)$$

denote a set of symbolic values at states, and $(e, s) \in Z_S$ denote the symbolic value of an expression e at a state s. Since (e, s) is symbolic, it represents all values of e at s. Concrete values and a number of arrival times are abstracted.

$(e_1, s_1) \in Z_S$ and $(e_2, s_2) \in Z_S$ are equivalent when the following conditions are satisfied.

- Conditions to reach s_1 and s_2 from the initial states for the same number of times are equivalent.
- Values of e_1 and e_2 are always equivalent when the FSMD(s) arrives at s_1 and s_2 for the same number of times, respectively.

This equivalence is denoted by an operator "≡" as

$$(e_1, s_1) \equiv (e_2, s_2)$$

For example, in Figure 4.12, $(a, s_2) \equiv (b, s_b)$ is true when the inputs in_1 and in_2 are corresponding. Here, the values of in_1 and in_2 at k-th cycle are represented by in_1^k and in_2^k, respectively. The conditions to reach s_2 and s_b for n times are as follows.

$$\bigwedge_{i=1}^{n-1} in_1^i$$

$$\bigwedge_{i=1}^{n-1} in_2^i$$

Thus, the first condition of the definition is satisfied. The values of a and b on nth arrivals at s_2 and s_b, respectively, are both n. Therefore, the second condition is satisfied, and the equivalence is valid.

Definition 8 (**Symbolic value on transition).** Let

$$Z_T \subseteq (E_1 \times T_1) \cup (E_2 \times T_2)$$

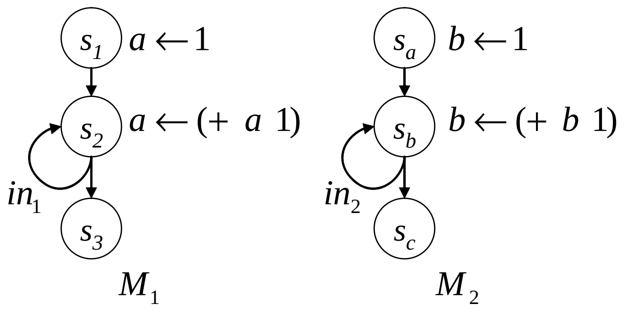


Figure 4.12: Example for Equivalence Definitions

denote a set of symbolic values on sequences of state transitions, and a pair $(e, t) \in Z_T$ denote the symbolic value of an expression e on a sequence of state transition $t = (s_0, s_1, \cdots, s_n)$. Since (e, t) is symbolic, it represents all values of e at $s_n \in S$ when M_1 or M_2 transits through t. Concrete values and a number of transition times are abstracted.

$(e_1, t_1) \in Z_T$ and $(e_2, t_2) \in Z_T$ are equivalent when the following conditions are satisfied.

- Conditions to transit through t_1 and t_2 from the initial states for the same number of times are equivalent.
- If $t_1 = (s_{10}, s_{11}, \cdots, s_{1n})$ and $t_2 = (s_{20}, s_{21}, \cdots, s_{2m})$ are true, then the values of e_1 and e_2 are always equal when arriving at s_{1n}, s_{2m} for the same number of times, respectively.

This equivalence is denoted by an operator "≡" as

$$(e_1, t_1) \equiv (e_2, t_2)$$

For example, in Figure 4.12, $(a, (s_2, s_2)) \equiv (b, (s_b, s_b))$ is true when in_1 and in_2 are corresponding. Conditions to transit through the transitions (s_2, s_2) and (s_b, s_b) for n times are as follows.

$$\bigwedge_{i=1}^{n} in_1^i$$

$$\bigwedge_{i=1}^{n} in_2^i$$

Then the first condition of the definition is satisfied. The values of a and b on nth arrivals at s_2 and s_b, respectively, are both n. Therefore, the second condition is satisfied, and the equivalence is valid.

Any function can be applied to those symbolic expressions, $(e_1, s_1) \in Z_S$ and $(e, t) \in Z_T$. If all argument expressions of a function are at a same state or on a same transition, such a function can be considered as a function at the state or on the transition. This can be represented by the next equations.

$$\begin{aligned}(f\ (e_1,\ s)\ (e_2,\ s) \cdots) &\equiv ((f\ e_1\ e_2 \cdots), s)\\ (f\ (e_1,\ t)\ (e_2,\ t) \cdots) &\equiv ((f\ e_1\ e_2 \cdots), t)\end{aligned}$$

where $f \in F$, $e_1, e_2, ... \in E$, $s \in S$, $t \in T$, and functions are represented by a Lisp-like style.

In addition, when two symbolic values $z_1, z_2 \in Z_S \cup Z_T$ are equivalent and a part of a symbolic value $z_3 \in Z_S \cup Z_T$ corresponds to z_1, the part can be substituted as the substitution of z_1 with z_2. z_3s before and after the substitution are equivalent. For example, when $(e_1, s_1) \equiv (e_2, s_2)$ is true, $((f\ e_1), s_1) \equiv ((f\ e_2), s_2)$, where $f \in F$, is clearly proved to be true by substituting e_1 at s_1 with e_2 at s_2.

Next, two types of equivalence classes each of which represents a set of equivalent symbolic values are defined.

Definition 9 (Equivalence class). Equivalence class of states is a set $Z_{S1} \subseteq Z_S$ that all contained elements are equivalent. Similarly, equivalence class of sequences of state transitions is a set $Z_{T1} \subseteq Z_T$ where all contained elements are equivalent. If same elements are contained in more than one equivalence classes, those equivalence classes can be merged to a single equivalence class.

With those equivalence classes, correspondences of inputs and outputs which are given by users are described as follows.

$$\{\{(in_{1i}, s_{1in_i}), (in_{2i}, s_{2in_i})\} | 1 \leq i, in_{1i} \in I_1, in_{2i} \in I_2, s_{1in_i} \in S_1, s_{2in_i} \in S_2\}$$
$$\{\{(out_{1i}, s_{1out_i}), (out_{2i}, s_{2out_i})\} | 1 \leq i, out_{1i} \in O_1, out_{2i} \in O_2, s_{1out_i} \in S_1, s_{2out_i} \in S_2\}$$

s_{1in_i} and s_{2in_i} are states where the ith inputs are valid, respectively. s_{1out_i} and s_{2out_i} are states where the ith outputs are valid, respectively.

Finally, the equivalence of two designs (FSMDs) is defined as follows.

Definition 10 (Equivalence of FSMDs). Two FSMDs M_1 and M_2 are equivalent when the next equation is true.

$$\bigwedge_i ((in_{1i}, s_{1in_i}) \equiv (in_{2i}, s_{2in_i})) \Longrightarrow \bigwedge_i ((out_{1i}, s_{1out_i}) \equiv (out_{2i}, s_{2out_i}))$$

where $in_{1i} \in I_1$ and $out_{1i} \in O_1$ are the i th input and output of M_1, respectively, and $in_{2i} \in I_2$ and $out_{2i} \in O_2$ are the i th input and output of M_2, respectively.

Therefore, in the proposed method, equivalences between all corresponding outputs of two designs are checked under an assumption that all corresponding inputs of the two designs are equivalent.

4.4.5 Equivalence Checking of Symbolic Expressions

To apply the equivalence checking method explained in the latter two sections, equivalences of the symbolic expressions are checked. This section explains a method to check the equivalence of the symbolic expressions for states or sequences of state transitions defined in the previous section.

A symbolic expression consists of an expression ($e \in E$), and a state($s \in S$) or a sequence of state transition ($t \in T$), and an expression consists of a combination of variables, inputs, outputs, function calls, and sub expressions.

With the relation described in the previous section, a function at a state or on a sequence of state transitions, such as $((f\ e_1\ e_2\ \cdots), s)$ or $((f\ e_1\ e_2\ \cdots), t)$, where $f \in F$, $e_1, e_2, \cdots \in E$, $s \in S$, $t \in T$, can be converted into $(f\ (e_1,\ s)\ (e_2,\ s)\ \cdots)$ or $(f\ (e_1,\ t)\ (e_2,\ t)\ \cdots)$, respectively. This conversion is repeatedly applied to symbolic expressions while it can be applied.

Then, the expressions are represented only by variables at states, variables on sequences of state transitions (they are denoted as symbolic variables), and functions which are applied to those symbolic variables. Here, a symbolic variable is treated as a unit, and same symbolic variables (same variables at same states or on same sequences of state transitions) are equivalent.

With a conversion of each unit into a variable and each function into an Uninterpreted Function(UF), decision procedures for a Logic of Equality with Uninterpreted Function(EUF)[26] can be applied to check the equivalence. If an EUF formula that says two symbolic expressions are equivalent is valid, the two symbolic expressions are proved to be equivalent.

Here, it is important to make as much functions as possible same UFs to improve the possibility to prove such equivalences. As explained in Section 4.3, computations with same control signals from controllers can be converted into same UFs after the conversion which makes two designs have identical datapaths. In addition, expressions which are proved to be equivalent in Section 4.4.1 can be converted into the same UFs.

Symbolic expressions in an equivalence class are also assumed to be equivalent when the validities of the EUF formulas are checked. Practically, this step is performed with decision procedures (SMT solvers) which can handle EUF, such as CVC3[40].

4.4.6 Equivalence Checking of FSMDs by Symbolic Simulation

In this section, the method to apply equivalence class based symbolic simulation is explained. It has been introduced in Section 2.4, but modified to check the equivalence of FSMDs defined in Section 4.4.4.

Before the verification, all loops must be unrolled since symbolic simulation cannot handle them. The verification is performed by the following steps.

1. From the initial states, transitions are traversed forwardly with getting equivalences of left-hand sides and right-hand sides of assignments. The left-hand side data register value at the next state is equivalent to the right-hand side expression value at the current state. The left-hand side output value at the current state is equivalent to the right-hand side expression value at the current state.

2. When there are more than one next states, the current checking process forks.

3. FSMDs are equivalent when the equivalence of FSMDs (Definition 10) is satisfied in all checking processes.

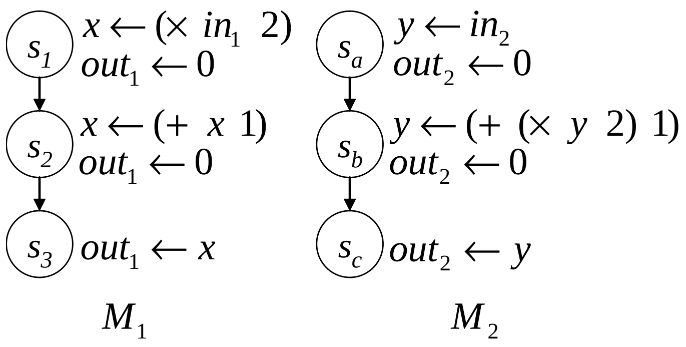


Figure 4.13: Example 1

An example of the verification is shown with the two FSMDs in Figure 4.13. $s_1, s_2, s_3 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states, $x \in V_1$ and $y \in V_2$ are data registers, $in_1 \in I_1$ and $in_2 \in I_2$ are corresponding inputs, $out_1 \in O_1$ and $out_2 \in O_2$ are corresponding outputs, $0, 1, 2 \in K_1 \cap K_2$ are constants, and $+, \times \in F$ denote addition and multiplication, respectively. The given initial equivalence class is

$$\{(in_1, s_1), (in_2, s_a)\}$$

The output equivalence to be proved is

$$(out_1, s_3) \equiv (out_2, s_c)$$

First, from the assignments in s_1 and s_a, the following equations are proved to be true.

$$(x, s_2) \equiv ((\times\ in_1\ 2), s_1)$$
$$(out_1, s_1) \equiv 0$$
$$(y, s_b) \equiv (in_2, s_a)$$
$$(out_2, s_a) \equiv 0$$

Then, with substitutions, the equivalence classes become as follows.

$$\{(in_1, s_1), (in_2, s_a), (y, s_b)\}$$
$$\{(x, s_2), ((\times\ in_1\ 2), s_1), ((\times\ y\ 2), s_b)\}\}$$
$$\{(out_1, s_1), (out_2, s_a), 0\}$$

Next, from the assignments in s_2 and s_b, the following equations can be obtained.

$$\begin{aligned}
&(x, s_3) \equiv ((+\ x\ 1), s_2) \\
&(y, s_c) \equiv ((+\ (\times\ y\ 2)\ 1), s_b) \\
&(out_1, s_2) \equiv 0 \\
&(out_2, s_b) \equiv 0
\end{aligned}$$

Then, with substitutions, the equivalence classes become as follows.

$$\begin{aligned}
&\{(in_1, s_1), (in_2, s_a), (y, s_b)\}, \\
&\{(x, s_2), ((\times\ in_1\ 2), s_1), ((\times\ y\ 2), s_b)\}, \\
&\{(x, s_3), ((+\ x\ 1), s_2), ((+\ (\times\ y\ 2)\ 1), s_b)\}, \\
&\{(y, s_c), ((+\ (\times\ y\ 2)\ 1), s_b)\}\} \\
&\{(out_1, s_1), (out_2, s_a), (out_1, s_2), (out_2, s_b), 0\}
\end{aligned}$$

Since the third and forth equivalence classes include the same entry, those can be merged.

From the assignment in s_3 and s_c, the following equations are generated.

$$\begin{aligned}
&(out_1, s_3) \equiv (x, s_3) \\
&(out_2, s_c) \equiv (y, s_c)
\end{aligned}$$

Then, with substitutions, the equivalence classes becomes as follows.

$$\begin{aligned}
&\{(in_1, s_1), (in_2, s_a), (y, s_b)\}, \\
&\{(x, s_2), ((\times\ in_1\ 2), s_1), ((\times\ y\ 2), s_b)\}, \\
&\{(x, s_3), ((\times\ x\ 1), s_2), ((+\ (\times\ y\ 2)\ 1), s_b), (y, s_c), (out_1, s_3), (out_2, s_c)\} \\
&\{(out_1, s_1), (out_2, s_a), (out_1, s_2), (out_2, s_b), 0\}
\end{aligned}$$

Since the third equivalence class includes both (out_1, s_3) and (out_2, s_c), M_1 and M_2 are proved to be equivalent for the given equivalence specification.

As mentioned in Section 4.1, this method is fast and reasonable when there are a small number of control branches and loops whose numbers of iterations are small. In addition, if there are infinite loops, the correctness of the results is guaranteed only up to the unrolling number can be got. For such cases, a rule-based method is proposed in the next section.

4.4.7 Rule-Based Equivalence Propagation

In this section, a rule-based equivalence checking method is proposed. This method can be applied to FSMDs directly, and five rules explained below propagate the equivalences of inputs to those of outputs.

Rule for Output Assignments

Rule 1. Let $s \in S$ be a state, and $a \in A_O$ be an assignment to an output at the state which satisfies $(s, a) \in P$. Next, let $o \in O$ be an output, and $e \in E$ be an expression which satisfy $(o, e) = L_A(a)$. Then the next equation is true.

$$(o, s) \equiv (e, s)$$

Proof. Since an assignment to an output at a state is always performed when the FSMD reaches the state, and the output value is updated immediately. Therefore, the left-hand side and the right-hand side of the assignment must be identical value at the state. □

This rule simply expresses the relationship between left-hand sides and right hand sides of assignments to outputs. Since the value of the left-hand side outputs are updated immediately to the value of the right-hand side expressions when states include those assignments are reached.

In the example FSMDs in Figure 4.13 which were used to explain the symbolic simulation based method, outputs are out_1 and out_2. From this rule, the following equations are generated for those outputs.

$$\begin{aligned}
(out_1, s_1) &\equiv 0 \\
(out_1, s_2) &\equiv 0 \\
(out_1, s_3) &\equiv (x, s_3) \\
(out_2, s_a) &\equiv 0 \\
(out_2, s_b) &\equiv 0 \\
(out_2, s_c) &\equiv (y, s_c)
\end{aligned}$$

Those equations are generated from the assignments at $s_1, s_2, s_3, s_a, s_b, s_c$, respectively.

Rule for Data Register Assignments

Rule 2. If $r = (s_1, s_2) \in R$ satisfies the following equation

$$\{\forall s_a \in S, (s_a \neq s_2) \rightarrow ((s_1, s_a) \notin R)\} \wedge \{\forall s_b \in S, (s_b \neq s_1) \rightarrow ((s_b, s_2) \notin R))\}$$

then the next equation is true.

$$\forall a \in A_V, \forall v \in V, \forall e \in E, \{((s_1, a) \in P \wedge a = (v, e)) \rightarrow ((v, s_2) \equiv (e, s_1))\}$$

Proof. r is the only transition from s_1, and it is also the only transition to s_2. Therefore, when M_1 or M_2 reaches to s_1, it always reaches to s_2 after the next transition. Then, the transition condition to reach s_1 and s_2 for the same number of times from the initial state must be equivalent. Also, the values of the data registers at the left-hand sides of the assignments are always updated to the values of the right-hand sides after the transition. Therefore, the value of v at s_2 is always equivalent to that of e at s_1 when the FSMD arrives at s_1 and s_2 for the same number of times, respectively. Since the two conditions in Definition 8 are satisfied, $(v, s_2) \equiv (e, s_1)$ are true. □

From the FSMDs in Figure 4.13, this rule generates the following equations.

$$\begin{aligned}
&(x, s_2) \equiv ((\times\ in_1\ 2), s_1) \\
&(x, s_3) \equiv ((+\ x\ 1), s_2) \\
&(y, s_b) \equiv (in_2, s_a) \\
&(y, s_c) \equiv ((+\ (\times\ y\ 2)\ 1), s_b)
\end{aligned}$$

This rule and Rule 1 correspond to symbolic simulation applied in the previous section. Therefore, the combination of above equations and those of Rule 1's example return the same results.

Combination of Rule 1 and Rule 2 returns the same results as that of symbolic simulation as introduced in the previous section, unless conditional branches exist in FSMDs.

Rule for Sequence of State Transitions

Rule 3. Let t_1 and t_2 be sequences of state transitions such that:

$$t_1 = (s_{10}, s_{11}, \cdots s_{1m}) \in T_1$$
$$t_2 = (s_{20}, s_{21}, \cdots s_{2n}) \in T_2$$

Let c_{1i} and c_{2i} be transition conditions for each transitions in t_1 and t_2, respectively, such that

$$c_{1i} = Q((s_{1i}, s_{1i+1})) \in E_2 | 0 \leq i \leq m-1$$
$$c_{2i} = Q((s_{2i}, s_{2i+1})) \in E_2 | 0 \leq i \leq n-1$$

When t_1 and t_2 are assumed to be executed, Rule 2 can be applied for all states in them since there are no joins and branches. Rule 1 can also be applied for all states. Under the assumption, for each $(e_1, e_2) \in E_1 \times E_2$, if the following equation is true,

$$\left\{ \bigwedge_{0 \leq i \leq m-1} ((c_{1i}, s_{1i})) \equiv \bigwedge_{0 \leq i \leq n-1} ((c_{2i}, s_{2i})) \right\} \wedge ((e_1, s_{1m}) \equiv (e_2, s_{2n}))$$

then $(e_1, t_1) \equiv (e_2, t_2)$ is satisfied, where $\bigwedge$ represents AND.

Proof. If the second half part of the equation is true, transition conditions to reach the last states in t_1 and t_2 for the same number of times from the initial states must be equivalent from Definition 7. From this part, the second condition in Definition 8 is also satisfied. In addition, transition conditions between t_1 and t_2 are equivalent from the first half part of the equation. Then, with taking conjunction of those transition conditions, respectively, the conditions to transit t_1 and t_2 for the same number of times from the initial states become equivalent. Then, the first condition in Definition 8 is satisfied. Since the two conditions in Definition 8 are satisfied, $(e_1, t_1) \equiv (e_2, t_2)$ is proved to be true. $\square$

Figure 4.14 shows an example where Rule 3 can be applied. $s_1, s_2 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states. $a, x \in V_1$ and $y, b \in V_2$ are data registers. $+, > \in F$ denote

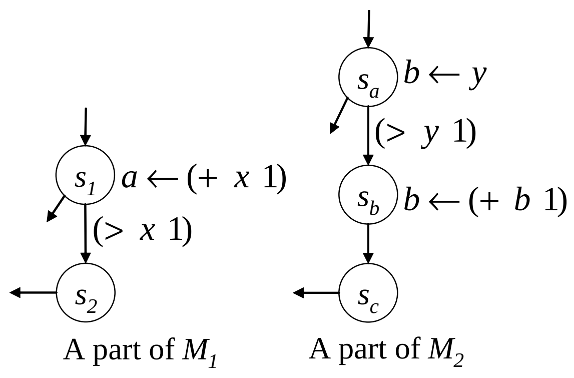


Figure 4.14: Example 2

addition and greater than operation, respectively. $1 \in K_1 \cap K_2$ is a constant. An initial equivalence class is given as follows:

$$\{(x, s_1), (y, s_a)\}$$

First, let's assume that M_1 transits through $t_1 = (s_1, s_2)$, and M_2 transits through $t_2 = (s_a, s_b, s_b, s_c)$. Then, with applying Rule 1, the following equivalence classes can be obtained.

$$\{(x, s_1), (y, s_a), (b, s_b)\}$$
$$\{(a, s_2), ((+\ x\ 1), s_1)\}$$
$$\{(b, s_c), ((+\ b\ 1), s_b)\}$$

Since $((>\ \ x\ 1), s_1) \equiv ((>\ \ y\ 1), s_a)$ becomes true from the first equivalence class, the transition conditions are equivalent. From a substitution and a merger, he following equivalence class is also generated.

$$\{(a, s_2), ((+\ x\ 1), s_1), (b, s_c), ((+\ b\ 1), s_b)\}$$

Therefore, $(a, s_2) \equiv (b, s_c)$ is true. Finally, $(a, t_1) \equiv (b, t_2)$ is proved by Rule 3.

Rule for State

Rule 4. Let $T_a \subseteq T$ be a set of sequences of state transitions, $S_{T_a} \subseteq S$ be a set of states included in the transitions of T_a, and $s \in S$ be a state. If there is no sequence of state transitions $t \in T$ whose first state is s and the states in t are not included in S_{T_a}, T_a covers all paths to s.

Let T_{a1} and T_{a2} denote sets of sequence of states transitions such that

$$T_{a1} = \{t_{1i} | 0 \leq i \leq m-1, t_{1i} \in T_1\}$$
$$T_{a2} = \{t_{2i} | 0 \leq i \leq n-1, t_{1i} \in T_2\}$$

which reach states $s_1 \in S_1$ and $s_2 \in S_2$ with covering all paths to s_1 and s_2, respectively.

Then the next formula is true.

$$\left\{(m = n) \wedge \bigwedge_{i=0}^{m-1} ((e_1, t_{1i}) \equiv (e_2, t_{2i}))\right\} \to (e_1, s_1) \equiv (e_2, s_2)$$

where $e_1 \in E_1$ and $e_2 \in E_2$.

This rule shows that if all paths to s_1 and s_2 have corresponding paths where e_1 and e_2 are equivalent, then the values of e_1 at s_1 and e_2 at s_2 are always equivalent. In this rule, the number of corresponding paths in the two FSMDs must be same. It means FSMDs which have the same structures of conditional branches can be verified with this rule. This limitation is also valid in Rule 4, since Rule 3 is performed to apply Rule 4.

Proof. Each equivalence of corresponding sequences of state transitions shows that the transition conditions to transit through those transitions from the initial states are equivalent, respectively. Therefore, the orders to reach s_1 and s_2 among those corresponding transitions are fixed, and completely equivalent in each pair of corresponding transitions. Then, the first condition in Definition 8 is satisfied. The second condition in Definition 8 is also clearly satisfied by the equivalences of e_1 and e_2 on corresponding sequences of state transitions. Therefore, both the conditions in Definition 7 are satisfied, and $(e_1, s_1) \equiv (e_2, s_2)$ is proved to be true. $\square$

Figure 4.15 shows an example where Rule 4 can be applied. $s_1, s_2, s_3, \in S_1$ and $s_a, s_b, s_c \in S_2$ are states, and $a \in V_1$ and $b \in V_2$ are data registers. $t_1 = (s_1, s_3)$, $t_2 = (s_2, s_3)$, $t_a = (s_a, s_c)$, and $t_b = (s_b, s_c)$ are sequences of state transitions. Here,

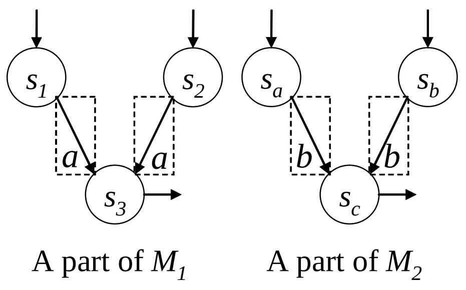


Figure 4.15: Example 3

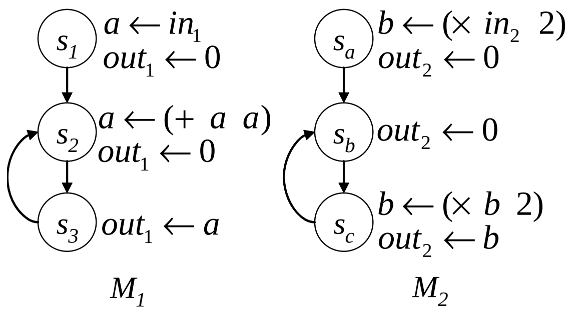


Figure 4.16: Example 4

$T_{a1} = \{t_1, t_2\}$ and $T_{a2} = \{t_a, t_b\}$ cover all paths to s_3 and s_c, respectively. Assume that the following equivalence classes have already been proved.

$$\{(a, t_1), (b, t_a)\}$$
$$\{(a, t_2), (b, t_b)\}$$
$$\{(a, t_3), (b, t_c)\}$$

Then, all the paths to s_3 and s_c have corresponding paths where a and b are equivalent. From Rule 4, $(a, s_3) \equiv (b, s_c)$ is true.

Rule for Loop The last rule is for FSMDs which have loops such as M_1 and M_2 in Figure 4.16. The equivalence of such FSMDs cannot be proved only with Rule 1∼4 since previous results of the computation are used in each iteration. The next rule can be applied in such cases.

Rule 5. Let $s_1 \in S_1$ and $s_2 \in S_2$ denote one of the states in different loops, respectively. Let $T_{11} = \{t_i^1 | 1 \leq i \leq l, t_i^1 \in T_1\}$ and $T_{12} = \{t_i^2 | 1 \leq i \leq m, t_i^2 \in T_2\}$ denote sets of sequences of state transitions reaching s_1 and s_2 which cover all paths from the inside of the loops to s_1 and s_2, respectively. Let $T_{13} = \{t_i^3 | 1 \leq i \leq n, t_i^3 \in T_1\}$ and $T_{14} = \{t_i^4 | 1 \leq i \leq k, t_i^4 \in T_2\}$ also denote sets of sequences of state transitions reaching s_1 and s_2 which cover all paths from the outside of the loops to s_1 and s_2, respectively.

Then, $(e_1, s_1) \equiv (e_2, s_2)$ where $e_1 \in E_1, e_2 \in E_2$ is true when the following two conditions are satisfied.

- The next equation is true
$$(n = k) \wedge \bigwedge_{i=1}^{n} ((e_1, t_i^3) \equiv (e_2, t_i^4))$$
- Under an assumption that $(e_1, s_1) \equiv (e_2, s_2)$ is true, the next equation is true with Rule 1∼4.
$$(l = m) \wedge \bigwedge_{i=1}^{l} (e_1, t_i^1) \equiv (e_2, t_i^2))$$

Proof. This rule is proved with unrolling the loops as shown in Figure 4.17 and the following induction. Let ${}^is_1, {}^is_2$ denote ith s_1 and s_2 after the loops are unrolled, respectively. Let ${}^it_j^1, {}^it_j^2$ also denote ith t_j^1 and t_j^2. The first condition in Rule 5 is the basic case which proves $(e_1, {}^1s_1) \equiv (e_2, {}^1s_2)$ by Rule 4. The second condition is the inductive step which proves $(e_1, {}^{i+1}s_1) \equiv (e_2, {}^{i+1}s_2)$ by Rule 4 under the assumption $(e_1, {}^is_1) \equiv (e_2, {}^is_2)$. Therefore, the next equation is inductively proved.
$$\bigwedge_{i=1}^{\infty} (e_1, {}^is_1) \equiv (e_2, {}^is_2)$$
This is equivalent to $(e_1, s_1) \equiv (e_2, s_2)$. □

As written in the proof, equivalences are propagated from the assumption to apply Rule 4 for the state s_1 and s_2 in the inductive step. This propagation is performed by applying Rule 1∼4 multiple times. Here, only the assumption has to be finally proved, and the way to apply Rule 1∼4 does not have to be considered. Then, only the final step where Rule 4 is applied is defined in the rule. Therefore,

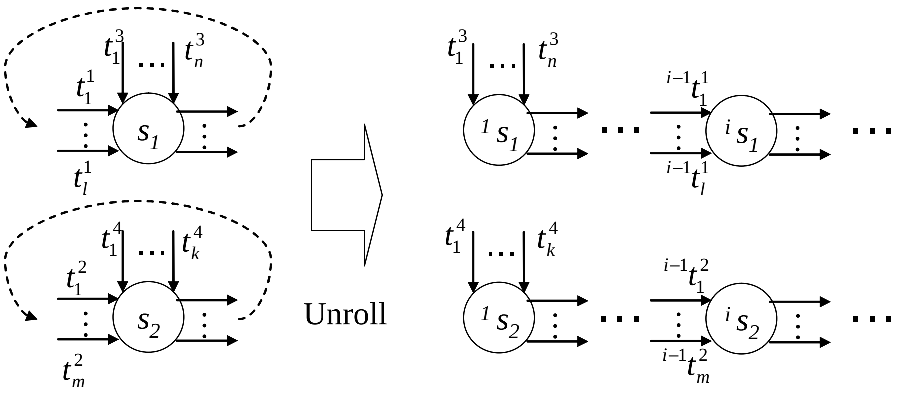


Figure 4.17: Loop unrolling for the proof of Rule 5

the first states of t_i^1 and t_i^2 in the rule can be arbitrary states in the insides of the loops.

With this rule, the equivalence of M_1 and M_2 in Figure 4.16 can be proved. $s_1, s_2, s_3 \in S_1$ and $s_a, s_b, s_c \in S_2$ are states. $in_1 \in I_1$ and $in_2 \in I_2$ are corresponding inputs. $out_1 \in O_1$ and $out_2 \in O_2$ are corresponding outputs. $a \in V_1$ and $b \in V_2$ are data registers. $0, 2 \in K_1 \cap K_2$ are constants. $+, * \in F$ are addition and multiplication, where $(+\ x\ x) = (*\ x\ 2)$ for $x \in V$ has already been proved to be equivalent.

Initial equivalence class is

$$\{(in_1, s_1) \equiv (in_2, s_a)\}$$

The goal is to prove $(out_1, s_3) \equiv (out_2, s_c)$.

First, the following equation is proved with Rule 3.

$$(a, (s_1, s_2, s_3) \equiv (b, (s_a, s_b, s_c)) \tag{4.1}$$

Next, Rule 5 is applied to prove the next equation

$$(a, s_3) \equiv (b, s_c) \tag{4.2}$$

Basic Case

Equation 4.1 satisfies the first condition of Rule 5.

Inductive Step

Equation 4.2 is assumed. Under the assumption, the next equation is proved with Rule 3.

$$(a, (s_3, s_2, s_3)) \equiv (b, (s_c, s_b, s_c)) \tag{4.3}$$

With Equation 4.3, the second condition of Rule 5 is satisfied.

Then, Equation 4.2 is proved by Rule 5 so that Equation 4.3 is also satisfied.

Next, the following equations are proved on states s_3 and s_c, respectively, from Rule 1.

$$(out_1, s_3) \equiv (a, s_3) \tag{4.4}$$

$$(out_2, s_c) \equiv (b, s_c) \tag{4.5}$$

Then, $(out_1, s_3) \equiv (out_2, s_c)$ is proved by Equation 4.2, Equation 4.4, and Equation 4.5.

Now, the equivalence of outputs is proved and M_1 and M_2 are proved to be equivalent.

This rule can handle nested loops by recursively applying the rule to inner loops under the assumption of the inductive step.

Algorithm to Apply the Rules In this section, an algorithm to apply the proposed five rules to designs is discussed. Types of designs which can be verified by this rule-based method are also discussed.

Algorithm 7, Algorithm 8, and Algorithm 9 show a simple algorithm to apply the proposed five rules. Algorithm 7 and Algorithm 8 are subroutines used in Algorithm 9.

This algorithms consist of the following four steps:

1. Equivalences of inputs given by users are added to the equivalence classes.
2. Rule 1 and Rule 2 are applied to all transitions (Algorithm 9).
3. Rule 3 and 4 are applied to each state (Algorithm 7).
4. Rule 5 is applied to prove the equivalence of loops (Algorithm 8).

Algorithm 7 *Sub_1*: Subroutine to apply Rule 3 and Rule 4

```
Require: $z_s \subseteq Z_S$ {A set of existing equivalence classes for states}, $z_t \subseteq Z_T$ {A set of
    existing equivalence classes for sequence of state transitions}
 1: loop
 2:   $z'_s \leftarrow \emptyset \subseteq Z_S$ {An local set of equivalence classes for states}
 3:   $z'_t \leftarrow \emptyset \subseteq Z_T$ {An local set of equivalence classes for sequence of state transitions}
 4:   for each $(s_1, s_2) \in S_1 \times S_2$ do
 5:     $t_1 \leftarrow \{t \mid t \in T_1, Size(t) \leq L, LastState(t) = s_1\}$ {Collect sequences of state
        transitions reach $s_1$ whose length is equal to or less than $L$}
 6:     $t_2 \leftarrow \{t \mid t \in T_2, Size(t) \leq L, LastState(t) = s_2\}$ {Collect sequences of state
        transitions reach $s_2$ whose length is equal to or less than $L$}
 7:     for each $(e_2, e_2) \in E_1 \times E_2 \mid \{(e_1, s_1), (e_2, s_2)\} \notin z_s \cup z'_s$ do
 8:       for each $(t_a, t_b) \in T_1 \times T_2 \mid (e_1, t_1), (e_2, t_2)\} \notin z_t \cup z'_t$ do
 9:         if $Rule3(z_s \cup z'_s, (e_1, t_a), (e_2, t_b))$ =true then
10:           $z'_t \leftarrow z'_t \cup \{(e_1, t_a), (e_2, t_b)\}$ {When equivalence is proved, an equivalence
              class is added to the local set}
11:         end if
12:       end for
13:       if $z_t \cup z'_t \neq \emptyset$ then
14:         if $Rule4(z_t \cup z'_t, (e_1, s_1), (e_2, s_2))$ = true then
15:           $z'_s \leftarrow z'_s \cup \{(e_1, s_1), (e_2, s_2)\}$ {When equivalence is proved, an equivalence
              class is added to the local set}
16:         end if
17:       end if
18:     end for
19:   end for
20:   if $z'_s = \emptyset$ then
21:     return $(z'_s, z'_t)$ {When no additional equivalence class is added, newly generated
        equivalence classes are returned, and this routine finishes}
22:   end if
23: end loop
```

Algorithm 8 Sub_2: Subroutine to apply Rule 5

```
Require: $z_s \subseteq Z_S$ {Set of existing equivalence classes for states}, $z_t \subseteq Z_T$ {Set of existing
    equivalence classes for sequence of state transitions}, $loop \in Loop$ {Current loop}
 1: $Loop_{sub} \leftarrow getInnerLoops(loop)$ {Get the set of loops which are one level inner from
    $loop$ when it is not $NULL$. Otherwise, get the most outside loops}
 2: loop
 3:   $z'_s \leftarrow \emptyset \in Z_S$ {An local set of equivalence classes for states}
 4:   $z'_t \leftarrow \emptyset \in Z_T$ {An local set of equivalence classes for sequences of state transitions}
 5:   for each $loop_{sub} \in Loop_{sub}$ do
 6:     for each $(s_1, s_2) \in loop_{sub}$ do
 7:       $t_1 \leftarrow \{t \mid t \in T_1, Size(t) \leq L, LastState(t) = s_1, FirstState(t) \notin loop_{sub}\}$
          {Collect sequences of state transitions reach $s_1$ whose length is equal to or less
          than $L$ and first state is out of the loop}
 8:       $t_2 \leftarrow \{t \mid t \in T_2, Size(t) \leq L, LastState(t) = s_2, FirstState(t) \notin loop_{sub}\}$
          {Collect sequences of state transitions reach $s_2$ whose length is equal to or less
          than $L$ and first state is out of the loop}
 9:       for each $(e_2, e_2) \in E_1 \times E_2 \mid \{(e_1, s_1), (e_2, s_2)\} \notin z_s \cup z'_s$ do
10:         for each $(t_a, t_b) \in T_1 \times T_2 \mid (e_1, t_1), (e_2, t_2)\} \notin z_t \cup z'_t$ do
11:           if $Rule3(z_s \cup z'_s, (e_1, t_a), (e_2, t_b))$ =true then
12:             $z'_t \leftarrow z'_t \cup \{(e_1, t_a), (e_2, t_b)\}$ {When equivalence is proved, an equivalence
                class is added to the local set}
13:           end if
14:         end for
15:         if $z_t \cup z'_t \neq \emptyset$ then
16:           if $Rule4(z_t \cup z'_t, (e_1, s_1), (e_2, s_2))$ = true then
17:             $z_{assumed} \leftarrow \{(e_1, s_1), (e_2, s_2)\} \subseteq Z_S$ {Basic case is proved, and an as-
                sumption is made to prove inductive case}
18:             $(z''_s, z''_t) \leftarrow Sub_1(z_s \cup z'_s \cup z_{assumed})$ {Rule 3 and Rule 4 are applied to
                prove inductive case}
19:             if $z_{assumed} \notin z''_s$ then
20:               $z''_s \in Sub2(z_s \cup z'_s \cup z''_s, z_t \cup z'_t \cup z''_t, loop_{sub})$ {If the assumption is not
                  proved, Rule 5 is incrementally applied to the inner loops}
21:             end if
22:             if $z_{assumed} \in z''_s$ then
23:               $z'_s \leftarrow z''_s$ {If the assumption is proved, local equivalence classes are
                  updated}
24:               $z'_t \leftarrow z''_t$
25:             end if
26:           end if
27:         end if
28:       end for
29:     end for
30:   end for
31: end loop
```

Algorithm 9 Algorithm to apply the rules

```
1: $z_s = \emptyset \subseteq Z_S$ {A set of equivalence classes for states}
2: $z_t = \emptyset \subseteq Z_T$ {A set of equivalence classes for sequences of state transitions}
3: $z_s \in \mathit{GetInputEquivalenceClasses}()$ {Add initial input equivalence classes given by users}
4: for each $(s, a) \in (S_1 \times A_2) \cup (S_2 \times A_2)$ do
5:    $z_s \in z_s \cup \mathit{Rule}1(s, a)$ {Rule 1 is applied to check the equivalence of outputs for each state}
6:    $z_s \in z_s \cup \mathit{Rule}2(s, a)$ {Rule 2 is applied to check the equivalence of data registers for each state}
7:    if $\mathit{GetOutputEquivalenceClasses}() \subseteq z_s$ then
8:       return true {Case that output equivalence could be proved only by Rule 1 and Rule 2}
9:    end if
10:   $(z'_s, z'_t) \leftarrow \mathit{Sub}1(z_s, z_t)$ {Rule 3 and Rule 4 are applied by Algorithm 7}
11:   $z_s \leftarrow z_s \cup z'_s$
12:   $z_t \leftarrow z_t \cup z'_t$
13:   if $\mathit{GetOutputEquvalenceClasses}() \subseteq z_s$ then
14:      return true {Case that output equivalence could be proved without Rule 5}
15:   end if
16:   $(z'_s, z'_t) \leftarrow \mathit{Sub}2(z_s, z_t)$ {Rule 5 is applied by Algorithm 8}
17:   $z_s \leftarrow z_s \cup z'_s$
18:   $z_t \leftarrow z_t \cup z'_t$
19:   if $\mathit{GetOutputEquvalenceClasses}() \subseteq z_s$ then
20:      return true
21:   else
22:      return false
23:   end if
24: end for
```

Rule 5 is recursively applied to handle nested loops. Assumptions of the second step of Rule 5 are incrementally made from outer loops to inner loops, and those assumptions are guaranteed from that of the inner loops to that of the outer loops. L is a parameter which defines the maximum length of sequences of state transitions when Rule 3 is applied. If L becomes larger, the number of target sequences of state transitions also becomes large, and the complexity becomes higher. Then, L should be started from 1 and incremented until the equivalence is proved.

The termination of this algorithm is proved as follows. There are two infinite loops at line 1 in Algorithm 7 and line 2 in Algorithm 8. Both of them break when no more equivalence classes are generated in the loops. The number of equivalence classes is finite since the number of equivalence candidates is finite. Also, the recursive call of $Sub2$ eventually stops since the levels of multiple loops are finite. Therefore, this algorithm must terminate.

Limitations Here, it must be mentioned that the proposed rule-based verification method (including the five rules and the algorithm) is not complete. It just says "equivalent" in particular cases when the rules can prove equivalences. In other cases, the proposed method just says "indeterminable". However, the propose method is fast, and when a result is equivalent the result is guaranteed to be true.

By the proposed five rules to propagate equivalences, as mentioned in the explanation of Rule 4, FSMDs which have same structures of conditional branches can be verified. Note that lengths of transitions can be different between two FSMDs under verification unless there are no branches in the transitions. Outsides and insides of corresponding loops in two FSMDs must be also equivalent, respectively. Therefore, the method can verify designs before and after scheduling, retiming, or some optimizations like common sub-expression elimination, unless such optimizations are applied beyond loops.

4.5 Experimental Results

The verification flows shown in Section 4.3.1 was applied to realistic examples.

4.5.1 Tool Implementations

To generate a controller for a given datapath (explained in Section 4.3.2), an on-line NISC complier demo[163] was used. The separation of controllers and datapaths of the designs, and the translation from RTL description into FSMD were done by hand.

Two tools to check the equivalence of FSMDs had been implemented with C and C++. One is a symbolic simulator in which the method described in Section 4.4.6 is implemented. The other is a rule-based verifier in which the method explained in Section 4.4.7 is implemented. Both tools run on a PC with a 3GHz processor (dual core) and 1GB memory.

4.5.2 Examples

Three examples, DCT (Discrete Cosine Transform), IDCT (Inverse Discrete Cosine Transform), and Ellip (Elliptical Filter) were used. All examples are originally written in C, and the details are in Table 4.1.

Optimizations and high-level syntheses were applied to those examples by hand. Therefore, there are three versions for each example such as, (1) original design, (2) design after behavioral optimization, (3) design after high-level synthesis. The optimizations were removal of temporal variables, refinement of operations, and others. All synthesized designs use the same datapath which is about 1000 lines in Verilog, and those designs are pipelined. In all examples, variable names are not corresponding. The numbers of states, inputs, outputs, and variables in translated FSMDs are also shown in Table 4.1.

Table 4.1: Information of Examples

Example	Version	LOC in C	Control structure	Num. of states	Num. of inputs	Num. of outputs	Num. of variables
DCT	(1)	54	Trilaminar structure of 8-iterations	14	64	64	68
	(2)	52		13	64	64	66
	(3)	-		11	64	64	66
IDCT	(1)	134	Two large 8-iterations	98	64	64	83
	(2)	120		95	64	64	83
	(3)	-		90	64	64	75
Ellip	(1)	74	One large infinite loop	36	1	1	37
	(2)	67		33	1	1	35
	(3)	-		20	1	1	31

4.5.3 Verification Results

For each example, it took about 10 seconds to synthesize each controller by NISC compiler.

The verification time of equivalence checking between the FSMDs are shown in Table 4.2. All results were equivalent, and they were correct. Since all examples include loops, symbolic simulation could be applied only after unrolling the loops. Since Ellip examples include infinite loops, they were unrolled for only 1-iteration. Then the results are not complete. Rule-based verification could be successfully applied to all examples directly. In these experiments, the parameter L in Algorithm 7 and Algorithm 8 was set to 1. Since the methods explained in Section 4.4.1 and Section 4.4.5 were not implemented, equivalences of symbolic expressions were also checked only with simple replacements of equivalent expressions in equivalent classes.

The results show that symbolic simulation could verify the DCT and Ellip examples faster since there are no conditional branches. Rule-based verification checks all candidates of equivalence exhaustively. Moreover, when an equivalence of a candidate is proved, all other candidates are checked again since their equivalences can be proved with the information of the newly proved equivalence. Then, each candidate of equivalence may be checked multiple times. However, since symbolic simulation checks the equivalences of expressions at states from the initial state only once for each execution path, it is basically faster to verify designs without many conditional branches than rule-based verification. In addition, if there are loops in target designs, Rule 4 is applied for "$number\ of\ states\ in\ loops \times number\ of\ expressions$" times in the worst case. Then rule-based verification becomes much slower. However, even there are a lot of conditional branches in the IDCT examples, rule-based verification could verify them within relatively short times which are not so different from the other examples. Symbolic simulation could not verify them within 24 hours. This is because the complexity is square to the number of conditional branches in rule-based verification, and exponential in symbolic simulation.

From these experimental results, the following facts could be confirmed.

- The overall proposed method can successfully applied to real designs.
- Verification time of rule-based verification is not strongly affected by the num-

Table 4.2: Verification time of rule-based equivalence propagation

Example	Target	Symbolic simulation	Symbolic simulation with loop unrolling	Rule-based verification
DCT	(1) vs (2)	-	< 1s	2.4s
	(2) vs (3)	-	< 1s	3.1s
IDCT	(1) vs (2)	-	> 24h	24.3s
	(2) vs (3)	-	> 24h	30.8s
Ellip	(1) vs (2)	-	< 1s	7.5s
	(2) vs (3)	-	< 1s	7.9s

bers of conditional branches

- Rule-based verification can directly verify designs which include loops without unrolling the loops.

4.6 Conclusion

In this chapter, a word-level equivalence checking method in bit-level accuracy with synthesizing two designs with a same datapath was proposed. A new word-level rule-based comparison method was also proposed, and the experimental results showed that the proposed method is fast and it can verify some designs which cannot be verified by symbolic simulation. Since the proposed method is a rule-based method the range of verifiable designs can be expanded by introducing additional rules.

Chapter 5

Formal Verification of Hardware/Software Co-Design with Translation into FSMD

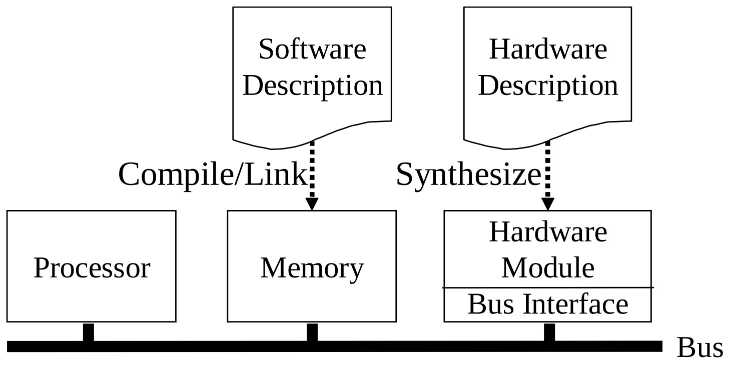


Figure 5.1: Typical hardware/software co-design implementation

5.1 Introduction

As mentioned in Section 1.1, currently one of the standard starting points of the flow shown in Figure 1.1 is the design stages that software is written as a program code and hardware is written as an HDL code in RTL. In such the design stages, hardware and software portions are verified together to check the whole functionality after verifying each part independently. Such the verification is more difficult than the independent verifications because of the following two difficulties.

One is the communication between hardware and software parts. To verify the whole functionality, models of the interface parts between hardware and software must be created. Figure 5.1 shows a typical implementation of hardware/software co-design. The hardware part is implemented as a specialized hardware module and connected to a bus through an interface module. The software part is stored in a memory to be executed by the processor. The hardware and software parts interact through the bus interface using memory mapped I/O or interruption. On the precise interface part model, the functionality of the processor, the bus controller, and the interface module must be described.

The other is the differences of languages and abstraction levels between hardware and software parts. For example, a program code written in ANSI-C does not have the notion of time since only the order of execution is defined in program code. On the other hand, a behavior at each clock cycle is written with HDL in RTL.

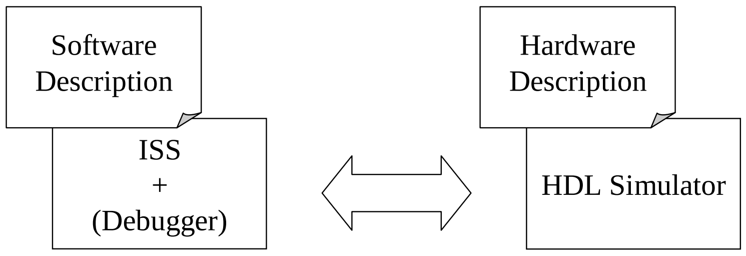


Figure 5.2: Hardware/software co-simulation

In hardware/software co-simulation, a widely used technology for hardware/software co-verification, those problems are solved with the simulation environment as shown in Figure 5.2. In such a simulation environment, independent simulation environments of software and hardware are connected and co-executed through a communication model. Software part is executed on a debugger or Instruction Set Simulator (ISS) after compilation and linking so that the precise behavior on the processor can be simulated. Hardware part is executed on an HDL simulator which can simulate the behavior after syntheses. Interface parts in Figure 5.1 are also used together to simulate whole the behavior. There are many commercial products for hardware/software co-simulation, such as Seamless[148] from Mentor Graphics.

One critical problem on hardware/software co-simulation is the simulation speed which is usually much (about 10000 times) slower than that of actual chips. Though acceleration methods [183, 141, 149, 72] have been widely researched, it is still impossible to simulate all possible input sequences exhaustively. Therefore, corner-case problem is also critical like verification of hardware or software.

As mentioned in Section 1.2, formal verification is a strong technique to apply exhaustive analysis independent from input sequences. However, it is not widely used for hardware/software co-designs because of the problems not only the two mentioned above but also their computation amount. Since hardware/software co-design includes hardware, software, and interface parts, its design size become mach larger than each part. Since the computation amount of formal method increases exponentially with design size, this problem is critical.

There are some existing works[176, 109] which try to solve those three problems. In [176], both hardware and software parts are translated into a common representation (S/N, the input language of a model checker COSPAN[67]). The interface part between hardware and software parts is represented by an abstract model where software and hardware portions are modeled as concurrent processes, memory-mapped I/O is modeled as accesses to shared variables, and interruption is modeled with an interruption controller model. In COSPAN, partial order reduction [166, 55] technique is used to accelerate the verification. In [109], abstraction-refinement framework for hardware-software co-design with assume-guarantee reasoning is also proposed. However, those methods cannot solve the above three problems completely since the verification performance is still not enough even using those methods. In addition, those methods target on program code written in Executable UML (xUML)[142] and it is not popular, and they also do not give detailed translation methods.

The biggest problem among three is the computation amount. One reason except design size is because the verification model is a concurrent model. Not only the hardware and software parts are modeled as different processes, each interruption sequence becomes additional process in [176, 109]. Then state explosion problem may be caused since execution orders among those processes must be considered. Partial order reduction can reduce the number of such states, but it is not effective when interactions between hardware and software occur frequently.

Based on the above discussion, a framework to apply formal verification to hardware/software co-design written in program code and HDL code in RTL is proposed in this chapter.

In the proposed framework, both hardware and software parts are translated into FSMD as the common representation. Since program code is much different from FSMD, the translation from program code to FSMD is explained in detail. During the translation into FSMD, interaction between hardware and software is also modeled with a minimal model. Memory mapped I/O is modeled as accesses to shared variables. The hardware part, the software part, and the interruption sequences are translated to different concurrent processes.

After the translation, a sophisticated state reduction technique is applied to reduce the number of execution orders among the concurrent processes which must be considered. First, synchronization points which restrict the execution order are detected. Such detection is performed with focusing on specific kinds of hardware-software interaction processes, such as polling loop and interruption. Then, conditions for the execution orders are generated from those synchronization points and solved with an extension of the approach proposed in [144] using a Satisfiability Modulo Theories (SMT) instead of Integer Linear Programming (ILP). This sequentialization step exhaustively generates sequential processes each of which is functionally equivalent to the set of original concurrent processes and satisfies the conditions. Finally, a state merging technique with data dependence analysis is applied to each of the generated sequential processes. The proposed methods are preprocesses before applying formal verification. Then, the proposed framework can use arbitrary back-end formal verification engines.

The differences from the existing works[176, 109] are as follows.

- The proposed method can handle designs whose software portion is written in a C-based languages.
- Details of the translation and modeling processes are explained. Then the proposed method can be directly applied to practical designs.
- The proposed state reduction technique works much better than existing methods, such as partial order reduction.

The organization of this chapter is as follows. First, in Section 5.2, existing techniques used in the proposed method are introduced as basic notions. Second, in Section 5.3, the proposed translation and state reduction methods are explained. Then in Section 5.4, some experimental results with practical examples are presented to show the effectiveness of the proposed method. Finally in Section 5.5, this chapter is concluded.

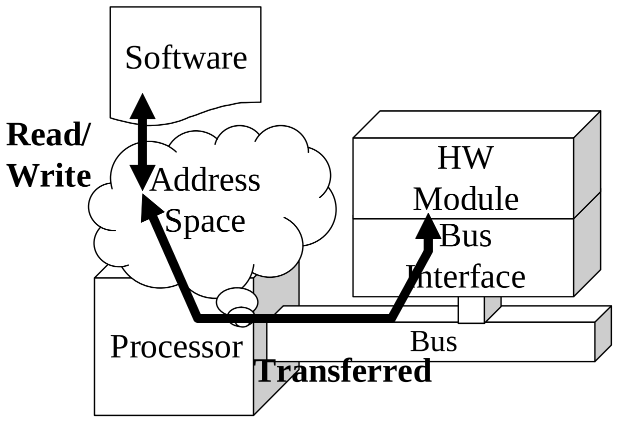


Figure 5.3: Memory Mapped I/O

5.2 Basic Notions

5.2.1 Modeling of Connection between Hardware and Software

The interactions between hardware and software on hardware/software co-design can be classified into memory mapped I/O and interruption.

As shown in Figure 5.3, in memory mapped I/O, some hardware resources are assigned to the address space of a processor, and data transmission is performed through the access to the addresses. Such functionality of memory mapped I/O can be modeled as accesses to shared variables by treating each hardware resource and corresponding assigned address as an identical resource[176]. Information of the correspondences between hardware resources and assigned addresses are required to apply such modeling. Since the actual components between hardware and software, such as processor and bus, are not included in the model, unnecessary size increase of the model can be avoided.

Figure 5.4 shows the functionality of interruption. When a interruption signal of a processor is triggered by a hardware, the processor stops the execution of software part, and runs a pre-defined interruption sequence. After the termination of the interruption sequence, the software part execution is resumed. The occurrences of interruptions can be controlled with masking specific processor registers or executing

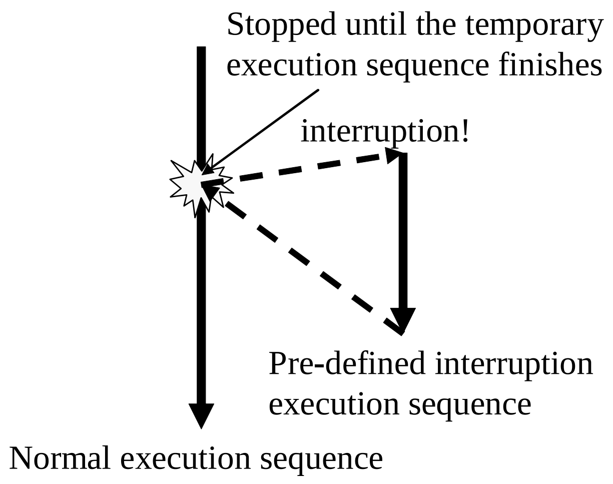

Stopped until the temporary execution sequence finishes
interruption!
Pre-defined interruption execution sequence
Normal execution sequence

Figure 5.4: Interruption Driven I/O

interruption enabling/disabling functions. While interruption is disabled with such a way, interruption does not occur even if the interruption signal is triggered. In [176], each interruption sequence is modeled as an independent concurrent process, and interruption signals are modeled as shared variables. An interruption scheduler process is added to control the executions of software and interruption processes, since software processes must be stopped while interruption processes are running.

In the proposed method, a similar modeling method is applied, but the additional scheduler for interruption is not inserted since the scheduling is considered at the sequentialization step. Since such a scheduler includes many dynamic behaviors, such as priority control of interruptions, it does not suit to formal verification. The proposed method handles those dynamic behaviors by converting them into static ones through the execution order reduction technique so that formal verifiers do not have to handle such dynamic issues.

5.2.2 Sequentialization of Concurrent Processes with Race Condition Verification

Race condition is a situation that the result of an access to a shared resource can be changed by the execution order of concurrent processes. Figure 5.5 shows a SpecC

```
int main(){
  int a = 0, b;
  par{
    a = 1;
    b = a;
  }
  return b;
}
```

Figure 5.5: Example SpecC code having race condition

code which has a race condition since the return value of **main** function can be changed with the execution order of two statements $\mathbf{a = 1}$ and $\mathbf{b = a}$ under a **par** statement which are executed concurrently.

In [114, 145], an equivalence checking method for designs which have concurrency is proposed. In their method, concurrent processes in a design are translated into a single sequential process as a pre-process of the equivalence checking with symbolic simulation. At the sequentialization step, synchronization verification including deadlock and race condition validation proposed in [144] is applied. The race condition validation checks the uniqueness of the execution order of each pair of basic blocks (sequence of statements which does not include conditional branch and synchronization) which access to a same shared resource. They check write/read and write/write accesses, and does not check read/read accesses since the value of the shared variable cannot be changed only with read accesses. The details of the race condition validation are as follows.

Let B denote a set of basic blocks in a design, W denote a set of **wait** statements, N denote a set of **notify** statements, and $Sync \subseteq W \times N$ denote synchronization relation which defines pairs of corresponding wait and notify statements. $T : W \cup N \rightarrow Z$, where Z is a set of integers, is a function which returns an execution time of a **wait** or **notify** statement. $T_b : B \rightarrow Z$ and $T_e : B \rightarrow Z$ are functions return the beginning time and the ending time of an argument basic block execution.

With the above definitions, formulae about execution timings are generated from design descriptions. For an arbitrary basic block $b \in B$, the next condition is true since the beginning time of a basic block must be earlier than the ending time of the basic block.

$$T_b(b) < T_e(b)$$

For two basic blocks $b_1, b_2 \in B$ which are executed sequentially, the next condition is true since the execution of the second basic block starts after the execution of the first basic block.

$$T_e(b_1) < T_b(b_2)$$

Similarly, if a **wait** or **notify** statement $s \in W \cup N$ and a basic block b are executed sequentially in the order $s \rightarrow b$, the next condition is true.

$$T(s) < T_b(b)$$

In the case that the execution order of the **wait** or **notify** statement and a basic block is $b \rightarrow s$, the next condition is true.

$$T_e(b) < T(s)$$

Since the execution timing of a **wait** statement is later than that of the corresponding **notify** statement, the next equation must be true.

$$\forall < w, n > \in Sync, T(w) > T(n)$$

With assuming all the above formulae are true, the satisfiabilities of the following two conditions are checked for each pair of basic blocks $< b_1, b_2 | b_1, b_2 \in B >$ accessing to a same shared variable to check the uniqueness of the execution order of them.

$$T_b(b_1) > T_e(b_2)$$
$$T_e(b_1) < T_b(b_2)$$

The first condition represents that the beginning time of b_1 is later than the ending time of b_2, in other words, b_1 is executed after the execution of b_2 finished. The second condition also represents that the ending time of b_1 is earlier than beginning time of b_2, in other words, b_1 execution always finishes before executing b_2. This process can be performed with Integer Linear Programming (ILP) solvers. In the case that one of the two conditions is satisfiable and the other is not, the execution order of those two basic blocks is proved to be unique. When both the conditions are satisfiable, the execution order is not unique and race condition occurs. The case that both of them are unsatisfiable cannot happen. When the execution order of all basic blocks in a concurrent design are proved to be unique, by sorting those basic block in the order, we can sequentialize the design into a single process without changing its functionality.

An example of sequentialization is shown with the code in Figure 5.6. This example has two concurrent behaviors **A** and **B** which are synchronized with a pair of **wait** and **notify** statements. Though these behaviors access to a same shared variable **x**, the synchronization avoids race condition. From the code, the following formulae are generated with the method explained above.

$$T_b(b_0) < T_e(b_0)$$
$$T_b(b_1) < T_e(b_1)$$
$$T_b(b_2) < T_e(b_2)$$
$$T_e(b_0) < T(w_1)$$
$$T_e(b_0) < T_b(b_2)$$
$$T(w_1) < T_b(b_1)$$
$$T_e(b_2) < T(n_1)$$
$$T(n_1) < T(w_1)$$

With assuming those formulae are true, the following two conditions are checked since b_1 and b_2 access to a same shared variable **x**.

$$T_b(b_1) > T_e(b_2)$$
$$T_e(b_1) < T_b(b_2)$$

In this case, only the first condition is satisfiable. Therefore a sequentialized code shown in Figure 5.7 is generated.

In the proposed method, this sequentialization is applied after detecting synchronization points with using the information of those points. However, the above method cannot be directly applied for the purpose since designs to be sequentialized

```
main() {
  x = 0; // b0
  par {
    A.main();
    B.main();
  }
}
behavior A {
  main(){
    wait(e); //w1
    a = x + 10; // b1
  }
}
behavior B {
  main() {
    x = 20; // b2
    notify(e); //n1
  }
}
```

Figure 5.6: Example SpecC code for sequentialization

```
main() {
  x = 0; // b0
  x = 20; // b2
  a = x + 10; // b1
}
```

Figure 5.7: Sequentialized code from Figure 5.6

may include interruptions. Then the above method is extended to take interruption priorities into account. Since the extended method uses logic of integer theory, Satisfiability Modulo Theories (SMT) solvers are used instead of ILP solvers. In the proposed method, the existing sequentialization method is also modified to generate all possible execution orders giving different functional results, instead of giving up sequentialization.

5.3 Formal Verification Method Based on Translation into FSMD

Figure 5.8 shows the proposed verification flow. Inputs of the flow are a hardware part and a software part of a hardware/software co-design which are written in program code and RTL code, respectively.

First, the communication between hardware and software is abstracted with a method similar to the one introduced in Section 5.2.1, and then the both parts are translated into FSMDs. Next, synchronization points in the design are detected among those FSMDs. With the information, a sequentialization method is applied, and a set of sequential FSMDs is generated. Finally, a state merging technique based on data dependency analysis is applied to reduce the number of states in the FSMD set. Existing formal verification methods, such as model checking and equivalence checking, can be directly applied to the FSMD set.

In this section, the details of each step of the flow in Figure 5.8 are explained. In addition, the proposed method is compared with partial order reduction, and the restriction on applying the proposed method is discussed. In the following sections, it is assumed that the software program code is written in ANSI-C. However, the same method can be also applied to other programming languages since the proposed method does not have language specific.

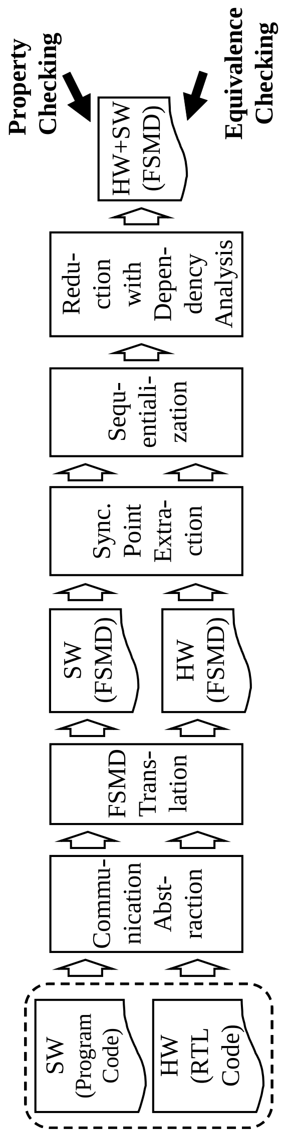


Figure 5.8: Proposed Verification Flow

Table 5.1: Memory map for the design in Figure 5.9

Address	HW resource name
0x2000000	in1
0x2000010	in2
0x2000020	out
0x2000030	str

5.3.1 Abstraction of Communication between Hardware and Software

As the first step, the communication between hardware and software parts is abstracted by a method similar to [176] introduced in Section 5.2.1.

In the proposed method, correspondences between addresses and assigned hardware resources are described as a memory map. A memory map is a table shows the correspondences. Such a memory map must be given by a user, but it is easy since such correspondences are usually obvious. For example, in an example software ANSI-C code shown in Figure 5.9, four addresses are assigned to hardware resources and the accesses to those addresses are mapped to the virtual symbols **HW_IN1, HW_IN2, HW_OUT, HW_STR** with **define** statements. This software code communicates with an example hardware Verilog-HDL code shown in Figure 5.10. Each of the virtual symbols in the software code has a corresponding hardware port. This relation is represented in the memory map shown in Table 5.1.

With a memory map, the accesses to the addresses can be easily replaced with symbols corresponding to hardware resources. As a result, the accesses to the addresses are represented by with the accesses to shared variables. In the 5.9, following variable replacements are applied.

$$HW\_IN1 \rightarrow in1$$
$$HW\_IN2 \rightarrow in2$$
$$HW\_OUT \rightarrow out$$
$$HW\_STR \rightarrow str$$

Then the software code in Figure 5.9 is modified to the code shown in Figure 5.11.

```
#define HW_IN1 ((volatile char*)(0x20000000))
#define HW_IN2 ((volatile char*)(0x20000010))
#define HW_OUT ((volatile char*)(0x20000020))
#define HW_STR ((volatile char*)(0x20000030))
char i1, i2, i3, i4, o1, o2, done;
char proc(){
  local_irq_disble();
  HW_IN1 = i3;
  HW_IN2 = i4;
  done = 0;
  STR = 1;
  local_irq_enable();
  o1 = i1 * i2;
  o1 = o1 + 1;
  while(!done);
    return o1 + o2;
}
void handler(){
  o2 = HW_OUT;
  done = 1;
}
```

Figure 5.9: Example Software code in C

In the proposed method, interruption sequences are treated as independent processes. Here, the number that each interruption occurs must be given by users, and each process is duplicated for the number of times for static analysis. In the code in Figure 5.9, a function **handler** is an interruption sequence. In the following sections, the maximum number of interruption occurrences of **handler** is assumed to be 1.

5.3.2 Translation into FSMD

After the communication abstraction explained in the previous section, both software and hardware parts are translated into FSMDs.

In the translation of the software part, syntax elements which cannot directly be translated into FSMD are removed beforehand. They include pointers, recursive

```
module multi(in1, in2, out, str, nirq, clk, rst);
  input [7:0] in1, in2;
  output [7:0] out;
  input str, clk, rst;
  output nirq;
  reg state;
  assign out = (state == 0) ? 0 : in1 * in2;
  assign nirq = (state == 0) ? 1 : 0;
  always@(posedge clk)begin
    if(rst)
      state <= 0;
    else case(state)
      0: if(str)
           state <= 1;
      1: state <= 0;
      endcase
  end
endmodule
```

Figure 5.10: Example hardware code in Verilog-HDL

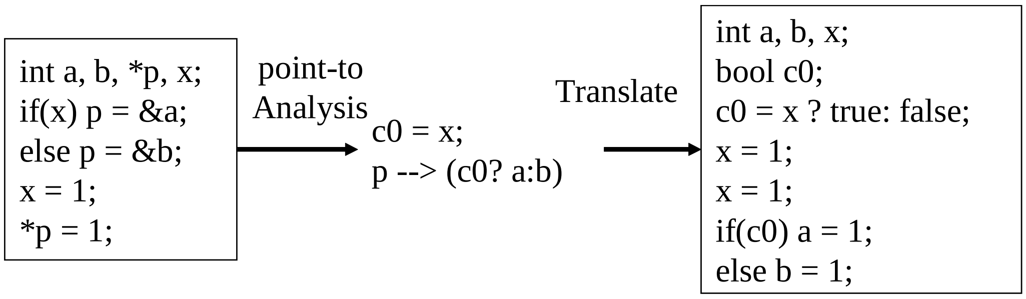
function calls, and dynamic memory allocations. Pointers are removed with point-to analysis[70] which analyzes the pointed resources as shown in Figure 5.12.

The translation step is performed by translating one statement in a program code into one state in FSMD. Each variable and conditional branch in the code are translated into a data variable and a set of multiple state transitions from a same state, respectively. Executions of interruption control functions are removed, but the original executed places are memorized to utilize in the synchronization point detection step. Each interruption sequence is translated into multiple identical concurrent FSMDs, and its number of copies is specified by users as the number of interruption occurrences.

For example, the code in Figure 5.11 is translated into the FSMDs in Figure 5.13 (a) and (b). Figure 5.13 (a) corresponds to **proc**, and (b) corresponds to **handler**. **proc** is the main function and **handler** is an interruption sequence. **local_irq_disable** and **local_irq_enable** are interruption control functions which

```
char i1, i2, i3, i4, o1, o2, done;
char proc(){
  local_irq_disble();
  in1 = i3;
  in2 = i4;
  done = 0;
  str = 1;
  local_irq_enable();
  o1 = i1 * i2;
  o1 = o1 + 1;
  while(!done);
    return o1 + o2;
}
void handler(){
  o2 = out;
  done = 1;
}
```

Figure 5.11: Software code after memory map replacement from Figure 5.9



```
int a, b, *p, x;
if(x) p = &a;
else p = &b;
x = 1;
*p = 1;
```

```
c0 = x;
p --> (c0? a:b)
```

```
int a, b, x;
bool c0;
c0 = x ? true: false;
x = 1;
x = 1;
if(c0) a = 1;
else b = 1;
```

Figure 5.12: Pointer removal with point-to analysis

enable and disable interruption, respectively. The area surrounded by the dashed lines shows the portion where an interruption may occur.

Hardware parts are also translated into FSMD. As mentioned in Section 2.3.1, since FSMD is in the same abstraction level as RTL design, it is easy to translate to each other. Figure 5.13 (c) shows an example hardware FSMD translated from the Verilog-HDL code in Figure 5.10. The original HDL code has two states, and they

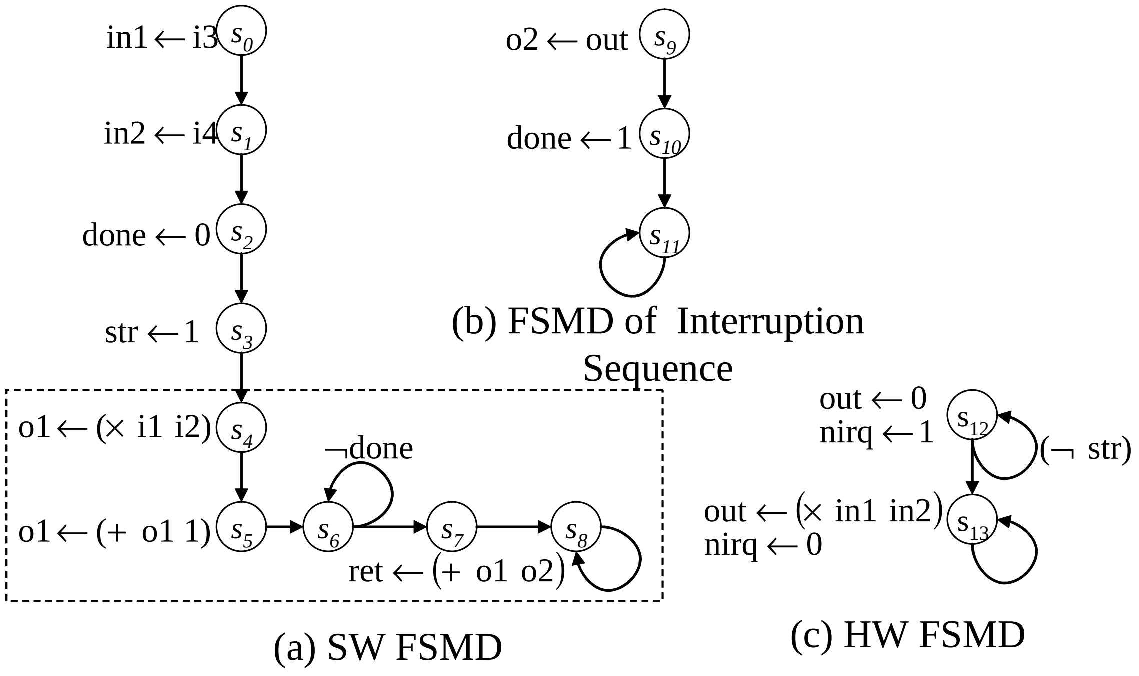


Figure 5.13: Example FSMDs of HW/SW Co-Design

are directly converted to the states in FSMD. Here, **nIRQ** is an interruption signal.

Before applying the later steps, FSMDs must be unrolled since the proposed sequentialization method cannot handle loops. This unrolling is applied to loops which are not polling loops. A polling loop detection method is described in the next section.

5.3.3 Definition of Synchronization Point and Its Detection

Synchronization point is a pair of states $(s_1, s_2 | s_1, s_2 \in S)$ where s_2 is reached only after s_1 is executed. This relation between two states corresponds to that of wait and notify statements in SpecC, SystemC, and Java.

Since the sequentialization technique[145] introduced in Section 5.2.2 targets on wait and notify statements in SpecC, a similar method can be used for FSMD with introducing synchronization points.

In the translated FSMDs, synchronization points can be detected as the following four types of pairs.

- The state (s_2) after a polling loop, and a state (s_1) which triggers the signal waited in the polling loop
- The first state (s_2) of an interruption process, and a state (s_1) which triggers the interruption signal.
- The first state (s_2) of an interruption process, and the last state (s_1) of an interruption disabled area,
- The first state (s_2) of an interruption process copy, and the last state (s_1) of the previous interruption copy.

The first type corresponds to synchronization by polling. The second type represents the relation that an interruption sequence starts when the corresponding interruption signal is triggered. The third type represents the relation that an interruption only happens when it is enabled. Since other synchronization methods, such as hand-shaking, are combinations of polling, they correspond to a set of the first type pairs. The last type represents that the FSMDs representing a same interruption process are not executed simultaneously.

To determine synchronization points automatically, polling loop is defined as follows.

Definition 11 (Polling loop)**.** Assume that there is a looped transition $r_1 = (s_1, s_1) \in R$ to a state $s_1 \in S$, and its transition condition $Q(r_1)$ is an expression $e = L_G(Q(r_1)) \in E$. Let $E_P \subseteq E$ denote a set of symbols or a negation of a symbol such that

$$E_P = I \cup V \cup O \cup \{f_{call} = (\neg, e_1) | f_{call} \in F_{call}, e_1 \in I \cup V \cup O\}$$

Let $A_P = \{a | a \in A, L_A(a) = (e_1, e_2), e_1 = e_2\}$ also denote a set of assignments each of which maintains the same value.

Then, when all assignments to data registers on s_1 are included in A_P such that

$$\forall a \in A_V, ((s_1, a) \in P) \rightarrow (a \in A_P)$$

and $e \in E_P$, then the state s_1 is in a polling loop, The set of states $S_{next} \subseteq S$ such that

$$S_{next} = \{s|(s_1, s) \in R, s \in S, s_1 \neq s\}$$

is the set of the state after the polling loop.

When there is an assignment which assigns a value to make e *true* in another concurrent FSMD, it triggers the polling.

In the proposed method, polling loops are detected with this definition. Obviously, only polling loops in the simplest form that only wait for a single signal can be detected, and no operations are performed while the loop. However, this restriction is enough, since the sequentialization method can handle only when each synchronizing pair is one-to-one relation.

Correspondence between interruption trigger signals and interruption sequences, and functions that control interruptions are specified by users. Then, synchronization points related to them are automatically detected by finding assignments which trigger the signals and function calls of interruption controlling functions.

Synchronization points of the FSMDs in Figure 5.13 are shown in Figure 5.14. Dotted arrows show synchronization points. There are four synchronization points, two from polling loops, one from interruption, and one from interruption enabled area. Since the number of interruption occurrences is one, there are no synchronization points which are classified to the fourth type.

Since the data transportation behavior of the design in Figure 5.14 may not be practical, an example design using a commonly used hardware protocol, Open Core Protocol(OCP)[125], for data transportation is shown in Figure 5.15. This design transfers a data (**MY_DATA**) from the master to the slave with using OCP write protocol. **MCmd** and **MAddr** are connected to the bus command lines and address lines, respectively, which are outputs from the master module. **MY_ADDR** is the address of the slave module. **MData** is connected to the bus data lines. **SCmdAccept** is connected to a bus line which represents the acceptance signal from slave modules. **Reg** is a data resister in the slave module and data received from the master module is stored. The two dotted arrows show synchronization points, and both of them are from polling loops. This example shows that the proposed method can be applied to practical protocols.

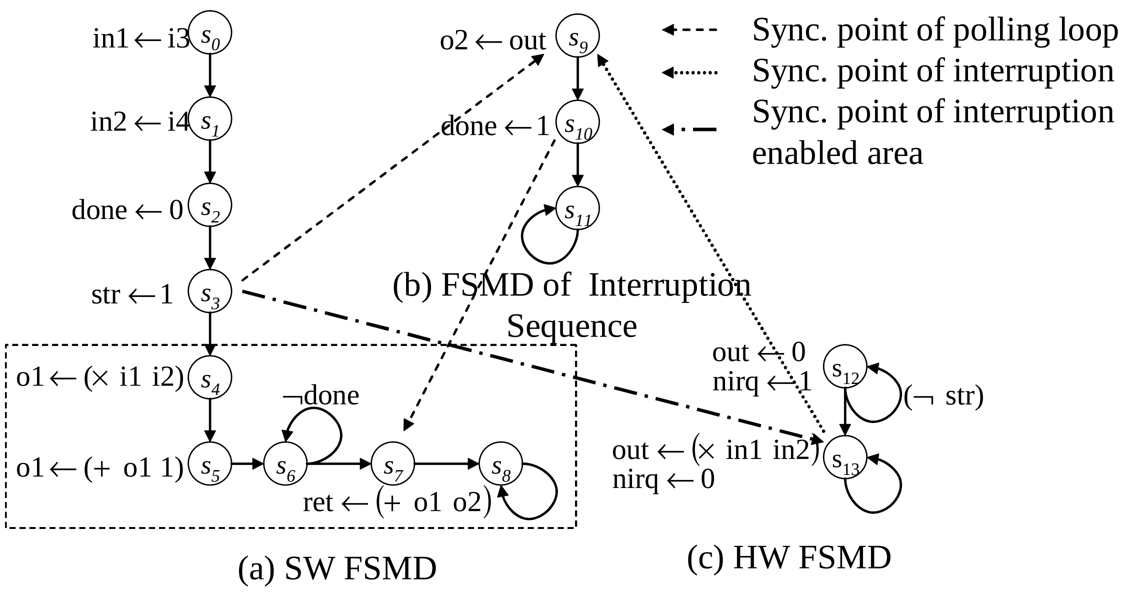


Figure 5.14: Synchronization Points on the FSMDs in Figure 5.13

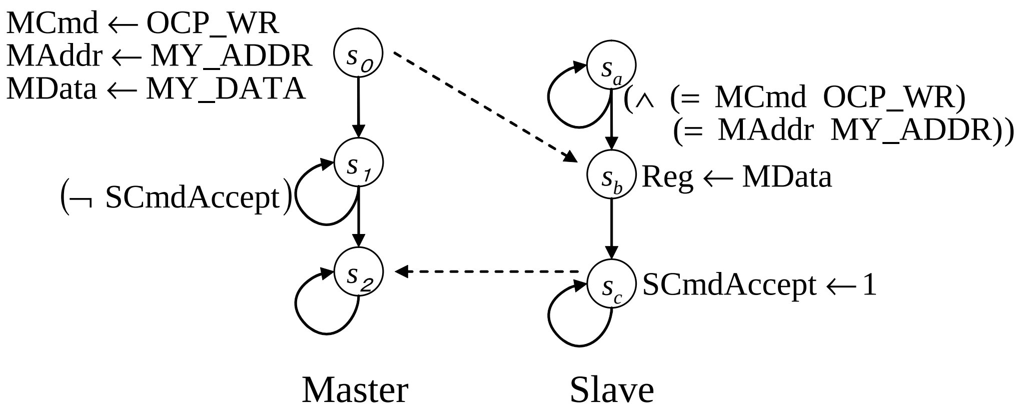


Figure 5.15: Design using OCP and its synchronization points

5.3.4 Applying Sequentialization

With the synchronization points detected in the previous section, sequentialization[114, 145] which has introduced in Section 5.2.2 can be applied to FSMD. Since a synchronization point corresponds to a pair of corresponding **wait** and **notify** statements in SpecC, a similar method can be applied. However, the original sequentialization method[114, 145] cannot be applied directly since it cannot handle interruptions. In addition, since the original method is for equivalence checking of SpecC designs, it does not suit to general formal verification of FSMDs. Therefore, the original method is modified in the following points.

- In the formulation, a single execution time is assigned to each sequential portion (basic block) instead of a beginning time and an ending time.
- Additional conditions defining the behaviors of interruptions are added to the formula to be solved. Since the added conditions include logical operators, the updated formula is solved by SMT solvers.
- Though [114, 145] check each race condition candidate separately, the proposed method checks all race condition candidates at once.
- Though [114, 145] stop the generation of a sequentialized process when race conditions are detected, the proposed method generates all sequential processes which have different execution results in such cases.

Removal of Synchronization Related Portions Before applying the sequentialization, synchronization related portions, such as polling loops, and accesses to waited variables and interruption signals are removed. Since the information of synchronization is already known in this stage, such portions are redundant for the verification.

Figure 5.16 shows an example after this step is applied. Polling loops at s_6 and s_{12}, assignments to the waiting signal of the polling loops and interruption signals in s_2, s_3, s_{10}, s_{13} are removed from Figure 5.13.

Generation of Timing Conditions To apply sequentialization, race conditions among concurrent FSMDs must be checked. As shown in Section 5.2.2, it is performed by generating and checking conditions of execution timings and orders. Before generating conditions, two functions represent an execution timing of a sequence of states and an execution order of two sequences of states, respectively, are introduced.

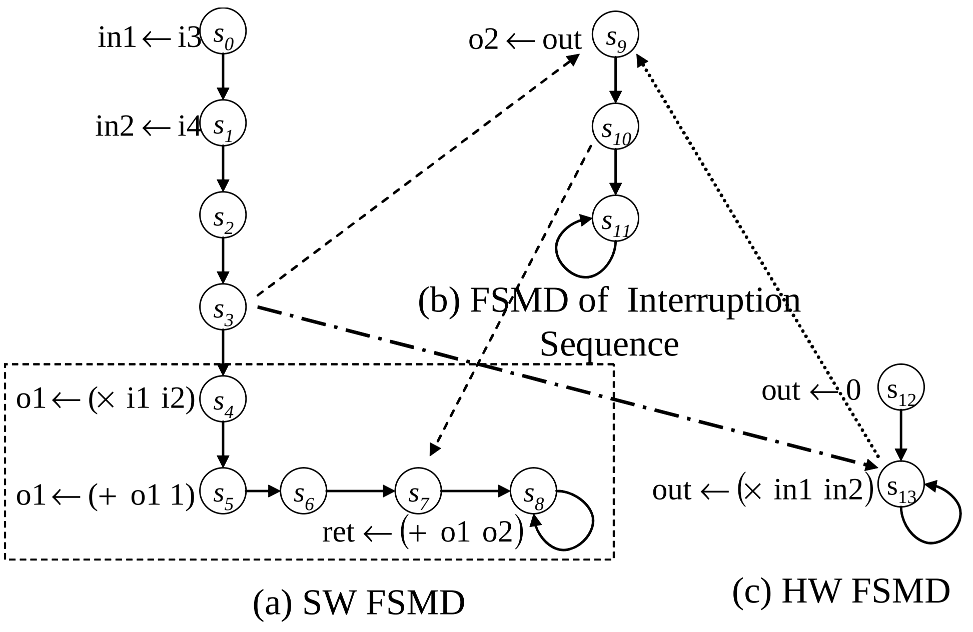


Figure 5.16: Synchronization related portion removal from Figure 5.13

First, Let $time : T \rightarrow Z$, where Z denote a set of integers, denote a function which returns the execution timing of a given sequence of states. The argument sequences of states cannot have accesses to shared symbols or synchronization points with other FSMDs more than once. This corresponds to **Basic block (BB)** in [145]. In the conditions to be generated, such sequences of states are treated as units. From the idea of partial order reduction[166, 55], executions only using local symbols do not affect executions of other concurrent processes. For example, in Figure 5.16, (s_1, s_2) is a sequence of states which accesses to a shared symbol (in_2) only once, and an executions of s_2 does not affect other FSMDs' execution results. In the proposed method, since the accurate execution timing is not necessary, execution time of each sequence of states is fixed to 1 to give the condition of execution timing easily.

Second, let $order : T \times T \rightarrow \{true, false\}$ denote an execution order of two sequences of states. The return value is $true$ when the first argument sequence of states is executed before the second argument sequence of states, and otherwise it is $false$. For example, when (s_0) is executed before (s_1, s_2), $order((s_0), (s_1, s_2))$ is *true*.

In this method, the following five types of conditions are generated.

- Range of execution timing and exclusiveness of those timings
- Relation between execution timings and execution orders
- Local execution order in each FSMD
- Synchronization
- Priorities of interruptions

The first condition represents the range of execution timings. Since an execution time of each sequence of states is 1, all the execution timings are between 0 and $n-1$ when the number of sequences of states is n. For example, the following condition is generated from the design in Figure 5.16.

$$\begin{aligned}
&(0 \leq \mathit{time}((s_0)) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_1, s_2)) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_3, s_4, s_5, s_6)) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_7, s_8)) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_9)) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_{10}, s_{11})) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_{12})) \leq 7) \wedge \\
&(0 \leq \mathit{time}((s_{13})) \leq 7)
\end{aligned}$$

Since all concurrent FSMDs are executed concurrently, two sequences of state cannot be executed at a same timing. Then execution timings must be exclusive. For example, the following condition is generated from the design in Figure 5.16.

$$\begin{aligned}
&\mathit{time}((s_0)) \neq \mathit{time}((s_1, s_2)) \neq \mathit{time}((s_3, s_4, s_5, s_6)) \neq \\
&\mathit{time}((s_7, s_8)) \neq \mathit{time}((s_9)) \neq \mathit{time}((s_{10}, s_{11})) \neq \\
&\mathit{time}((s_{12})) \neq \mathit{time}((s_{13}))
\end{aligned}$$

The second condition represents the relation between execution timings and execution orders. When the execution timing of one sequence of states is earlier than the execution timing of another sequence of states, the execution order of the first sequence of states is also earlier than the second sequence of states. Same as the method in [145], only the execution orders of portions accessing same shared resources which are candidates of race conditions have to be considered. Execution orders of the other portions are not related to the execution results of the design. For example, since shared variables are $in1, in2, out$ in the FSMDs in Figure 5.16, orders have to be considered are $order((s_0),(s_{13}))$, $order((s_1,s_2),(s_{13}))$, $order((s_7,s_8),(s_9))$, $order((s_9),(s_{12}))$, $order((s_9),(s_{13}))$. Then the following conditions can be generated.

$$\begin{aligned}
&(time((s_0)) < time((s_{13}))) \leftrightarrow order((s_0),(s_{13})) \wedge \\
&(time((s_1,s_2)) < time((s_{13}))) \leftrightarrow order((s_1,s_2),(s_{13})) \wedge \\
&(time((s_7,s_8)) < time((s_9))) \leftrightarrow order((s_7,s_8),(s_9)) \wedge \\
&(time((s_9)) < time((s_{12}))) \leftrightarrow order((s_9),(s_{12})) \wedge \\
&(time((s_9)) < time((s_{13}))) \leftrightarrow order((s_9),(s_{13}))
\end{aligned}$$

The third type of the condition just shows that states in a sequence of states in an FSMD are executed sequentially. For a sequence of state transitions $t = (s_0, s_1, \cdots, s_n) \in T$, the following condition is generated.

$$\bigwedge_{j=0}^{n-1} (time((s_j)) < time((s_{j+1})))$$

When some of them are grouped as explained above, such a sequence is treated as a single state in the above condition. The same condition can be generated for a sequence of sequences of state transitions, instead of a sequence of state transitions. For example, from the design in Figure 5.16, the following condition is generated.

$$\begin{aligned}
&time((s_0)) < time((s_1,s_2)) \wedge time((s_1,s_2)) < time((s_3,s_4,s_5,s_6)) \wedge \\
&time((s_3,s_4,s_5,s_6)) < time((s_7,s_8)) \wedge \\
&time((s_9)) < time((s_{10},s_{11})) \wedge
\end{aligned}$$

$$time((s_{12})) < time((s_{13}))$$

The fourth type of the condition shows the relation in synchronization points. Since the execution order between the two states in each synchronization point is fixed, the following condition is generated from each synchronization point $(s_1, s_2) \in S^2$.

$$time((s_2)) < time((s_1))$$

When such a synchronizing state is included in a sequence, then, timing of the state is replaced with that of the sequence. For example, the following condition is generated from Figure 5.16.

$$\begin{aligned}&time((s_3, s_4, s_5, s_6)) < time((s_9)) \wedge \\ &time((s_3, s_4, s_5, s_6)) < time((s_{13})) \wedge \\ &time((s_{10}, s_{11})) < time((s_7, s_8)) \wedge \\ &time((s_{13})) < time((s_9))\end{aligned}$$

The last type of condition represents the specification of interruption. As explained in Section 5.2.1, when interruption occurs, the current execution of software or interruption sequence suspends and another interruption sequence starts. After the interruption sequence finishes, the previous sequence is restarted from the suspended point. A priority is assigned to each interruption, and only interruptions having equal or higher priorities can occur when another interruption sequence is executed. In addition, the portions where interruptions can occur can be controlled by interruption enable/disable functions and register maskings. Let $(s_{11}, s_{12}, \cdots s_{1m}) \in S$ be a sequence of states in a software or interruption sequence where interruption is enabled. Let $(s_{21}, s_{22}, \cdots s_{2n}) \in S$ be a sequence of states in another interruption sequence which has equal or higher priority than the previous one. Assume that all states are in the different groups. When execution of the later sequence starts, the former sequence must stop until the execution finishes. Then, the following condition is generated.

$$\bigwedge_{j=0}^{n-1} (time((s_{21})) < time(s_{1j})) \rightarrow (time((s_{2n})) < time(s_{1j}))$$

This condition shows that when the first state of an interruption sequence is executed before a state of a lower priority sequence, then the last state of the interruption sequence must be executed before the state of the lower priority sequence. This condition is generated for each pair of FSMDs where interruptions between them can occur. For example, the following condition is generated from Figure 5.16.

$$(\mathit{time}((s_9)) < \mathit{time}((s_3, s_4, s_5, s_6))) \rightarrow (\mathit{time}((s_{10}, s_{11})) < \mathit{time}((s_3, s_4, s_5, s_6))) \wedge$$
$$(\mathit{time}((s_9)) < \mathit{time}((s_7, s_8))) \rightarrow (\mathit{time}((s_{10}, s_{11})) < \mathit{time}((s_7, s_8)))$$

Generation of Sequential Description with Sequentialization Satisfiability of the conjunction of the generated conditions is checked with an SMT solver with treating function calls of *time* and *order* as variables. When the result is unsatisfiable, there are no execution orders which satisfy the synchronization conditions. It means that the design has deadlocks. On the other hand, when the result is satisfiable, there is at least one execution order which satisfies the synchronization conditions so that the design can be sequentialized. Some SMT solvers, such as Yices[179], can generate a set of assignments to variables which satisfies the conditions. The assigned values to the function *order* show the execution orders to be considered, and the assigned values to the function *time* show a set of concrete execution timings after sequentialization. Then, a set of function *time* values represents an example of sequentialized execution order which satisfies the values of the function *order*. For example, the conditions generated from the design in Figure 5.16 is satisfied with the following variable assignments.

$$\begin{aligned}
\mathit{time}((s_0)) &= 1\\
\mathit{time}((s_1, s_2)) &= 2\\
\mathit{time}((s_3, s_4, s_5, s_6)) &= 3\\
\mathit{time}((s_7, s_8)) &= 7\\
\mathit{time}((s_9)) &= 5\\
\mathit{time}((s_{10}, s_{11})) &= 6\\
\mathit{time}((s_{12})) &= 0\\
\mathit{time}((s_{13})) &= 4
\end{aligned}$$

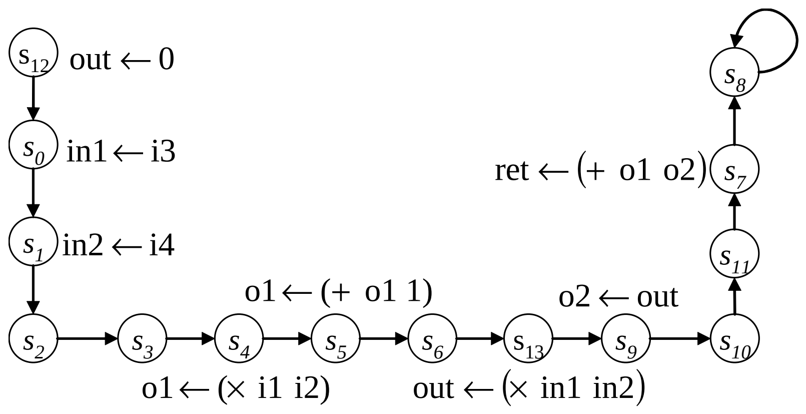


Figure 5.17: Sequentialization Result of the FSMDs in Figure 5.13

$$\begin{aligned}
order((s_0),(s_{13})) &= true\\
order((s_1,s_2),(s_{13})) &= true\\
order((s_7,s_8),(s_9)) &= false\\
order((s_9),(s_{12})) &= false\\
order((s_9),(s_{13})) &= false
\end{aligned}$$

Figure 5.17 shows the FSMD after sequentialization with this result.

Figure 5.18 (a) also shows the result of sequentialization for the design using OCP shown in Figure 5.15.

Generating all sequential designs having different execution results When there are race conditions in a design, the design behavior may change with the execution orders of accesses to shared variables. The proposed method generates all sequential designs giving different execution results in such cases. Execution orders of the accesses to shared variables are shown by function *order* values. Different combinations of function *order* values show execution orders which may give different execution results. Therefore, when a set of assignments to function *order* is generated by an SMT solver, a new set of assignments giving a different execution result can be generated by adding a constraint not to generate the same set of assignments to function *order*. When another set of assignments is obtained, there is a race condition in the design, and a different execution result can be obtained

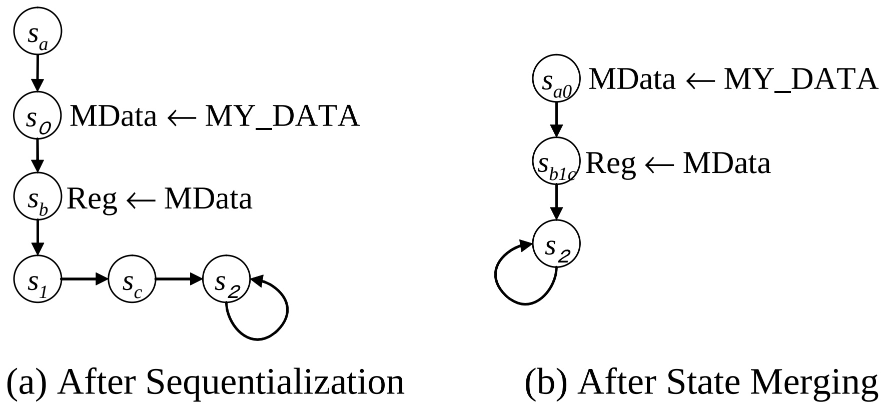


Figure 5.18: Sequentialization and State Merging Results of the Design using OCP in Figure 5.15

with the set of assignments to *order*. By applying this condition refinement iteratively until no additional set of assignments is generated, all execution orders giving different execution results can be obtained.

For example, in the case of the design in Figure 5.16, the following condition is added to the generated conditions, and satisfiability of the updated condition is checked.

$$\begin{aligned} \neg(\quad & order((s_0),(s_{13})) \land \\ & order((s_1,s_2),(s_{13})) \land \\ & \neg order((s_7,s_8),(s_9)) \land \\ & \neg order((s_9),(s_{12})) \land \\ & \neg order((s_9),(s_{13}))) \end{aligned}$$

Since the updated condition is unsatisfiable, the design in Figure 5.16 does not have any race conditions, and only one sequential design is generated.

5.3.5 State Merging with Data Dependence Analysis

Since one state transition in FSMD corresponds to one clock cycle, computation amounts of formal verification methods generally increase exponentially with the

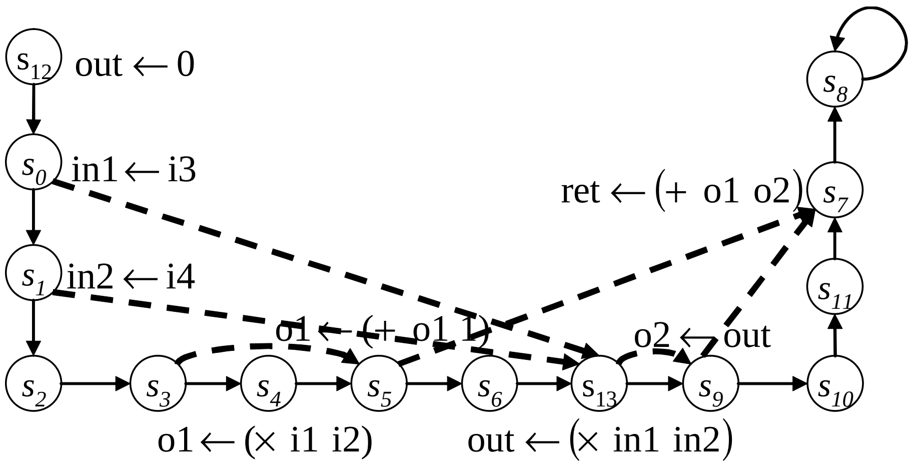


Figure 5.19: Data Dependences of the FSMD in Figure 5.20

number of states in FSMD. Then, the computation amount can be reduced dramatically by merging multiple states to a single one. This step can be applied only to a sequential FSMD since executions of other FSMDs' states can be inserted between the executions of two consecutive states in an FSMD in concurrent FSMDs.

To keep the execution results same, only states without data dependence can be merged. This data dependence is the same notion as data dependence edge in SDG introduce in Section 2.3.3, such as the value of a variable using at a state can be assigned at the other state. Then for each data register or output symbol in left sides of assignments, a data dependence edge is drawn to the state from states where value can be assigned to the symbol. For example, data dependence in the FSMD shown in Figure 5.17 is as shown in Figure 5.19.

Two states which are the source and the destination of a data dependence edge cannot be merged, since those states have accesses to a same symbol. On the other hand, other states can be merged. The state merging can be applied not only between two states but also among more than two states. For example, in the FSMD shown in Figure 5.19, $s12$, s_0, s_1, s_2, s_3, and s_4 can be merged. But s_5 cannot be merged with s_3 since there is a data dependence edge from s_3 to s_5. Figure 5.20 shows a result of the state merging step applied to the FSMD shown in Figure 5.17. The number of ways of state merging is not just one. This example merged states

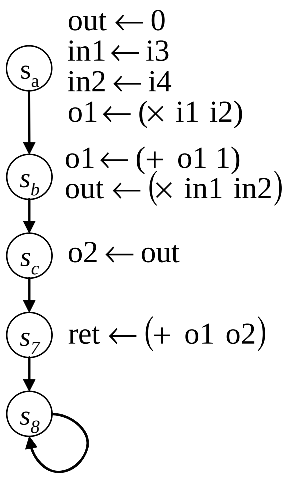


Figure 5.20: State Merging Result of the FSMD in Figure 5.17

from the initial state. Merged states are as follows.

$$\{s_{12}, s_0, s_2, s_3, s_4\} \rightarrow s_a$$
$$\{s_5, s_6, s_{13}\} \rightarrow s_b$$
$$\{s_9, s_{10}, s_{11}\} \rightarrow s_c$$

Figure 5.18 (b) also shows a result of the state merging for the design using OCP shown in Figure 5.18 (a).

5.3.6 Comparison with Partial Order Reduction

Partial order reduction[166, 55] is a state reduction technique for concurrent processes, and also used in the formal verification for hardware/software co-design[176]. Though partial order reduction is originally for the explicit method[34] which has been introduced in Section 2.1.1, an application for symbolic model checking[117] is also proposed in [107]. Static approach which is applied as a preprocess is also proposed in [104].

In concurrent processes, the execution order between portions only accessing to local variables is guaranteed not to affect to the execution result. Partial order reduction does not generate states correspond to another execution order. Then, partial order reduction can be effectively applied when the design accesses to shared variables infrequently. However, if the number of such interactions increases, its efficiency decreases.

On the other hand, the proposed sequentialization based method can be effectively applied even if the number of accesses to shared variables are large since the efficiency of the method depends on the number of possible execution orders. Since the number of synchronization points can be considered as propositional to the number of interactions, the number of possible execution order decreases if the number of interactions increases. Even in the case that number of interaction is small, same as partial order reduction, the proposed method is effective since the execution orders between basic blocks which access only to local variables are not considered.

For example, when applying partial order reduction to the FSMDs shown in Figure 5.13. Shared variables are **i3**, **i4**, **done**, **str**, and **o2**. Numbers of portions separated by accesses to shared variables in the software FSMD, in the interruption sequence FSMD, and in the hardware FSMD are, 6 ($\{s_0\}$, $\{s_1\}$, $\{s_2\}$, $\{s_3\}$, $\{s_4, s_5, s_6\}$, $\{s_7, s_8\}$), 2 ($\{s_9\}$, $\{s_{10}, s_{11}\}$), 2 ($\{s_{12}\}$, $\{s_{13}\}$), respectively. Then the number of execution orders considered in partial order reduction is as follows.

$$\frac{(6+2+2)!}{6! \times 2! \times 2!} = 1260$$

On the other hand, only one execution order must be considered in the proposed method as shown in Figure 5.17. In addition, number of states can be reduced more with the state merging, and the FSMD finally generated has only 5 states as shown in Figure 5.20.

5.3.7 Limitation

There are some limitations in the proposed method.

First, the proposed method assumes that interruptions occur only between the executions of consecutive statements of program codes. However, since a single statement can be compiled into multiple orders of a processor, interruptions can happen during the execution of a single statement. Moreover, interruptions can occur even during a single order of a processor, it is quite difficult to handle such interruptions completely. Therefore, to guarantee the verification correctness, users must guarantee that all execution results with such interruptions are included in the execution results of the generated sequential designs. This limitation must be also common in all methods which can handle interruptions.

Second, numbers of interruption are limited in the proposed method since interruption sequences must be duplicated for the numbers of interruption occurrences. The proposed method also becomes inefficient when the number of interruption increases, since interruption sequences are duplicated for the number of times.

Third, when synchronization points are under complex control flow, such as many conditional branches, sequentialization[145, 114] cannot be applied. In the proposed method, those synchronization points are ignored. Then, some accesses to a shared variable are not proved to be race condition free. Therefore, the numbers of generated sequential FSMD will increase in such cases.

Forth, the proposed method cannot verify some types of properties. In model checking with partial order reduction, only local variables can be used in properties. In addition, X operator in Linear Temporal Logic (LTL) cannot be used. The proposed method also has some limitations about property as follows.

- Properties must not include the notion of constant cycles (ex. X operator in LTL).
- Properties must not include multiple variables.
- Properties must not include symbols related to synchronization (symbols removed in Section 5.3.4)

For example, both LTL formulae $a = 1 \rightarrow X(a) = 2$ and $F(a \wedge b)$, where $a, b \in I \cup V \cup O$, cannot be verified correctly by the proposed method since the first formula includes X operator which represents "one cycle later", and the latter formula includes two variables a and b. AG and AF in CTL can be verified since they do not have the notion of constant cycles. To guarantee the correctness of verifications with properties satisfying the above conditions, a proof of the following theorem is shown.

Theorem 1. *Sequential FSMDs after the sequentialization and state merging contain all possible value update sequences for all symbols which are not related to syn-*

chronizations in the original concurrent FSMDs.

A value update sequence represents an ordered set of value updates for a single symbol (variable).

Proof. Value update sequences of shared symbols are not changed nor removed in the sequentialization since it checks race conditions between each access to each symbol and generates all possible cases when race conditions exist. Value update sequences of local symbols are not changed nor removed in the sequentialization since the sequentialization does not touch the value update sequences of local symbols. Though their value depends on shared symbol values, all possible cases of shared symbol values should be included in sequential FSMDs generated by the sequentialization. State merging does not change nor remove value update sequences of symbols since it does not merge states which have data dependence to each other. Then, there cannot be multiple assignments for a same variable in a state, and data sequences are unchanged. Since neither the sequentialization nor the state merging changes value update sequences of symbols, and the sequentialization generates all possible value update sequences in the original design, sequential FSMDs after the sequentialization and the state merging contain all possible value update sequences for all symbols which are not related to synchronizations in the original concurrent FSMDs. □

In equivalence checking, since the execution timings are changed by the sequentialization and the state merging, the timings when output values should be equivalent must be specified by users after them.

5.4 Experimental Results

To show the effectiveness of the proposed method, experiments were performed with practical designs.

Examples Two hardware/software co-designs which implement Inverse Discrete Cosine Transformation (IDCT) and Discrete Cosine Transformation (DCT), respectively, were used in the experiments. In these designs, the hardware parts execute

Table 5.2: Examples

Design	#lines		#states in FSMD		
	SW	HW	SW	Int.	HW
DCT	364	166	353	3	11
IDCT	369	249	352	3	18

low and column transformations, and the software parts manage vectors and control the hardware parts. Memory mapped I/O, hand-shaking, and interruption are used in those designs. Table 5.2 shows the details of the designs. The second and third columns shows numbers of lines in software part (ANSI-C) and hardware part (Verilog-HDL), respectively. The third and later columns show the information of generated FSMDs by the method proposed in Section 5.3.2. The third, fourth, and fifth columns show numbers of states in software FSMDs, interruption sequence FSMDs, and hardware FSMDs, respectively.

Sequentialization Results First, the proposed sequentialization method was applied to those examples. Before applying sequentialization, those examples were unrolled as explained in Section 5.3.2. Both hardware portions and software portions of DCT and IDCT examples have large loops. The software and hardware portions of DCT are unrolled for 16 times. The software and hardware portions of IDCT are unrolled for 8 times since IDCT uses two different hardware modules for the column computation and the row computation, respectively.

The sequentialization was performed on a Linux workstation with Intel Core2Duo 3.16GHz processor and 4GB memory. An SMT solver Yices[179] was used as the engine.

Table 5.3 shows the result of the sequentialization. The second column shows the number of sequences of states. The third column shows the number of detected synchronization points. The fourth column shows the number of race condition candidates. The fifth column shows the number of generated sequential designs. The sixth column shows the maximum size of used memory. The seventh column shows the computation time. The computation time includes all executions of satisfiability checking until the result become unsatisfiable.

Table 5.3: Result of Sequentialization

Design	#seq. of states	#sync points	#race cands	#generated seq. designs	memory usage	comp. time
DCT	400	80	8192	1	1.3GB	691.08s
IDCT	400	80	6144	1	1.6GB	988.71s

The results show that even such the complicated examples having more than 6000 race condition candidates can be sequentialized within the realistic execution times.

Formal Verification Results Second, model checking was applied to the generated sequential designs by the sequentialization method. Before applying model checking, an abstraction was applied to the designs where 54 of 64 inputs in total were fixed to 0, and the other input took only 0 or 1 since the designs were too large.

Model checking was performed by a model checker Spin[73] which implements partial order reduction. The verified property was "Computation eventually terminates". This property should be satisfied even the designs are abstracted since the property does not depends on bit-widths of the inputs. Model checking is executed on the same Linux workstation having Intel Core2Duo 3.16GHz processor and 4GB memory.

Table 5.4 shows the experimental results with Spin model checker. Five types of verification methods were applied, such that standard model checking without partial order reduction, standard model checking with partial order reduction, standard model checking with synchronization points but without partial order reduction, standard model checking with synchronization points and partial order reduction, and the proposed method. Results of them are shown in the first, second, third, fourth, fifth lines of each example, respectively. The first column shows the names of verified designs. The second column shows the types of the applied methods, the third column shows the numbers of FSM states generated during the verification, the forth column shows the verification times, and the fifth column shows the used memory sizes. Verifications of DCT without synchronization points resulted in out of memories regardless of the use of partial order reduction.

Table 5.4: Results of Model Checking with SPIN

Design	Method	#states	time[s]	memory[MB]
DCT	Without POR	5746678	279.89	1205.701
	With POR	1534966	77.36	329.904
	With Sync. Points	3911672	240.69	910.233
	With Sync. Points & POR	1305592	58.74	308.787
	Proposed Method	383990	19.74	82.029
IDCT	Without POR	3227342	214.38	689.435
	With POR	473559	19.58	102.970
	With Sync. Points	2334360	118.79	587.722
	With Sync. Points & POR	396165	15.09	101.254
	Proposed Method	92724	3.00	18.747

The results in Table 5.4 show that both the synchronization point detection and the sequentialization improved the performances in both the examples. With the combination of synchronization point detection and sequentialization, verification can become more than 3 times faster. Therefore, the proposed method can be much more efficient than existing methods, such as partial order reduction.

5.5 Conclusion and Future Work

In this chapter, a framework to verify hardware/software co-designs written in program code and RTL was proposed. The proposed method can handle general types of interactions between hardware and software, such as memory mapped I/O and interruption with priority. Compared to the method proposed in [176], the strong state reduction techniques which extends the existing sequentialization techniques[145, 114] and state merging can reduce the number of states dramatically. Then, more practical and larger designs can be handled.

Chapter 6

ExSDG: Design Representation for Efficient High-Level Design Verification

6.1 Introduction

As mentioned in Section 1.1, C-based designs have become important in the system LSI (or SoC) design flow. C-based design languages are used in system level, behavioral level, and program code to describe behaviors of the designs. In such abstracted design levels, designers can write designs without considering hardware/software partitioning. Although most designs are still started in program code and RTL, it is expected that the initial design stage will shift to more abstracted levels in the future. To support the design stage shift, efficient verification methods and tools for C-based designs are strongly required.

One problem in C-based design verification is the high complexity of some languages. For example, SystemC and SpecC have channels and interfaces to describe communications separately from behaviors. However, these new notions require modifications or complete re-development of existing sophisticated hardware and software verification methods and tools. Pointer is also a difficult syntax element to handle in verification or analysis. Typically, verifications and analyses mainly target on design behaviors, and communication portions tend to be verified separately. Then such difficult syntax elements should be removed at a preprocess step before applying verification or analysis methods.

Another problem in C-based design verification is the existence of a number of C-based design languages, such as SystemC[160], SpecC[54], SystemVerilog[82], BachC[177], BDL[172], and CataplutC[28], and for that reason there are no common intermediate design representations for verification. Then verification tools must be developed for each language, and this situation is very inefficient and time consuming for verification software development. For RTL and gate-level designs, net-list based representations, such as Verific HDL Component Software[169] and And-Inverter Graph (AIG) based internal representations[120, 126], are widely used as standard intermediate representations for verification and design analysis in both industrial and academic fields. These net-list based representations can theoretically handle both Verilog-HDL and VHDL by translating them with tentative syntheses and elaborations. In the software field, intermediate representations in compiler frameworks, such as Low Level Virtual Machine (LLVM)[164], SUIF[165], and COINS[38], are also commonly used for verification or software analysis. Intermediate representations in these compiler frameworks can be generated from arbitrary software codes if front-ends for them have been developed. Such intermediate representations for RTL and software are sufficiently simple and easy to handle in verification or analysis tools. Although there are some existing frameworks providing intermediate representations for C-based designs, such as Pinapa[132] and SpecC Intermediate Representation/Syntax Independent Representation (SIR)[54, 170], they were not originally developed for multiple languages and includes many complex elements which do not suit to verification nor analysis.

Based on the above discussion, a new intermediate representation for high-level design verification and analysis is proposed in this chapter. Although there are many different C-based design languages, they are quite similar in the sense of language design perspectives. SystemC and SpecC are two major C-based design languages, and they are based on C++ and C, respectively. Although they can be syntactically very different, the ways to describe system LSI design in higher abstracted levels are quite similar, or it can be simply said that they are the same. The notions of modules and structural hierarchy, introduction of concurrent executions and their synchronization mechanisms, and various ways to control software and hardware interactions are the same in the two languages. Moreover, other C-based languages are sharing the same concepts as these with their own syntax. Therefore, it is very natural to believe that a common design representation can be defined to which various C-based design descriptions can be converted. In this chapter, a representation for SystemC, SpecC, and other various C-based design languages as well as RTL design descriptions in Verilog and VHDL is defined, and how widely used C-based design descriptions can be converted into them are presented.

The proposed representation is based on System Dependence Graph (SDG)[75] introduced in Section 2.3.3 which has been used in program slicing tools in software fields. SDG can be generated from program descriptions where each sentence in the program corresponds to one node in SDG, and edges in SDG show various kinds of dependency, such as data, control, declaration, and others. Those edges in SDG give the similar functions to net-lists of gate-level designs. For example, by analyzing net-lists, logic cones of influence can be identified. It enables to reduce the number of gates to be analyzed. With SDGs, the similar reduction can be realized by program slicing which is an operation to extract all relevant nodes to a specified node. Debugging using this technique is called algorithm debugging. By Data Flow Graphs (DFGs), data dependencies can also be traced. However, it can be said that analysis with SDG is more efficient than those with DFG since the nodes in SDGs represent statements and expressions while the nodes in DFGs are usually variables and operators. Therefore, SDG can be considered one of the most effective and efficient representations of word-level designs for CAD tools. In addition, the proposed representation integrates Abstract Syntax Tree (AST) and Control Flow Graph (CFG) as well as SDG since AST is also necessary to analyze designs in syntax level, and control flow information is useful to analyze exact design behaviors. The representation is named "Extended System Dependence Graph (ExSDG)", and it has the following advantages.

- The syntax of the AST in ExSDG is simple enough to verify, but also extended ANSI-C enough to describe hardware designs in both behavioral level and RTL. The syntax is free from language particular problems.
- AST, CFG, and SDG are tightly integrated in ExSDG. Actually, edges of SDG and CFG are drawn directly to some AST nodes. It enables verification tool developers to analyze designs easily.
- The numbers of nodes and edges are less than that of existing SDGs since the SDG in ExSDG is highly optimized for the limited syntax. Then, the verification time of methods using SDG is expected to be shorter.

A verification framework with ExSDG is also proposed in this chapter. In the proposed framework, design descriptions in C-based languages or HDL in RTL are automatically converted into ExSDG, and various verification methods using SDG can be applied. This framework enables users to verify various designs easily and efficiently.

The remainder of this chapter is organized as follows: Section 6.2 explains the proposed verification framework with ExSDG. Section 6.3 describes the overview of the AST in ExSDG and shows how to translate from designs written in existing languages. In Section 6.4, ExSDG and its structure are explained. Preliminary experimental results of the comparison between ExSDG and an existing SDG are also given in this section. In Section 6.5, a survey of existing verification methods using SDG is given. Some of them have already been implemented using ExSDG. Finally, Section 6.6 gives a conclusion of this work.

6.2 Verification Framework

Figure 6.1 shows the verification flow of the proposed framework.

At the first step, all designs are translated into the ASTs. In this work, the AST is not defined as a concrete language in text but as a data structure of nodes and edges. Developers do not have to look into the locations of portions of codes after the translation since each node in the AST has an unique ID number. The proposed AST has three different styles to represent different abstraction levels, such as untimed level, timed level and register transfer level. The details of the AST syntax is shown in Section 6.3, and how to translate existing languages into AST is described in Section 6.3.3.

At the next step, a simplification is applied to the generated AST as a preprocess step to remove some syntax elements which make verification and analysis more difficult. That simplification includes channel/interface elimination, pointer analysis, transformation to single value update model, and for/switch elimination. The ASTs after the simplification are simple enough to easily handle in the later verification and analysis steps.

Then, those simplified ASTs are analyzed and converted into ExSDGs by the dependency and control flow analysis. In the ExSDGs, some nodes, various dependence edges and control flow edges are added to the ASTs. The dependence edges and control flow edges are drawn directly between nodes in the ASTs. Then, developers can get dependency information from the nodes of the ASTs directly. The details of ExSDG are described in Section 6.4.

At the last step, various verification and analysis methods are applied to the ExSDGs. Some methods using other existing SDGs have already been proposed [4, 146, 144, 113, 52], and some of them have been re-implemented with ExSDG.

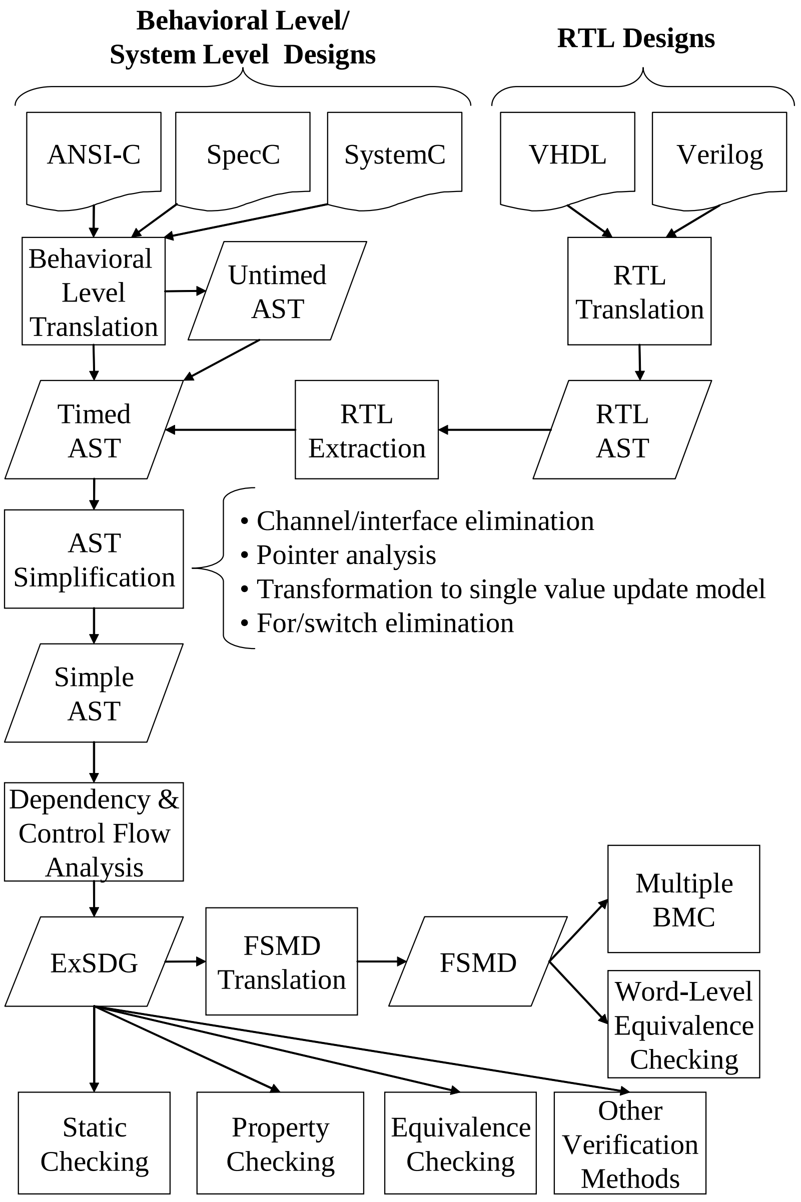


Figure 6.1: Proposed Verification flow

They are briefly introduced in Section 6.5. FSMDs can be directly generated from ExSDG by the method explained in Section 5.3.2, and the multi-level bounded model checking proposed in Chapter 3 and the word-level equivalence checking method explained in Chapter 4 can be applied to the generated FSMDs.

6.3 Language of the AST in ExSDG

6.3.1 Basic Syntax

The syntax of ExSDG AST extends that of ANSI-C to describe hardware and RTL designs. The extension list is as follows.

- Concurrency : A **par** statement represents concurrent execution of statements under it. Those statements start simultaneously as concurrent threads, and end in the same timing. In other words, starts and ends of those threads are synchronized.

- Synchronization : Synchronizations are represented by **event** variables, **wait** statements, and **notify** statements. An execution of a **wait** statement stops the thread until the execution of a **notify** statement whose parameter is the same **event** variable.

- Bit-vector : **bit** type is introduced to represent bit-vectors. Widths can be specified on **bit** variables. With **bit** type, bit-level operators, such as bit-slice [**left-bit:right-bit**] and concatenation **@**, are introduced.

- Timed behavior : To represent timed behavior, **waitfor** statement which pushes one time unit forward is introduced. A **buffered** declarator specifies that the variable is a register, and its value is not updated until the time is forwarded for one time unit. A **wire** declarator specifies that the variable is a wire, and its value is update immediately when the value of the assigned expression is updated. Therefore, only one assignment is allowed for each **wire** variable.

- Hierarchical structure : A **module** statement represents a structural unit, and a **port** statement represents a port of a module for communications with the other modules. Sub-modules can be represented by creating their instances inside a module. A **channel** is an unit where communication related portions are encapsulated, and an **interface** is a pure abstract class which is an interface of channels.

6.3.2 Three Representations in Different Levels

The AST of ExSDG has three different levels to translate directly from other representations in different abstraction levels, such as system-level, behavior level, and register transfer level.

Meanwhile, the language of the AST do not have notations in text since it is only defined as an AST. Therefore, the descriptions in this section which is written in the language of the AST is just an example with a compatible notation for helping readers' understandings.

Untimed Behavior Level (UBL) Untimed behavior level (UBL) does not have the notion of time, and just the execution order of operations is fixed. In UBL, use of timed behavior syntax, such as **waitfor** statement, and **buffered** and **wire** declarators, is not allowed. This level corresponds to program code, such as ANSI-C and Java, behavioral level language, such as BachC, BDL, and CataplutC, and system level languages, such as SystemC and SpecC where timing statements are not used. Since UBL does not have the notion of time, it is assumed that the main function of a top level module specified by users or global main function is executed once for each time unit.

Figure 6.2 shows an example UBL code. This example receives three input values for each of five time units (15 values in total), and returns the maximum value among them. There are two modules, and the top level module is **Max35**. Then the **main** function of **Max35** is executed for each time unit.

Timed Behavior Level (TBL) In Timed Behavior Level (TBL), the notion of time is added to UBL. In particular, **waitfor** statement can be used in TBL. **buffered** and **wire** declarators are still prohibited. Clearly, UBL is a subset of TBL. TBL corresponds to System Level language, such as SystemC and SpecC. In TBL, the main function of a top level module specified by users is just executed once at the beginning.

Figure 6.3 shows an example code of TBL. This code is equivalent with Figure 6.2, but written in TBL. In this example, an infinite loop is used in the main function since the main function is executed only once. A **waitfor** statement is also inserted

```
module Max(in int in1, in int in2, out int out1){
  void main(){
    out1 = (in1 > in2) ? in1 : in2;
  }
};
module Max35(in int in1, in int in2, in int in3, out int out1, in bool rst){
  event e1;
  int current, i, tmp1, tmp2;
  Max m1(in1, in2, tmp1), m2(in3, current, tmp2), m3(tmp1, tmp2, current);
  void main(){
    if(rst){
      i = 0;
      current = 0;
      out1 = 0;
    }
    else{
      par{
        { m1.main();
          wait(e1);
          m3.main(); }
        { m2.main();
          notify(e1); }
      }
      if(i == 4){
        out1 = current;
        current = 0;
        i = 0;
      }
      else{
        out1 = 0;
        i++;
      }
    }
  }
};
```

Figure 6.2: Example code in untimed behavior level

to proceed the time for one time unit.

Register Transfer Level (RTL) In register transfer level (RTL), the notions of clock cycle, register, and wire are used. In addition, **waitfor** statement whose argument is 1 (**waitfor(1)**) can be used. **buffered** and **wire** declarators can also

```
module Max(in int in1, in int in2, out int out1){
  void main(){
    out1 = (in1 > in2) ? in1 : in2;
  }
};
module Max35(in int in1, in int in2, in int in3, out int out1, in bool rst){
  event e1;
  int current, i, tmp1, tmp2;
  Max m1(in1, in2, tmp1), m2(in3, current, tmp2), m3(tmp1, tmp2, current);
  void main(){
    while(1){
      if(rst){
        i = 0;
        current = 0;
        out1 = 0;
      }else{
        par{
          { m1.main();
            wait(e1);
            m3.main(); }
          { m2.main();
            notify(e1); }
        }
        if(i == 4){
          out1 = current;
          current = 0;
          i = 0;
        }else{
          out1 = 0;
          i++;
        }
      }
      waitfor(1);
    }
  }
};
```

Figure 6.3: Example code in timed behavior level

be used to represent registers and wires respectively. On the other hand, there are the following restrictions.

- Synchronization related syntaxes, such as **wait** and **notify** statements and **event** variable are prohibited.

- Normal variables without **buffered** nor **wire** declarator cannot be used.
- Use of **channel** and **interface** is prohibited.
- Ports must be **wire** variables.

In RTL, two special member functions such that **init** and **run_one_cycle** functions are included in each module.

- **init** function: It describes wire connections in a module. Then all included assignments must be to **wire** variables, and an assignment to each **wire** variable can only be written once. The value of a **wire** variable is immediately updated when any variable in the right hand side of the assignment to the **wire** variable is updated. This corresponds to **assign** statement in Verilog-HDL.
- **run_one_cycle** function: It describes the behavior of each clock cycle. Then this function is executed for each clock cycle. Only assignments to **buffered** variables are allowed in this function. The values of those buffered variables are updated at the beginning of the next clock cycle. This function corresponds to **always** statement describes sequential behavior in Verilog-HDL.

Each of these two functions can be omitted if designers like. The above restrictions and these two special functions are for the compatibility with RTL designs in HDLs. Then **run_one_cycle** functions in all module instances are parallely executed for each clock cycle.

Figure 6.4 shows an RTL design equivalent to Figure 6.2 and Figure 6.3 but in RTL.

6.3.3 Translation from Other Languages

In this section, the way of translations from four languages, ANSI-C, SpecC, SystemC, and Verilog-HDL is discussed.

Translation from ANSI-C Since ANSI-C is a subset of the ExSDG syntax, ANSI-C codes are untimed behavioral level descriptions without any modifications.

```
module Max(in wire int in1, in wire int in2, out wire int out1){
  void init(){
    out1 = (in1 > in2) ? in1 : in2;
  }
};
module Max35(in wire int in1, in wire int in2, in wire int in3,
             out wire int out1, in wire bool rst){
  wire int tmp1, tmp2, current_i, current_o;
  buffered int i, current;
  Max m1(in1, in2, tmp1), m2(in3, current_i, tmp2), m3(tmp1, tmp2, current_o);
  void init(){
    out1 = (i == 4) ? current : 0;
    current_i = current;
  }
  void run_one_cycle(){
    if(rst){
      i = 0;
      current = 0;
    }
    else{
      current = current_o;
      if(i == 4){
        i = 0;
      }
      else{
        i++;
      }
    }
  }
};
```

Figure 6.4: Example code in register transfer level

Translation from SpecC Most of the ExSDG syntax elements come from SpecC. Then, only small modifications are required. What only have to be done is to interprete **behavior** as **module**. A SpecC code is translated either into an UBL or TBL description, and it depends on the use of **waitfor** statement. If there are no **waitfor** statements, then the translated description is in UBL, otherwise it is in TBL.

However, the following syntaxes cannot be handled since those are not included in the syntax of ExSDG.

- pipe, piped, fsm, fsmd
- notifyone
- try, trap, interrupt
- do-timing
- buffered, signal

Translation from SystemC Since SystemC is based on C++, most operators and data types are same. Table 6.1 shows the main additional syntax elements in SystemC and their corresponding syntax elements in ExSDG. Most part of the translation can be performed by just following the table. However, the ways to represent concurrency are different in SystemC and ExSDG. Figure 6.5 shows the difference. In ExSDG, statements just under a **par** statement are executed concurrently. On the other hand in SystemC, **SC_CTOR**s in all **SC_MODULE**s are executed concurrently at the beginning of a simulation, and those **SC_CTOR**s execute processes with **SC_MODULE**, **SC_THREAD**, and **SC_CTHREAD**. Therefore, in the translation, a global main function is added, and a **par** statement which executes all top-level instances is inserted to the main function. A SystemC code is translated into TBL if **wait(sc_time)** or **wait()** in **SC_CTHREAD** is used, otherwise it is translated into UBL.

Figure 6.6 shows an example SystemC code and Figure 6.7 shows its translation to ExSDG. Two modules **Sender** and **Receiver** are executed concurrently. Those modules in SystemC are implicitly executed concurrently. On the other hand, those modules are explicitly executed concurrently by a **par** statement in SpecC. Since **wait(sc_time)** is not used in the design, the generated ExSDG description is in UBL.

Translation from Verilog-HDL A Verilog-HDL description can be translated into an RTL ExSDG code. Their syntax correspondence is shown in Table 6.3.3. The syntax elements listed in the table can be translated into an RTL ExSDG code by replacing those syntax elements with the corresponding ones in ExSDG. For

Table 6.1: Syntax correspondence between SystemC and ExSDG

SystemC	ExSDG
SC_MODULE	**module**
sc_in	input port (**in** declarator)
sc_out	output port (**out** declarator)
initialization in **SC_CTOR**	top of a main function
sc_int<int>	**bit[int]**
sc_time	**int**
sc_event	**event**
wait(sc_time)	**waitfor(int)**
wait(sc_event)	**wait(event)**
notify(sc_event)	**notify(event)**
notify(sc_event, sc_time)	**waitfor(int); notify(event);**
wait() in **SC_CTHREAD**	**waitfor(1)**

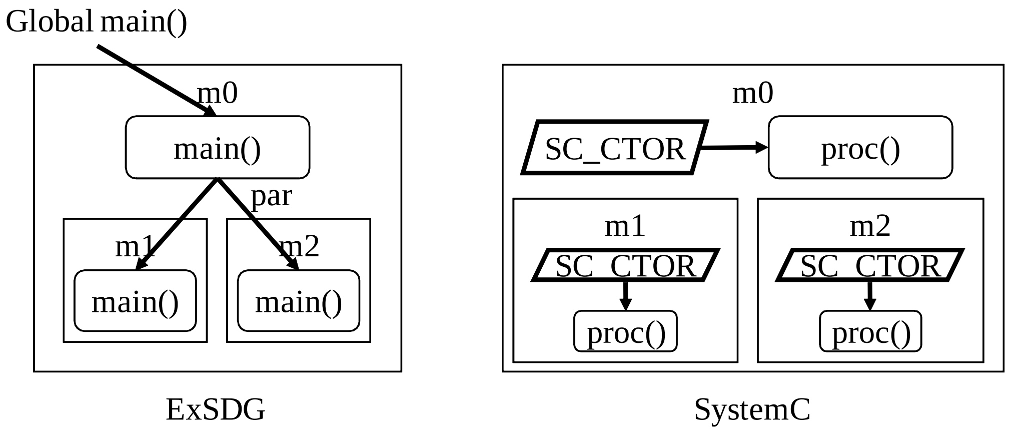


Figure 6.5: Difference of concurrent process execution in ExSDG and SystemC

example, **assign** statements are translated into assignments in an **init** function. In this translation, only single clock designs can be handled.

Figure 6.8 shows an example Verilog-HDL design equivalent to the design in Figure 6.4. Therefore, this design is translated to the design in Figure 6.4. As you can see from those designs, the translation is almost one by one replacement since the syntax of ExSDG RTL is close to that of Verilog-HDL.

```
class ReadInterface : public sc_interface{
public:
  virtual int read() = 0; };
class WriteInterface : public sc_interface{
public:
  virtual void write(const int) = 0; };
class MyChannel : public sc_channel, public ReadInterface, public WriteInterface{
  int data; sc_event e;
public:
  MyChannel(sc_module_name name) : sc_channel(name){}
  int read(){ wait(e); return data; }
  void write(const int x){ data = x; e.notify(SC_ZERO_TIME); }
};
SC_MODULE(Sender){
public:
  sc_port<WriteInterface> wi;
  void main(void){ wi->write(5); }
  SC_CTOR(Sender){ SC_THREAD(main); }
};
SC_MODULE(Receiver){
public:
  sc_port<ReadInterface> ri;
  void main(void){ int tmp = ri->read(); }
  SC_CTOR(Receiver){ SC_THREAD(main); }
};
int sc_main(int argc, char** argv){
  MyChannel mc("my_channel");
  Sender s("sender"); s.wi(mc);
  Receiver r("receiver"); r.ri(mc);
  sc_start();
}
```

Figure 6.6: Example code in SystemC

6.3.4 AST Simplification

In the framework shown in Figure 6.1, translated ExSDG description is simplified by removing some syntax elements which are difficult to handle in verification or dependency analysis. The removed syntax elements are as follows.

- channel/interface
- pointer

```
interface ReadInterface{ int read() = 0; };
interface WriteInterface : public sc_interface{ void write(const int) = 0; };
channel MyChannel : ReadInterface, WriteInterface{
  int data; event e;
  int read(){ wait(e); return data; }
  void write(const int x){ data = x; notify(e); }
};
module Sender(WriteInterface wi){
  void main(void){ wi.write(5); }
};
module Receiver(ReadInterface ri){
  void main(void){ int tmp = ri.read(); }
};
int Main(int argc, char** argv){
  MyChannel mc;
  Sender s(mc); Receiver r(mc);
  par{s.main(); r.main()}
}
```

Figure 6.7: ExSDG code translated from Figure 6.6

Table 6.2: Syntax correspondence between Verilog-HDL and ExSDG

Verilog-HDL	ExSDG
module	module
port(input/output/inout)	port(in/out/inout)
wire	wire variable
reg	buffered variable
sub modules	sub modules
assign statement	assignment in init function
always statement	statement in run_one_cycle function
blocking assignment	assignment
if-else	if-else
case	switch-case

- multiple value updates (assignments) in a single statement
- for/switch

```
module Max(in1, in2, out1);
  input [31:0] in1, in2;
  output [31:0] out1;
  assign out1 = (in1 > in2) ? in1 : in2;
endmodule

module Max35(in1, in2, in3, out1, rst, clk);
  input [31:0] in1, in2, in3;
  output [31:0] out1;
  input rst, clk;
  wire [31:0] tmp1, tmp2, current_i, current_o;
  reg [31:0] i, current;
  Max m1(in1, in2, tmp1); m2(in3, current_i, tmp2); m3(tmp1, tmp2, current_o);
  assign out1 = (i == 4) ? current : 0;
  assign current_i = current;
  always @(posedge clk) begin
    if(rst) begin
      i <= 0;
      current <= 0;
    end
    else begin
      current <= current_o;
      if(i == 4)
        i = 0;
      else
        i++;
    end
  end
endmodule
```

Figure 6.8: Example code in Verilog-HDL equivalent to Figure 6.4

Removing Channels and Interfaces Channels and interfaces represent communications between modules executed concurrently. However, this structure is too complex to verify the behaviors. Actually, a more simple structure of communication is composed of shared variables and wait/notify statements, and many verification methods including ones proposed in this thesis target on that structure. Therefore, channels and interfaces are removed from designs by translating them into accesses to global variables with synchronizations by wait/notify statements.

This translation is performed as shown in Figure 6.9. Since a channel is just a capsule of shared variables and methods accessing to those variables, they can

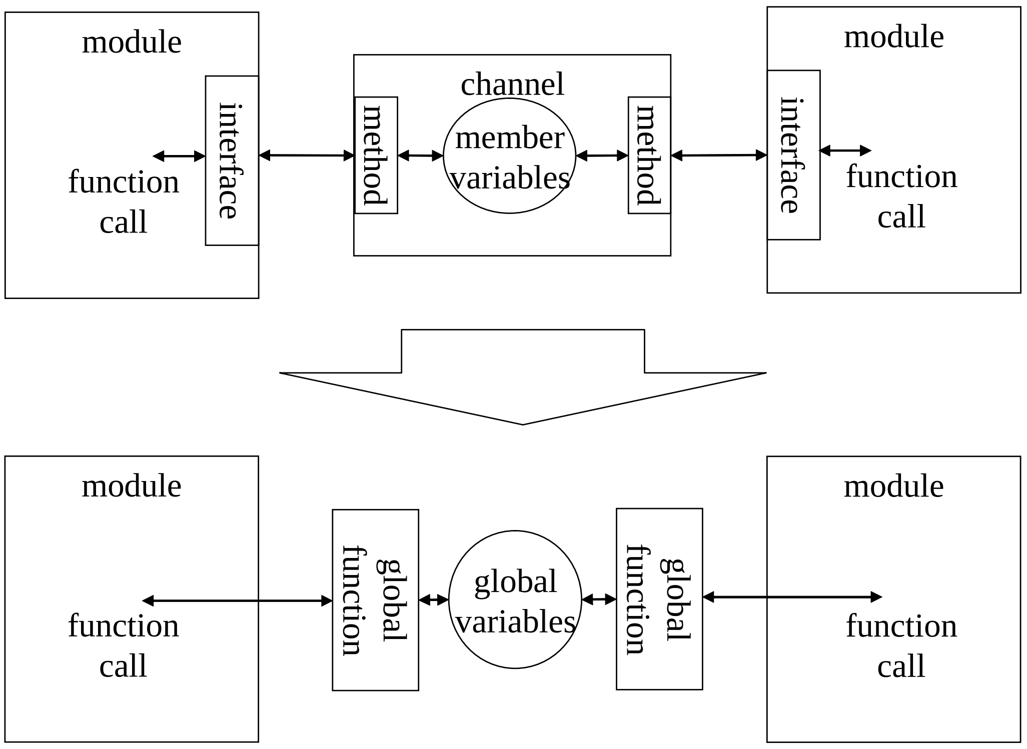


Figure 6.9: Conversion from channels and interfaces

be translated into global variables and functions, respectively. Function calls of interface methods are also replaced with function calls of those global functions. Note that this extraction must be performed for each channel instance. Therefore, when there are multiple instances for a single channel, one set of global variables and functions must be added for each instance.

Figure 6.10 shows an example description after this translation applied to the design in Figure 6.7.

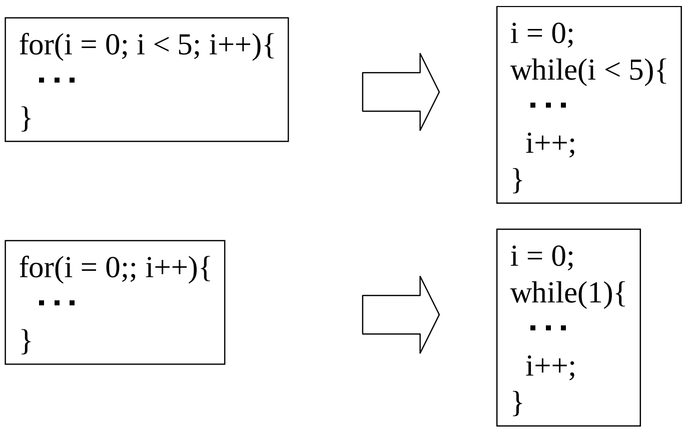
Removing Pointers Since resources pointed by pointers change dynamically, it is very difficult to statically verify designs including pointers. Therefore, in most verification methods, pointers are removed from designs in the preprocess stage by applying point-to analysis[70] as introduced in Section 5.3.2. Pointers are also removed in the same way in ExSDG.

```
int MyChannel_mc_data;
event MyChannel_mc_e;
MyChannel_mc_send(int x) { MyChannel_mc_data = x; notify(MyChannel_mc_e); }
int MyChannel_mc_receive() { wait(MyChannel_mc_e); return MyChannel_mc_data; }
module Sender() {
  void main() { MyChannel_mc_send(5); }
};
module Receiver() {
  void main() { int tmp = MyChannel_mc_receive(); }
};
module Main {
  Sender s; Receiver r;
  int main(int argc, char** argv) {
    par { s.main(); r.main(); }
  }
};
```

Figure 6.10: ExSDG code translated from Figure 6.6

Removing multiple value updates in a single statement In most dependency analysis techniques, dependence among expressions is analyzed. Here, an expression is a smallest element having values in statements. For example, in a statement **a = b + c;**, there are five expressions, such as **a**, **b**, **c**, **b + c**, **a = b + c**. Analyses must be performed on this granularity to handle multiple value updates in a single statement, such as **a = b++;**. In this statement, the values of two variables, **a** and **b** are updated. Because of the granularity, the number of nodes in dependence graphs tends to be large. Since the number of nodes can be considered as the design size in verification, it is better if number of nodes can be reduced. Therefore, multiple value updates are removed from ExSDG by splitting such statements, and dependency analyses are performed not for expressions but for statements in ExSDG. For example, a statement **a = b++;** is split into two statements, **b++;** and **a = b**; This technique can greatly reduce the number of nodes in dependence graphs.

Removing for/switch Statements **for** and **switch** statements represent iterative execution and conditional branch, respectively. However, those can also be



```
for(i = 0; i < 5; i++){
  . . .
}
```

```
i = 0;
while(i < 5){
  . . .
  i++;
}
```

```
for(i = 0;; i++){
  . . .
}
```

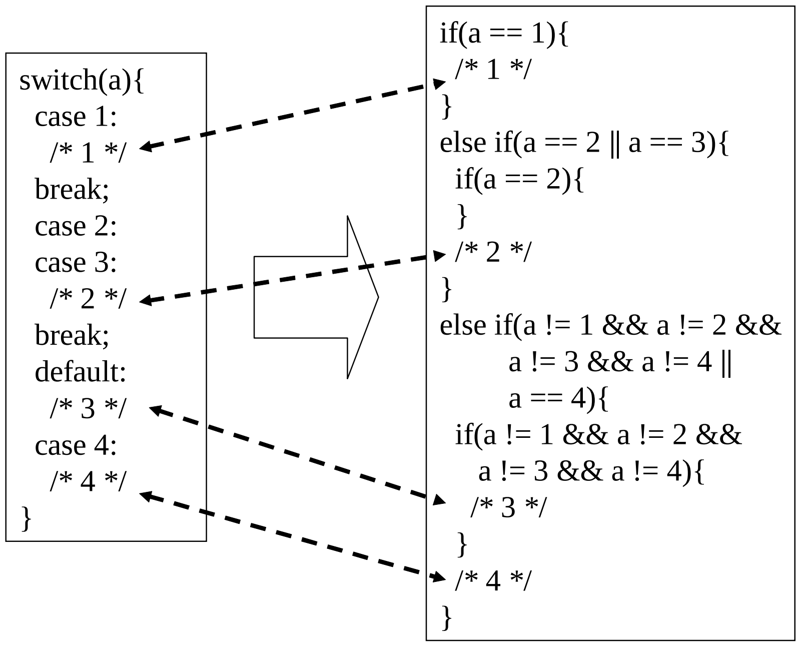
```
i = 0;
while(1){
  . . .
  i++;
}
```

Figure 6.11: Replacement of **for** statement with **while** statement

written with **while** and **if-else** statements. Therefore, tool implementations which handle both of those pairs have redundancy. To avoid such implementation redundancy, **for** and **switch** statements are replaced with **while** and **if-else** statements in ExSDG.

The replacement of a **for** statement with a **while** statement is simply performed by decomposing three expressions in the **for** statement as shown in Figure 6.11. The first expression is moved to the position before the **while** statement. The second expression is used as an expression in **while** statement. When the second expression is empty, the **while** statement is an infinite loop. The third expression is placed at the end of the iteration.

A **switch-case** statement is replaced with multiple nested **if-else** statements as shown in Figure 6.12. The top-level **if-else** statements in the design after the translation corresponds to the blocks separated by **break** statements in the original description. If there are multiple **case** or **default** labels in a single block, additional **if-else** statements are inserted to distinguish statements in different positions with those labels.



```
switch(a){
  case 1:
    /* 1 */
  break;
  case 2:
  case 3:
    /* 2 */
  break;
  default:
    /* 3 */
  case 4:
    /* 4 */
}
```

```
if(a == 1){
  /* 1 */
}
else if(a == 2 || a == 3){
  if(a == 2){
  }
  /* 2 */
}
else if(a != 1 && a != 2 &&
         a != 3 && a != 4 ||
         a == 4){
  if(a != 1 && a != 2 &&
     a != 3 && a != 4){
    /* 3 */
  }
  /* 4 */
}
```

Figure 6.12: Replacement of **switch** statement with **if-else** statement

6.4 Extended System Dependence Graph

6.4.1 Nodes in ExSDG

As mentioned in Section 6.1, ExSDG is a hybrid graph of AST and SDG, and some nodes are shared in the both. In this section, types of nodes in ExSDG are introduced.

The types of AST nodes are shown in Table 6.3. Nodes whose types are **Function**, **Variable**, and **Statement** are shared with SDG. The structure of an AST with those nodes are also shown in Figure 6.13. Edges in this tree show a relation between super-nodes and sub-nodes. For example, a **Design** node has **Module** nodes as its sub-nodes.

The types of SDG nodes are shown in Table 6.4. The first three types are shared with AST. The other five types are originally proposed in [75] to describe inter-procedural dependence. Nodes in those five types are added to an AST before applying dependency analysis.

Table 6.3: Node types in the AST

Name	Explanation
Design	Top node of a design
Module	Declaration of a module
Struct	Declaration of a structure
Scope	Scope of declarations
Label	Label used in a design
Port	Declaration of a port of a module
Function	Declaration of a function
Variable	Declaration of a variable
Argument	An argument of a function
Type	A type of variables or expressions
Statement	A statement in a function
Expression	An expression in a statement
Constant	A constant value in an expression

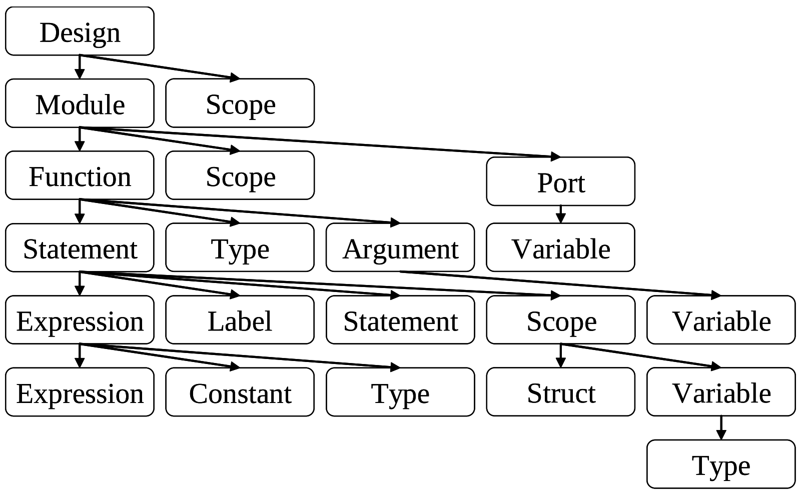


Figure 6.13: AST structure of ExSDG

6.4.2 Edges in ExSDG

Table 6.5 shows the types of SDG edges in ExSDG. Call edge, parameter-in edge, parameter-out edge, and inter-procedural dependence edge are the edges to describe the dependence related to function calls proposed in [75]. Communication depen-

Table 6.4: Node types in the SDG

Name	Explanation
Function	Entry point of a function (procedure)
Variable	Declaration of a variable
Statement	A statement in a procedure
End	Ending point of a function or a control statement
Call	A function call in a statement
Formal In	An argument in a function declaration
Formal Out	Return value in a function declaration
Actual In	An argument in a function calls
Actual Out	Return value in a function call

Table 6.5: Types of edges in SDG

Name	Meaning
Control dependence edge	A control dependence in a function
Data dependence edge	A data dependence in a function
Declaration dependence edge	Dependence between a variable declaration and a state
Call edge	Represent a function call
Parameter-in edge	An inter-procedural data dependence through an argument
Parameter-out edge	An inter-procedural control dependence thorough a return value
Interprocedural dependence edge	A data dependence between an argument and a return value
Interference dependence edge	A data dependence between concurrent threads
Communication dependence edge	A control dependence by synchronization
Control flow edge	Represent an execution order of statements

dence edge and interference edge are the edges to describe the dependence between concurrent threads proposed in [101]. Those edges are required since ExSDG can describe both function calls and concurrent threads. Control flow edges are also drawn to help control flow analysis.

In ExSDG, SDG edges are drawn only between AST nodes in selected types, such that **Function**, **Statement**, and **Variable**. Since the granularity is coarser than existing dependence graphs, numbers of edges and nodes in ExSDG are small. This makes verifications and analyses in the later step easier.

```
module Ex1(in bit [7:0] in1){
  bit [7:0] a;
  void main(){
    a = in1;
    a = a + 1;
  }
};
```

Figure 6.14: Example design

Table 6.6: Experimental results in SDG generation of IDCT

	# of nodes	# of edges	generation time [sec]
ExSDG	453	2061	8.5
SDG in [4]	6380	48073	19.3

6.4.3 Example of ExSDG

Figure 6.15 shows an ExSDG of the description shown in Figure 6.14. The shadow nodes are shared nodes between the AST and the SDG, and the other nodes are AST nodes. As shown in the figure, AST and SDG are tightly integrated in ExSDG. Then, users can easily access dependence edges from AST nodes, and also get the syntactical information of the nodes during dependency analysis.

6.4.4 Experimental Result

A prototype of dependence analysis tool which generates ExSDGs has been implemented. With that tool, an experiment was performed on the IDCT example in [4]. This experiment was carried out on a PC with Xeon 3.2 GHz CPU and 2GB memory, which is the same as that in [4].

Table 6.6 shows the result. The second line is the result from [4]. Since the SDG in [4] was generated by a commercial program slicer, CodeSurfer[37], it is expected to be similar to that in [162].

This experimental result shows that the numbers of nodes and edges are reduced

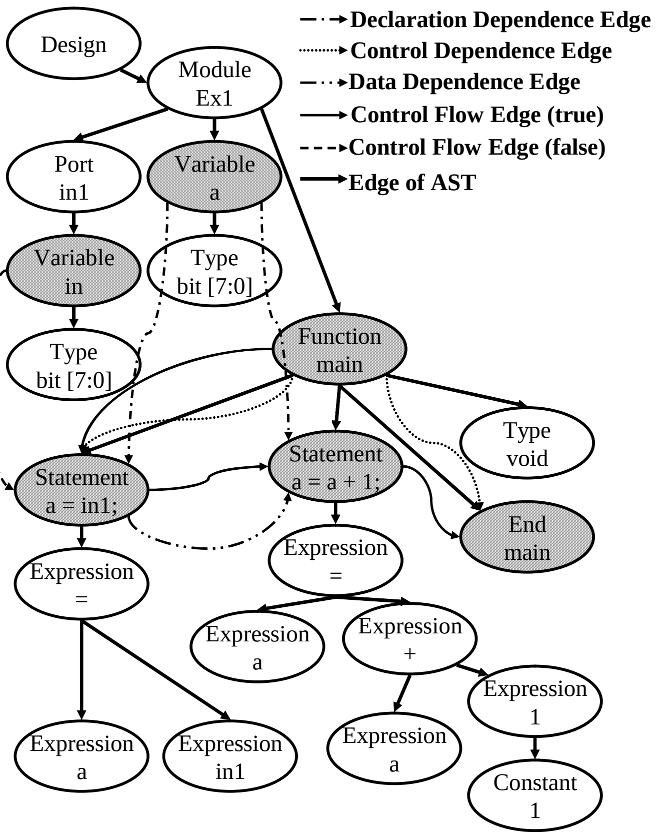


Figure 6.15: ExSDG of the design in Figure 6.14

dramatically in ExSDG so that the generation time also became shorter. This is because the granularity of ExSDG is coarser than that of the SDG of CodeSurfer.

6.5 Verification methods using SDG

In this section, several types of verification methods using SDG are introduced. Some of those methods have been implemented with ExSDG. Those methods utilize dependency information effectively to verify designs.

6.5.1 Design Error Detection

In [146, 4], static program checkers using SDG are proposed. Those program checkers can detect typical design errors, such as uninitialized variables, unused portions, nilpointer dereferences, out-of-bound array indexes, race conditions, and deadlocks. Since those program checkers just traverse SDG to detect those design errors, they can detect those errors quickly. For example, the detection of uninitialized variables took only 0.5 seconds for the libj2k example which has 2500 lines in [146]. However, since the detection is not complete, those program checkers should be used before applying precise checking.

6.5.2 Synchronization Verification

In [144], a property checking method for synchronization properties is proposed. Counterexample-Guided Abstraction Refinement (CEGAR) is used in the method, and it can be applied to large designs, such as an MPEG4 decoder. The existence of deadlocks and race conditions can be checked by the method. Different from the methods in [4], this method can check those properties completely. However the verification time becomes longer. It took 9.7 seconds to verify the mpeg4 example which has 48000 lines. SDG is used at the refinement process in the CEGAR to discover new predicates.

6.5.3 Equivalence Checking

In [113, 112, 115], an symbolic equivalence checking method between two designs written in ANSI-C is proposed. In the method, only different portions of two designs are compared and verification speed improves dramatically from the full comparison. This method can check the equivalence of two designs exhaustively since symbolic simulation is used as the verification engine. SDG is used in the extensions of verification areas to get related portions to different portions. By this method, the equivalence of the two portions (1160 lines) of an mpeg2 design was proved within 1.8 seconds.

In [114, 145], combination between the synchronization verification method in [144] and the equivalence checking method in [113] are proposed to check the equivalence between concurrent designs. In their experiments, IDCT and Vocoder examples were verified within 5 hours and 1 second respectively.

In [52, 151, 180, 181], rule-based formal equivalence checking methods are also proposed. In the methods, equivalences are established in a bottom-up fashion from textual equivalences by applying pre-defined rules based on the structure of the program. SDG with Control Flow Graph is the intermediate representation of the method used to analyze the structure. Since the verification is executed only by traversing the edges of the intermediate representation, it can check the equivalence of large designs quickly. For example, the equivalence of IDCT descriptions written in SpecC was proved within 3 seconds.

6.6 Conclusion

In this chapter, an SDG based intermediate representation ExSDG was proposed. A verification framework using ExSDG was also proposed. In the verification framework, designs written in various design language in system level, behavioral level, and RTL are translated into ExSDG. Then, dependency analyses are applied, and various verification methods using SDG can be applied. In addition, since ExSDG is optimized for verification, those verification methods can be easily and efficiently applied and implemented. Several types of verification methods using SDG were also introduced.

ExSDG is used as the intermediate representation of a verification framework FLEC[100], and many verification and analysis methods[51, 2, 99, 3, 98, 180, 181, 66] have been implemented on it.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, the following four methods with focusing on the separation of control and data portions in high level design were proposed. Those methods use Finite State Machine with Datapath (FSMD) which can describe control and data portions of a design separately as an intermediate representation.

In Chapter 3, a model checking heuristic which concatenates multiple bounded model checking results was proposed. The first bounded model checking is performed without initial state condition, and the second bounded model checking checks the reachability from the initial state to the generated counter example by the first bounded model checking. When the second bounded model checking failed, the initial state condition is refined not to generate the same counter example as generated by the first. The second bounded model checking can be recursively separated by the same way so that an arbitrary number of separation can be applied. Symbolic simulation technique generates a set of multiple counter examples on the same control path with the original counter example. It makes connection of separated bounded model checkings much more efficient. A strategy how to separate a large bound into small pieces was also proposed, and it enables large bound verification. Experimental results showed that the proposed method can actually accelerate verification with bounded model checking.

In Chapter 4, an equivalence checking method which separates control and data portions in designs as preprocess was proposed. In the proposed method, data portions of two designs are forcibly synthesized to be identical. Therefore the equivalence checkings of data portions and control portions can be completely separated, and the effort of the equivalence checking of data portions can be dramatically reduced. The separation also enables word-level equivalence checking of the control portions in bit-level accuracy. Existing symbolic simulation based equivalence checking methods can be used for the word-level equivalence checking of the control portions. A rule-based symbolic equivalence checking method for FSMD was also proposed as a complement of symbolic simulation based methods, and experimental results showed that the proposed method could actually verify designs having complicated control flows faster than a symbolic simulation based method.

In Chapter 5, a method to apply formal verification efficiently to hardware/software co-design written in program code and RTL code was proposed. Formal verification methods including two verification methods proposed in Chapter 3 and Chapter 4 can be directly applied to system-level design, software program code, behavioral hardware code, or RTL hardware code. In addition to those designs, the proposed method enables to apply such formal verification methods to hardware/software co-designs. In the proposed method, communications between hardware and software portions by memory mapped I/O are abstracted to accesses to shared variables and polling loops. Communications through interruption are also modeled by generating an additional concurrent process for each interruption sequence. Then, hardware portions, software portions, and interruption sequences are translated into FSMDs that run in parallel. Since FSMD is in the same level as RTL, many formal verification methods for RTL designs can be applied directly. In addition, the method applies a strong reduction technique based on sequentialization to the generated FSMDs. The sequentialization method generates conditions about execution timings and orders, and SMT solvers check the satisfiability of those conditions. Different from the existing sequentialization methods, the proposed method can check all race condition candidates at once, and generate all execution orders potentially giving different execution results. The numbers of states in the sequentialized designs can be reduced by applying state merging technique with dependence analysis. With the methods, much larger designs can be verified, and it was confirmed by the experiments.

In Chapter 6, ExSDG, a system dependence graph based intermediate representation for high-level designs, including system-level, behavioral level, RTL, and program code was proposed. It can avoid much effort to implement verification and analysis tools for multiple design languages by translating designs into ExSDGs. Then, those tools need to be developed only for ExSDG designs. In ExSDG, syntax tree and some graphs, such as control flow graph and system dependence graph, are integrated to a simple structure, and user can directly and easily access to the design information. Especially, system dependence graph can be used to make back-end algorithms more efficient, since only problem relevant portions can be extracted from input designs. Preprocess for ExSDG also performs a code simplification which removes some syntax elements which are redundant or difficult to be verified, such as pointers. Then dependency information in ExSDG becomes much simpler than existing ones, and it makes tools using system dependence graph faster. In addition back-end tools only have to handle the simplified designs which results in that makes tool developments much easier. This intermediate representation ExSDG is used as the front-end of a verification and debugging environment FLEC. Many methods have been implemented in the environment, and ExSDG have been contributing to the development of FLEC.

It is highly expected that the verification environment of both hardware and software are much improved by the proposed methods, and it can greatly contribute to the further improvement of embedded system verification.

7.2 Future Work

In this section, possible future works for further improvement from the proposed methods in this thesis are discussed.

- **Finding good intermediate point in reachability analysis.** As discussed in Section 3.7, it is very important to find good intermediate states which are probably reachable from the initial state in the multi-level bounded model checking method. One possible approach is to ask users the way to go. This idea assumes that users know the design under verification well, and they can give good suggestions on reachability analysis. [98] proposes a framework that users can give suggestions for reachability analysis with conditions to switch searching methods. The initial brief search is performed by random simulation, and if the given condition is satisfied, the searching method is switched to bounded model checking. This method assumes that users know buggy portions or situations of designs under verification. Another approach is to use distance metrics to guide state space search. [154] proposes a method to guide simulation by means of a distance function derived from the circuit structure. Abstraction guided simulation[178, 131, 129] is a similar approach that compute distances on abstracted pre-images instead of actual designs. It may be a good idea to guide the proposed multi-level bounded model checking by these methods to reach good intermediate points.

- **Additional rules for complicated control flow on the rule-based equivalence checking.** Although the rule-based symbolic equivalence checking method proposed in Section 4.4.7 can compare designs including many control branches and loops faster than symbolic simulation based methods, it cannot verify designs having much different control structures. This can be solved by introduction of new rules for typical control structure differences. Since the method is rule-based, adding new rules is easy, and additional rules only make the performance better.

- **Optimization of the rule based equivalence checking.** The proposed rule-based equivalence checking method in Section 4.4.7 has a room to optimize. First, in the current algorithm shown in Algorithm 9, the explorations of the five rules are done exhaustively. If potentially equivalent states or sequences of state transitions can be determined by some sort of simulations, then the search domains can be reduced drastically since the rules will be tried only to potential equivalence candidates. Second, utilizing internal equivalent points can improve the verification speed. In [46], a cut-point insertion method for equivalence checking between designs before and after high-level synthesis is proposed. Such kind of techniques can be utilized to reduce the search space of equivalent candidates.

- **Handling synchronizations under control branch on sequentialization.** As discussed in Section 5.3.7, the proposed sequentialization cannot handle synchronization points under control branches. Model checking method is required to completely detect such synchronization points, which needs a lot of computation time if synchronization structures are complicated. Therefore a tradeoff between the variety of detectable synchronization points and acceptable computation time should be considered. Utilizing such synchronization points in sequentialization is also not easy since the conditions to synchronize on those synchronization points must be included in the condition whose satisfiability is going to be checked. This is another tradeoff between the number of synchronization points under consideration and the satisfiability checking time.

- **Support of general RTL designs in ExSDG.** The current ExSDG syntax has some limitations on translating RTL designs. For example, it cannot handle multiple clocks, don't cares, high-impedance values. Therefore, to modify the RTL syntax of ExSDG and consider how to translate such RTL designs into TBL can be a good future work of the research. Here some directions to handle them are shown. To translate RTL designs having multi clock domains into TBL, the least common multiple of the clock periods can be considered as a true period. Since even TBL does not have a notion of don't care values, it may be required to add don't care to TBL syntax. In this case, back-end verification tools must also support don't cares. High-impedances can be considered as explicit expressions representing that other parallel modules assign values to the variables to which the high-impedances are assigned. Therefore, they must be checked in advance at a step which eliminates high-impedances in the translation from RTL to TBL.

Bibliography

[1] A. Abdi and D. Gajski. Functional validation of system level static scheduling. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 542–547, March 2005.

[2] D. Ando, T. Nishihara, T. Matsumoto, and M. Fujita. Performance estimation considering false-paths for system-level design (in japanese). *Technical report of IEICE*, 107(507):49–54, March 2008.

[3] D. Ando, T. Nishihara, T. Matsumoto, and M. Fujita. Performance estimation with automatic false-path detection for system-level designs. In *Proc. of International Workshop on Logic and Synthesis*, pages 15–22, June 2008.

[4] D. Ando, T. Nishihara, S. Sasaki, T. Matsumoto, and M. Fujita. Design error detection in system-level designs by dependence analysis and formal checker. In *Proc. of the International Workshop of Logic and Synthesis*, pages 255–262, June 2006.

[5] Z. Andraus, M. Liffiton, and K. Sakallah. Refinement strategies for verification methods based on datapath abstraction. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 19–24, January 2006.

[6] Z. Andraus, M. Liffiton, and K. Sakallah. Reveal: A formal verification tool for verilog designs. In *Proc. of International Conferences on Logic for Programming Artificial Intelligence and Reasoning*, volume 5330 of *Lecture Notes in Computer Science*, pages 343–352, November 2008.

[7] Z. S. Andraus and K. A. Sakallah. Automatic abstraction and verification of verilog models. In *Proc. of Design Automation Conference*, volume 41, pages 218–223, June 2004.

[8] R. Armoni, L. Fix, R. Fraera, S. Huddleston, N. Piterman, and M. Y. Vardi. SAT-based induction for temporal safety properties. In *Proc. of International Workshop on Bounded Model Checking*, volume 119(2) of *Electronic Notes in Theoretical Computer Science*, pages 3–16, March 2005.

[9] R. Arora and M. S. Hsiao. Enhancing SAT-based equivalence checking with static logic implications. In *Proc. of IEEE International Workshop on High-Level Design Validation and Test*, pages 63–68, November 2003.

[10] P. Ashar, A. Gupta, and S. Malik. Using complete-1-distinguishability for fsm equivalence checking. *ACM Trans. on Design Automation of Electronic Systems*, 6(4):569–590, oct 2001.

[11] P. Ashar, A. Raghunathan, A. Gupta, and S. Bhattacharya. Verification of scheduling in the presence of loops using uninterpreted symbolic simulation. In *Proc. of International Conference on Computer Design*, pages 458–466, October 1999.

[12] M. Awedh and F. Somenzi. Proving more properties with bounded model checking. In *Proc. of the International Conference on Computer-Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 96–108, July 2004.

[13] D. Babic and A. J. Hu. Exploiting shared structure in software verification conditions. In *International Haifa Verification Conference*, volume 4899, pages 169–184, October 2007.

[14] D. Babic and A. J. Hu. Calysto: Scalable and precise extended static checking. In *International Conference on Software Engineering*, pages 211–220, May 2008.

[15] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *Proc. of International Conference of Tools and Algorithms for Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 388–403, March 2004.

[16] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c program. In *Proc. of ACM Conference on Programming Language Design and Implementation*, volume 36(5) of *ACM SIGPLAN Notices*, pages 203–213, May 2001.

[17] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. *ACM Trans. on Programming Languages and Systems*, 27(2):314–343, March 2005.

[18] M. Benedetti and S. Bernardini. Incremental compilation-to-SAT procedures. In *Proc. of International Conference on Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 46–58, May 2004.

[19] M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Pro. of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33, April 2003.

[20] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for VLSI circuits. In *Proc. of International Conference on Computer-Aided Design*, pages 456–459, November 1989.

[21] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9:505–525, October 2007.

[22] A. Biere, A.Cimatti, E. M. Clarke, and Y.Zhu. Symbolic model checking without bdds. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, March 1999.

[23] P. Bjesse and K. Claessen. Sat-based verification without state space traversal. In *Proc. of FMCAD*, pages 372–389, nov 2008.

[24] Boolector. http://fmv.jku.at/boolector/.

[25] D. Brand. Verification of large synthesized designs. In *Proc. of the International Conference on Computer-Aided Design*, pages 534–537, November 1993.

[26] J. Burch and D. Dill. Automated verification of pipelined microprocessor control. In *Proc. of International Conference on Computer-Aided Verification*, pages 68–80, June 1994.

[27] J. R. Burch and V. Singhal. Tight integration of combinational verification methods. In *Proc. of International Conference on Computer-Aided Design*, pages 570–576, November 1998.

[28] CatapultC. http://www.mentor.com/products/esl/
high_level_synthesis/catapult_synthesis/.

[29] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, , and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of the International Conference on Computer-Aided Verification*, pages 359–364, July 2002.

[30] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. of the International Conference of Computer-Aided Verification*, pages 154–169, July 2000.

[31] E. Clarke, D. Kroening, and L. Flavio. A tool for checking ANSI-C programs. In *Proc. of the International Conference on Tools and Algorithms for the Constraction and Analysis of Systems*, pages 168–176, March 2004.

[32] E. Clarke, D. Kroening, J. Ouaknine, and O. Strichman. Completeness and complexity of bounded model checking. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation*, pages 96–108, January 2004.

[33] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proc. of the Design Automation Conference*, pages 368–371, June 2003.

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

[35] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Proc. of the Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[36] E.M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing for VHDL. *International Journal on Software Tools for Technology Transfer*, 4(1):125–137, October 2002.

[37] CodeSurfer. http://www.grammatech.com/products/codesurfer/.

[38] COINS Compiler Infrastructure. http://www.coins-project.org/international/.

[39] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu abd Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. of International Conference on Software Engineering*, pages 439–448, June 2000.

[40] CVC3. http://www.cs.nyu.edu/acsys/cvc3/.

[41] L. de Moura, H. Rueb, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Proc. of International Conference on Computer-Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, July 2003.

[42] S. Disch and C. Schollm. Combinational equivalence checking using incremental SAT solving, output ordering, and resets. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 938–943, January 2007.

[43] R. Drechsler. Towards formal verification on the system level. In *Proc. of International Workshop on Rapid System Prototyping*, pages 2–5, June 2004.

[44] N. Een and N. Sorensson. Temporal induction by incremental SAT solving. In *In Proc. of International Workshop on Bounded Model Checking*, volume 89(4) of *Electronic Notes in Theoretical Computer Science*, pages 543–560, July 2003.

[45] E. A. Emerson. Temporal and model logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 996–1072, 1990.

[46] X. Feng and A. J. Hu. Early cutpoint insertion for high-level software vs. rtl formal combinational equivalence verification. In *Proc. of Design Automation Conference*, pages 1063–1068, July 2006.

[47] Formality. http://www.synopsys.com/products/
verification/verification.html.

[48] M. Fujita. On equivalence checking between behavioral and RTL descriptions. In *Proc. of IEEE International Workshop on High-Level Design Validation and Test*, pages 179–184, November 2004.

[49] M. Fujita. Behavior-rtl equivalence checking based on data transfer analysis with virtual controllers and datapaths. In *Proc. of Conference on Correct Hardware Design and Verification Methods*, volume 3275 of *Lecture Notes in Computer Science*, pages 340–345, October 2005.

[50] M. Fujita. Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths. *ACM Trans. on Design Automation of Electronic Systems*, 10(4):610–626, October 2005.

[51] M. Fujita, Y. Kojima, T. Matsumoto, T. Nishihara, and D. Ando. Static checking and formal verification using ExSDGs for reliable system-level SoC designs. In *Proc. of International Symposium on Secure-Life Electronics*, pages 441–447, March 2008.

[52] M. Fujita, S. Shankar, and S. Shunsuke. Equivalence checking: A rule-based approach. In *Proc. of the International Conference on Formal Methods and Models for Co-Design*, July 2006.

[53] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publisher, 1992.

[54] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S.Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publisher, 2000.

[55] P. Godefroid. Partial-order methods for the verification of concurrent systems: An approach to the state-explosion problem. *Lecture Notes in Computer Science*, 1032, January 1996.

[56] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[57] E. Goldberg, M. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 114–121, March 2001.

[58] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Proc. of the International Conference on Computer-Aided Verification*, pages 72–83, June 1997.

[59] D. Grobe and R. Drechsler. Formal verification of ltl formulas for systemc designs. In *Proc. of the International Symposium on Circuits and Systems*, pages 245–248, May 2003.

[60] D. Grobe and R. Drechsler. Checkers for systemc designs. In *Proc. of the International Conference on Formal Methods and Models for Codesign*, pages 171–178, June 2004.

[61] A. Groce and W. Visser. Heuristic model checking for Java programs. In *Proc. of International Workshop on SPIN Model Checking*, volume 2318 of *Lecture Notes in Computer Science*, pages 242–245, April 2002.

[62] A. Groce and W. Visser. Heuristics for model checking Java programs. *International Journal on Software Tools for Technology Transfer*, 6(4), December 2004.

[63] Accellera Working Group. RTL semantics draft specification. `http://www.eda.org/alc-cwg/cwg-open.pdf`.

[64] W. Gunther, A. Hett, and B. Becker. Application of linearly transformed bdds in sequential verification. In *Proc. of ASP-DAC*, pages 91–96, jan 2001.

[65] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *Proc. of International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 354–371, November 2000.

[66] H. Harada, T. Nishihara, T. Matsumoto, and M. Fujita. Debugging support for synchronization of parallel execution in system level designs (in japanese). *Technical report of IEICE*, 108(463):37–42, March 2009.

[67] R. H. Hardin, Z. Har'El, and R. P. Kurshan. Cospan. In *Proc. of International Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427, July 1996.

[68] K. Heljanko, T. Junttila, and T. Latvala. Incremental and complete bounded model checking for full PLTL. In *Proc. of the International Conference on Computer-Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111, July 2005.

[69] T. A. Henzinger, R. Jhala, and R. Majumdar. Lazy abstraction. In *Proc. of Principles of Programming Languages*, volume 29, pages 58–70, January 2002.

[70] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, June 2001.

[71] P. H. Ho, T. Shiple, K. Harer, J Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *Proc. of International Conference on Computer-Aided Design*, pages 120–126, November 2000.

[72] A. Hoffman, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 760–765, March 2001.

[73] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[74] G. J. Holzmann and M. H. Smith. A practical method for the verification of event driven systems. In *Proc. of International Conference on Software Engineering*, pages 597–608, May 1999.

[75] S. Horwits, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programing Languages and Systems*, 12(1):26–60, January 1990.

[76] S. Y. Huang, K. T. Cheng, and K. C. Chen. Aquila: An equivalence verifier for large sequential circuits. In *Proc. of ASP-DAC*, pages 455–460, jan 1997.

[77] S. Y. Huang, K. T. Cheng, and K. C. Chen. Verifying sequential equivalence using atpg techniques. *ACM Trans. on Design Automation of Electronic Systems*, 6(2):244–275, apr 2001.

[78] S. Y. Huang, K. T. Cheng, K. C. Chen, and U. Glaser. An atpg-based framework for verifying sequential equivalence. In *Proc. of International Test Conference*, pages 865–874, oct 1996.

[79] W. Huang, P. Tang, and M. Ding. Sequential equivalence checking using cuts. In *Proc. of ASP-DAC*, pages 455–458, jan 2005.

[80] S. Ichinose and H. Yasuura M. Iwaihara. Program slicing on vhdl descriptions and its evaluation. *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E81-A(12):2585–2594, December 1998.

[81] S. Ichinose, M. Nomura, and H. Yasuura M. Iwaihara. Slicing algorithm for delta-delay vhdl descriptions (in Japanese). *IPSJ SIG Notes*, 95(119):43–50, December 1995.

[82] IEEE 1800, SystemVerilog. http://www.systemverilog.org/.

[83] IEEE 1850, Property Specification Language. http://www.eda-stds.org/ieee-1850/.

[84] M. Iwahara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on vhdl descriptions and its applications. In *In Proc of Asian Pasific Conference on Hardware description Language*, pages 132–139, January 1996.

[85] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. Word-level predicate-abstraction and refinement techniques for verifying RTL Verilog. In *Proc. of Design Automation Conference*, volume 42, pages 445–450, June 2005.

[86] H. Jain, D. Kroening, N. Sharygina, and E. M. Clarke. VCEGAR: Verilog counter example guided abstraction refinement. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13, pages 583–586, March 2007.

[87] J. H. R. Jiang and W. L. Hung. Inductive equivalence checking under retiming and resynthesis. In *Proc. of ICCAD*, pages 326–333, nov 2007.

[88] S. Jiang and R. Kumar. Supervisory control of discrete event systems with CTL* temporal-logic specifications. In *Proc. of IEEE Conference on Decision and Control*, volume 5, pages 4122–4127, December 2001.

[89] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In *Proc. of International Workshop on Bounded Model Checking*, volume 119(2) of *Electronic Notes in Theoretical Computer Science*, pages 51–65, March 2005.

[90] C. Karfa, M. Mandal, D. Sarkar, S. R. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. In *Proc. of the International Symposium on Quality Electronic Design*, pages 110–115, March 2006.

[91] C. Karfa, D. Sarkar, C. Mandel, and P. Kumar. An equivalence checking method for scheduling verification in high-level synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):556–569, March 2008.

[92] Z. Khasidashvili, J. Moondanos, and Z. Hanna. Trans: Efficient sequential verification of loop-free circuits. In *Proc. of HLDVT*, pages 115–120, oct 2002.

[93] Z. Khasidashvili, J. Moondanos, D. Kaiss, and Z. Hanna. An enhanced cut-points algorithm in formal equivalence verification. In *Proc. of IEEE International Workshop on High-Level Design Validation and Test*, pages 171–176, December 2001.

[94] Z. Khasidashvili, M. Skaba, D. Kaiss, and Z. Hanna. Theoretical framework for compositional sequential hardware equivalence verification in presence of design. In *Proc. of ICCAD*, pages 58–65, nov 2004.

[95] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. of International Conference of Tools and Algorithms for Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568, April 2003.

[96] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), July 1976.

[97] A. Koelbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, December 2005.

[98] Y. Kojima, T. Nishihara, T. Matsumoto, and M. Fujita. An interactive verification and debugging environment by concrete/symbolic simulations for system-level designs. In *Proc. of Asian Test Symposium*, pages 315–320, November 2008.

[99] Y. Kojima, T. Nishihara, T. Matsumoto, and M. Fujita. A technique of automatic input pattern generation for system-level design descriptions by concrete and symbolic simulations. *Technical report of IEICE*, 107(559):49–54, March 2008.

[100] Y. Kojima, T. Nishihara, T. Matsumoto, and M. Fujita. FLEC: A framework for system-level debugging support, formal verification and static analysis. In *Proc. of International Workshop on Synthesis and System Integration of Mixed Information Technologies*, pages 341–346, March 2009.

[101] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs.* PhD thesis, Universitat Passau, 2003.

[102] D. Kroening, E. M. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proc. of Design Automation Conference*, pages 368–371, June 2003.

[103] A. Kuehlmann and C. A. J. van Eijk. *Logic Synthesis and Verification*, chapter 13, pages 343–372. Kluwer Academic Publishers, 2002.

[104] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Static partial order reduction. In *Proc. of International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357, March 1998.

[105] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Autometa-Theoretic Approach.* Princeton University Press, 1994.

[106] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *Proc. of International Conference on Computer-Aided Verification*, volume 15, pages 141–153, July 2003.

[107] F. Lerda, N. Sinha, and M. Theobald. Symbolic model checking of software. *Electr. Notes Theor. Comput. Sci.*, 89(3), November 2003.

[108] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Proc. of International Workshop on SPIN Model Checking*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102, May 2001.

[109] J. Li, X. Sun, F. Xie, and X. Song. Component-based abstraction and refinement. In *Proc. of International Conference on Software Reuse*, pages 39–51, May 2008.

[110] N. Liveris, H. Zhou, and P. Banerjee. Complete-k-distinguishability for retiming and resynthesis equivalence checking without restricting synthesis. In *Proc. of ASP-DAC*, pages 636–641, jan 2009.

[111] F. Lu, M. K. Iyer, G. Parthasarathy, L. C. Wang, and K. T. Cheng. An efficient sequential sat solver with improved search strategies. In *Proc. of DATE*, pages 1102–1107, mar 2005.

[112] T. Matsumoto, S. Komatsu, and M. Fujita. An approach to equivalence checking by symbolic simulation between behavioral and RTL designs (in Japanese). *Technical report of IEICE*, 106(32):7–12, May 2006.

[113] T. Matsumoto, H. Saito, and M. Fujita. Equivalence checking of c programs by locally performing symbolic simulation on dependence graphs. In *Proc. of the International Symposium on Quality of Electronic Design*, pages 370–375, March 2006.

[114] T. Matsumoto, T. Sakunkonchak, H. Saito, S. Komatsu, and M. Fujita. An equivalence checking method for system-level designs under different schedulings applying sequentialization with ilp solvers (in Japanese). In *Proc. of Design Automation Symposium*, pages 157–162, July 2006.

[115] T. Matsumoto, K. Seto, and M. Fujita. Formal equivalence checking for loop optimization in C programs without unrolling. In *Proc. of IASTED International Conference on Advances in Computer Science and Technology*, pages 43–48, April 2007.

[116] Y. Matsunaga. An efficient equivalence checker for combinational circuits. In *Proc. of Design Automation Conference*, pages 629–634, June 1996.

[117] K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[118] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proc. of International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264, July 2002.

[119] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. of International Conference on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13, July 2003.

[120] A. Mishchenko, S. Chatterjee, and R. Brayton. Fraigs: A unifying representation for logic synthesis and verification. Technical report, University of California, Berkeley, March 2005.

[121] I. H. Moon. Compositional verification of retiming and sequential optimization. In *Proc. of DAC*, pages 131–136, jun 2008.

[122] I. H. Moon, P. Bjesse, and C. Pixley. A compositional approach to the combination of combinational and sequential equivalence checking of circuits without known reset states. In *Proc. of DATE*, pages 1–6, apr 2007.

[123] M. Moy, F. Maraninchi, and L. M. Contoz. Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In *Proc. of the International Conference on Application of Concurrency to System Design*, pages 26–35, June 2005.

[124] M. Nomura and M. Iwaihara. Program slicing on hardware descriptions and its application to design verification (in Japanese). *IPSJ SIG Notes*, 95(54):7–14, May 1995.

[125] Open Core Protocol. http://www.ocpip.org/.

[126] OpenAccess Gear. http://openedatools.si2.org/oagear/.

[127] J. S. Ostroff and W. M. Wonham. A framework for real-time discrete event control. *IEEE Trans. on Automatic Control*, 35(4):386–397, 1990.

[128] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGSOFT Software Engineering Notes*, 9:177–184, 1984.

[129] A. Parikh and M. S. Hsiao. On dynamic switching of navigation for semi-formal design validation. In *Proc. of IEEE International Workshop on High-Level Design Validation and Test*, pages 41–48, November 2008.

[130] C. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *Proc. of International Workshop on SPIN Model Checking*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181, April 2004.

[131] F. M. D. Paula and A. J. Hu. An effective guidance strategy for abstruction-guided simulation. In *Proc. of the Design Automation Conference*, pages 63–68, June 2007.

[132] Pinapa: A SystemC front-end. http://greensocs.sourceforge.net/pinapa/.

[133] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton. Using combinational verification for sequential circuits. In *Proc. of DATE*, pages 138–144, may 1999.

[134] S. Reda and A. Salem. Combinational equivalence checking using Boolean satisfiability and binary decision diagrams. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 122–126, March 2001.

[135] S. M. Reddy, K. Kunz, and D. K. Pradhan. Novel verifcation framework combining structural and OBDD methods in a synthesis environment. In *Proc. of Design Automation Conference*, pages 414–419, June 1994.

[136] M. Reshadi and D. Gajski. A cycle-accurate compilation algorithm for custom pipelined datapaths. In *Proc. of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 21–26, September 2005.

[137] G. Ritter. *Formal Sequential Equivalence Checking of Digital Systems by Symbolic Simulation.* PhD thesis, Darmastadt University of Technology and Universite Joseph Fourier, 2000.

[138] G. Ritter. Sequential equivalence checking by symbolic simulation. In *International Conference on Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 423–442, November 2000.

[139] G. Ritter, H. Eveking, and H. Hinrichsen. Formal verification of designs with complex control by symbolic simulation. In *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 238–250, September 1999.

[140] G. Ritter, H. Hinrichsen, and H. Eveking. Formal verification of descriptions with distinct order of memory operations. In *Asian Computing Science Conference*, volume 1742 of *Lecture Notes in Computer Science*, pages 790–791, December 1999.

[141] J. Rowson. Virtual synchronization for fast distributed cosimulation of dataflow task graphs. In *Proc. of International Symposium on System Synthesis*, pages 174–179, October 2002.

[142] M. J. Balcer S. J. Mellor. *Executable UML*. Addison-Wesley, 2002.

[143] H. Saito, T. Ogawa, T. Sakunkonchak, M. Fujita, and T. Nanya. An equivalence checking methodology for hardware oriented c-based specification. In *Proc. of International Workshop on High Level Design Varidation and Test*, pages 139–144, October 2002.

[144] T. Sakunkonchak, S. Komatsu, and M. Fujita. Synchronization verification in system-level design with ilp solvers. In *Proc. of the Formal Methods and Models for Co-Design*, pages 121–130, July 2005.

[145] T. Sakunkonchak, T. Matsumoto, H. Saito, S. Komatsu, and M. Fujita. Equivalence checking in c-based system-level design by sequentializing concurrent behaviors. In *Proc. of IASTED International Conference on Advances in Computer Science and Technology*, pages 36–42, April 2007.

[146] S. Sasaki, T. Nishihara, and M. Fujita. Slicing-based hardware/software co-design methodology from functional specifications. In *IPM International Workshop on Foundations of Software Engineering*, volume 159 of *Electronic Notes in Theoretical Computer Science*, pages 265–280, May 2006.

[147] P. Schaumont, S. Shukla, and I. Verbauwhede. Design with race-free hardware semantics. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 571–576, April 2006.

[148] Seamless. http://www.mentor.com/products/fv/
hwsw_coverification/seamless/.

[149] L. Semeria and A. Ghosh. Methodology for hardware/software co-verification in c/c++. In *Proc. of the Asia and South Pacific Design Automation Conference*, pages 405–408, January 2000.

[150] L. Semeria, A. Seawright, R. Mehra, D. Ng, A. Ekanayake, and B. Pangrle. C-based RTL methodology for designing and verifying a multi-thread processor. In *Proc. of the Design Automation Conference*, pages 123–128, June 2002.

[151] S. Shankar and M. Fujita. Rule-based approaches for equivalence checking of SpecC programs. In *Proc. of IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pages 39–48, June 2008.

[152] K. C. Shashidhar, M. Bruynooghe, F. Catthoor, and G. Janssens. Functional equivalence checking for verification of algebraic transformations on array- intensive source code. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 1310–1315, March 2005.

[153] M. Sheeran, S. Singh, , and G. Stalmarck. Checking safety properties using induction and a sat-solver. In *Proc. of the International Conference on Formal Methods in Computer Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125, November 2000.

[154] S. Shyam and V. Bertacco. Distance-guided hybrid verification with GUIDO. In *Proc. of Conference on Design, Automation and Test in Europe*, pages 1211–1216, April 2006.

[155] SLEC RTL. http://www.calypto.com/products/
SLEC_RTL.html.

[156] SLEC SYSTEM. http://www.calypto.com/products/
SLEC_SYSTEM.html.

[157] D. Stoffel and W. Kunz. Record & play : A structural fixed point iteration for sequential circuit verification. In *Proc. of ICCAD*, pages 294–399, nov 1997.

[158] D. Stoffel, M. Wedler, P. Warketin, and W. Kunz. Structural fsm traversal. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(5):598–619, may 2004.

[159] O. Strichman. Pruning techniques for the SAT-based bounded model checking problem. In *In Proc. of Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 58–70, September 2001.

[160] SystemC. http://www.systemc.org/.

[161] K. Tanabe, H. Saito, S. Komatsu, and M. Fujita. A program slicing for system level description written in specc language (in Japanese). *IEICE Technical Report*, 103(704):79–84, March 2004.

[162] K. Tanabe, S. Sasaki, and M. Fujita. Program slicing for system level designs in specc. In *Proc. of International Conference on Advances in Computer Science and Technology*, pages 252–258, November 2004.

[163] University of California, Irvine. NISC Technology. http://www.cecs.uci.edu/ nisc/.

[164] The LLVM Compiler Infrastructure. http://llvm.org/.

[165] The SUIF Compiler Group. http://suif.stanford.edu/.

[166] A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1(4):297–322, December 1992.

[167] C. A. J. van. Eijk. Sequential equivalence checking without state space traversal. In *Proc. of DATE*, pages 618–623, feb 1998.

[168] C. A. J. van. Eijk. Sequential equivalence checking based on structural similarities. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 19(7):814–819, jul 2000.

[169] Verific HDL Component Software. http://www.verific.com/products.html.

[170] I. Viskic and R. Domer. A flexible, syntax independent representation (sir) for system level design models. In *Proc. of EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools*, pages 288–294, October 2006.

[171] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.

[172] K. Wakabayashi and T. Okamoto. C-based SoC design flow and EDA tools : an ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1507–1522, December 2000.

[173] M. Weiser. Program slicing. In *Proc. of International Conference on Software Engineering*, pages 439–449, March 1981.

[174] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, July 1984.

[175] J. Whittemore, J. Kim, and K. A. Sakallah. Satire: A new incremental satisfiability engine. In *Proc. of Design Automation Conference*, pages 542–545, June 2001.

[176] F. Xie, X. Song, H. Chung, and R. Nandi. Translation-based co-verification. In *Proc. of the International Conference on Formal Methods and Models for Codesign*, pages 111–120, July 2005.

[177] A. Yamada, K. Nishida, R. Sakurai, A. Kay, T. Nomura, and T. Kambe. Hardware synthesis with the bach system. In *Proc. of IEEE International Symposium on Circuits and Systems*, volume 6, pages 366–369, July 1999.

[178] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of the Design Automation Conference*, pages 599–604, June 1998.

[179] Yices. http://yices.csl.sri.com/.

[180] H. Yoshida and M. Fujita. Improving the accuracy of rule-based equivalence checking of high-level descriptions by identifying potential internal equivalences (in Japanese). *Technical report of IEICE*, 108(298):109–114, November 2008.

[181] H. Yoshida and M. Fujita. Improving the accuracy of rule-based equivalence checking of system-level design descriptions by identifying potential internal equivalences. In *Proc. of IEEE International Symposium on Quality Electronic Design*, pages 366–370, March 2009.

[182] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *International Conference on Computer Aided Design*, pages 279–285, November 2001.

[183] V. Zivojnovic and H. Meyr. Compiled hw/sw co-simulation. In *Proc. of the Design Automation Conference*, pages 690–695, June 1996.

Publications

Journal

1. T. Nishihara, T. Matsumoto, M. Fujita, “Word-Level Equivalence Checking in Bit-Level Accuracy by Synthesizing Designs onto Identical Datapath,” *IEICE Transaction on Information and Systems*, Vol E92-D, No. 5, pp. 972–984, May 2009.

2. S. Sasaki, T. Nishihara, D. Ando, and M. Fujita, “Hardware/Software Co-design and Verification Methodology from System Level Based on System Dependence Graph,” *Journal of Universal Computer Science*, Vol. 13, No. 13, pp. 1972–2001, 2007.

3. T. Nishihara, T. Matsumoto, S. Komatsu, and M. Fujita, “Formal Verification of Hardware/Software Co-designs with Translation into Representations in State Transitions,” *Electronics and Communications in Japan*, Part 2 Electronics, Vol. 9, No. 7, pp.11–19, July 2007.

4. T. Nishihara, T. Matsumoto, S. Komatsu, and M. Fujita, “Formal Verification of Hardware/Software Co-designs with Translation into Representations in State Transitions (in Japanese),” *IEICE Transactions on Information and Systems (Japanese Edition)*, Vol. J89-D, No. 4, pp.651–659, April 2006.

International Conference

5. Y. Kojima, T. Nishihara, T. Matsumoto, and M. Fujita, “FLEC: A Framework for System-level Debugging Support, Formal Verification and Static Analysis,” In *Proc. of Workshop on Synthesis and System Integration of Mixed Information Technologies*, pp.341–346, Mar. 2009.

6. T. Nishihara, T. Matsumoto, and M. Fujita, “Multi-Level Bounded Model Checking to Detect Bugs Beyond the Bound,” In *Proc. of IEEE International Workshop on High Level Design Validation and Test*, pp.49–55, Nov. 2008.

7. T. Nishihara, T. Matsumoto, and M. Fujita, “Bounded Model Checking of RTL Designs with Property Decomposition and Refinement,” In *Proc. of GCOE 2007 Workshop between National Chiao Tung University & The University of Tokyo*, pp.121–125, Dec. 2007.

8. T. Nishihara, D. Ando, T. Matsumoto, and M. Fujita, “ExSDG : Unified Dependence Graph Representation of Hardware Design from System Level down to RTL for Formal Analysis and Verification,” In *Proc. of International Workshop of Logic and Synthesis*, pp.83–90, May 2007.

9. T. Nishihara, T. Matsumoto, and M. Fujita, “Equivalence Checking with Rule-Based Equivalence Propagation and High-Level Synthesis,” In *Proc. of IEEE International Workshop on High Level Design Validation and Test*, pp.162–169, Nov. 2006. nd Synthesis, pp.255-262, June 2006.

10. S. Sasaki, T. Nishihara, and M. Fujita, “Slicing-based Hardware/Software Co-design Methodology from Functional Specifications,” In *Proc. of IPM International Workshop on Foundations of Software Engineering, Electronic Notes in Theoretical Computer Science*, Vol. 159, pp.265–280, May 2006.

Domestic Conference

11. T. Nishihara, T. Matsumoto, M. Fujita, “State Reduction Technique on Formal Verification of Hardware/Software Co-Design with Synchronization Point Detection (in Japanese),” In *Proc. of Design Automation Symposium*, pp.49–54, Aug. 2009.

12. T. Nishihara, T. Matsumoto, M. Fujita, “Multi-Level Bounded Model Checking to Detect Bugs Beyond the Bound (in Japanese),” In *Proc. of Design Automation Symposium*, pp.121–126, Aug. 2008.

13. T. Nishihara, T. Matsumoto, M. Fujita, “Equivalence Checking with Datapath Abstraction Using High-Level Synthesis (in Japanese),” In *Proc. of Design Automation Symposium*, pp.151–156, July 2006.

14. T. Nishihara, T. Matsumoto, M. Fujita, “Verifying Deep Bugs by Model Checking with Inductive Approach (in Japanese),” *IEICE Technical Report*, Vol. 105, No. 644, pp.1–6, Mar. 2006.

15. T. Nishihara, T. Matsumoto, S. Komatsu, M. Fujita, “Formal Verification of HW/SW Co-design with FSM Translation (in Japanese),” *IPSJ Technical Report*, Vol. 2005, No. 27, pp.37–42, Mar. 2005.